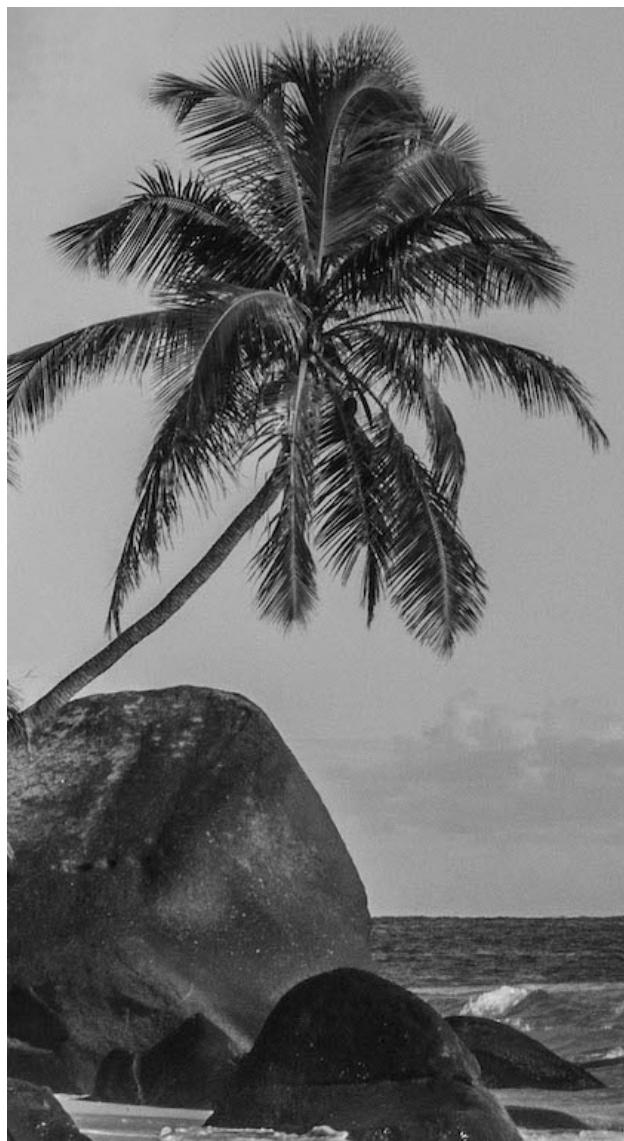


# Computing platforms

A.Shafarenko and S.P.Hunt



School of Computer Science  
University of Hertfordshire  
2017

# Contents

<b>1 About the Book</b>	<b>1</b>
1.1 Platforms . . . . .	1
1.2 Data manipulation . . . . .	1
1.3 The CdM-8 platform . . . . .	1
1.4 Programming CdM-8 Platform 3 <sup>1/2</sup> . . . . .	2
1.5 Circuits and Universal Platform 1 . . . . .	3
<b>2 About the Subject</b>	<b>4</b>
2.1 Tanenbaum's classification of platforms . . . . .	4
2.2 Every platform has a language . . . . .	5
2.2.1 A programmable platform has a programming language . . . . .	5
2.2.2 A “language” of circuits? . . . . .	5
2.3 All platforms ‘compute’ . . . . .	6
2.3.1 Hardware, software and virtual machines . . . . .	6
2.4 Translating between Tanenbaum Levels . . . . .	7
2.4.1 Aggregation . . . . .	7
2.4.2 Interpretation . . . . .	7
2.4.3 Compilation . . . . .	8
2.5 The Tanenbaum Levels . . . . .	9
2.5.1 Level 0 . . . . .	9
2.5.2 Level 1 . . . . .	9
2.5.3 Level 2 . . . . .	9
2.5.4 CdM-8 Platform 2 . . . . .	9
2.5.5 Level 3 . . . . .	10
2.5.6 Level 3 <sup>1/2</sup> . . . . .	10
<b>3 Representation of Data</b>	<b>12</b>
3.1 Discontinuity . . . . .	12
3.1.1 Computers utilise discrete data items . . . . .	12
3.1.2 Discrete models . . . . .	12
3.1.3 Discrete values are easy to implement . . . . .	13
3.2 Information . . . . .	14

3.2.1	Quantifying information . . . . .	14
3.2.2	Binary strings . . . . .	14
3.2.3	Manipulating data representations . . . . .	16
3.3	Operations on numbers . . . . .	16
3.3.1	Addition of $k$ -bit binary numbers is ineffective, and potentially unsafe . . . . .	17
3.3.2	Using a table to implement an operation . . . . .	18
3.3.3	Pros and cons of tables . . . . .	18
3.4	Positional number systems . . . . .	20
3.4.1	Base ten numbers (Decimal) . . . . .	20
3.4.2	Positional weighting and string length . . . . .	20
3.4.3	Base two numbers (Binary) . . . . .	21
3.4.4	Quantities, numbers and digit strings . . . . .	21
3.4.5	Conversion between decimal and binary number systems . . . . .	22
3.4.6	From decimal to binary . . . . .	22
3.4.7	Binary to decimal . . . . .	23
3.4.8	Base eight numbers (Octal) . . . . .	23
3.4.9	Base sixteen numbers (Hexadecimal) . . . . .	24
3.4.10	Hexadecimal digit strings . . . . .	24
3.4.11	Conversions . . . . .	24
3.4.12	Avoiding confusion between number bases . . . . .	24
3.5	Addition of non-negative numbers . . . . .	26
3.5.1	Digit strings of fixed length . . . . .	26
3.5.2	Adding together positional numbers . . . . .	26
3.5.3	Adding together binary numbers . . . . .	27
3.5.4	Overflow events . . . . .	27
3.5.5	Bit slicing . . . . .	28
3.5.6	Big- and little-endianness . . . . .	29
3.6	Subtraction and negative numbers . . . . .	30
3.6.1	Signed-magnitude representations . . . . .	30
3.6.2	Complement representations of signed numbers . . . . .	30
3.6.3	Ten's complement decimal numbers . . . . .	31
3.6.4	Ten's complement addition and subtraction . . . . .	32
3.6.5	Why ten's complement? . . . . .	33
3.6.6	Overflow events in ten's complement . . . . .	33
3.6.7	Two's complement . . . . .	33
3.6.8	Comparing numbers . . . . .	35
*3.7	Binary-Coded Decimal . . . . .	36
3.8	Multiplication . . . . .	37
3.8.1	Decimal example . . . . .	37
3.8.2	Unsigned binary multiplication . . . . .	38

3.8.3	Multiplication of signed numbers . . . . .	38
3.8.4	Sign extension . . . . .	39
3.8.5	Two's complement multiplication . . . . .	39
3.9	Division . . . . .	41
3.9.1	Division of signed numbers . . . . .	41
3.9.2	The division algorithm . . . . .	41
3.10	Sets . . . . .	44
3.10.1	Set representation . . . . .	44
3.10.2	Set operations . . . . .	45
3.10.3	Bit-slicing sets . . . . .	47
*3.11	Sparse sets . . . . .	48
3.11.1	Representing sparse sets . . . . .	48
3.11.2	Efficiency of list representations . . . . .	50
3.11.3	Operations on sparse sets . . . . .	50
3.12	Representing text . . . . .	56
3.12.1	Character encoding . . . . .	57
3.12.2	Control characters . . . . .	59
3.12.3	Modifier keys . . . . .	60
3.12.4	8 bit character codes . . . . .	60
3.12.5	ASCII objects . . . . .	61
3.12.6	Unicode . . . . .	62
3.13	Addresses . . . . .	66
3.13.1	Representing the location of a data item in memory . . . . .	66
3.13.2	How much data at each location? . . . . .	66
3.13.3	Addresses as numbers . . . . .	67
3.13.4	Referencing and de-referencing . . . . .	67
3.14	Data structures . . . . .	68
3.14.1	Properties of data structures . . . . .	68
3.14.2	Representing data structures . . . . .	69
3.14.3	Granularity . . . . .	70
3.14.4	Alignment . . . . .	71
3.14.5	Arrays . . . . .	71
3.14.6	Array indexing . . . . .	72
3.14.7	Arrays of character strings . . . . .	73
3.14.8	Defining arrays . . . . .	73
3.14.9	Records . . . . .	74
3.14.10	Defining records . . . . .	75
3.14.11	Null-terminated strings vs arrays of characters . . . . .	76
*3.15	Variant records . . . . .	77
3.15.1	Declaring variant record types . . . . .	77

3.16	General data structure types . . . . .	79
3.17	Addresses and pointers . . . . .	80
3.17.1	Address consistency . . . . .	80
3.17.2	Pointers . . . . .	80
3.18	References and linked data structures . . . . .	82
3.18.1	Linked list . . . . .	82
<b>4</b>	<b>Architecture and Operation of Platform 3<sup>1/2</sup></b>	<b>86</b>
4.1	Resources . . . . .	86
4.1.1	Memory . . . . .	86
4.1.2	Registers . . . . .	87
4.1.3	The Program Counter (PC) . . . . .	87
4.1.4	The Stack Pointer (SP) . . . . .	88
4.1.5	The Processor Status register (PS) . . . . .	88
4.1.6	Macro-assembly language . . . . .	89
4.1.7	Software tools . . . . .	89
4.2	How the platform operates . . . . .	90
4.2.1	Initialisation . . . . .	90
4.2.2	Execution . . . . .	90
4.2.3	Level 2 machine instructions . . . . .	91
<b>5</b>	<b>Introduction to Programming CdM-8 Platform 3<sup>1/2</sup></b>	<b>92</b>
5.1	Writing (macro-)assembly language programs . . . . .	92
5.1.1	Instructions . . . . .	92
5.1.2	Pseudo-instructions . . . . .	92
5.1.3	Programs . . . . .	93
5.1.4	<code>halt</code> and <code>end</code> . . . . .	93
5.2	Literals . . . . .	94
5.2.1	Number literals . . . . .	94
5.2.2	Character literals . . . . .	94
5.2.3	String literals . . . . .	94
5.3	Labelling program code . . . . .	95
5.3.1	Introducing labels . . . . .	95
5.4	Making programs understandable . . . . .	97
5.4.1	Comments . . . . .	97
5.4.2	Layout . . . . .	98
5.4.3	Use of ‘white space’ . . . . .	99
5.5	Placing code and data in memory . . . . .	100
5.5.1	Placement of machine instructions and data in memory . . . . .	100
5.5.2	Labelling data locations . . . . .	102
5.5.3	Referring to labels . . . . .	103

5.5.4	Relative addresses and offsets . . . . .	103
5.6	Loading and storing data . . . . .	105
5.6.1	Load Immediate . . . . .	105
5.6.2	Load . . . . .	106
5.6.3	Store . . . . .	107
5.7	The Arithmetic-Logic Unit (ALU) . . . . .	108
5.7.1	Arithmetic operations . . . . .	108
5.7.2	Logic operations . . . . .	110
5.7.3	Data movement operations . . . . .	113
5.8	Controlling the order of execution of instructions . . . . .	116
5.8.1	Conditional execution of a sequence (a simple <code>if</code> ) . . . . .	116
5.8.2	Control conditions . . . . .	117
5.8.3	Conditional execution with an alternative (an <code>if</code> with an <code>else</code> ) . . . . .	117
5.8.4	Complex conditions . . . . .	119
5.8.5	Complex conditions in macro-assembly language programs . . . . .	119
5.8.6	Lazy evaluation of conditions . . . . .	120
5.8.7	Repetition . . . . .	121
5.8.8	Program loops . . . . .	121
5.8.9	The <code>while</code> loop construct . . . . .	121
5.8.10	The <code>until</code> loop construct . . . . .	122
5.8.11	Nesting of control constructs . . . . .	123
5.8.12	Early termination of an iteration . . . . .	123
5.9	Problem solving and complexity . . . . .	125
<b>6</b>	<b>Subroutines, Stack and Heap</b>	<b>126</b>
6.1	Running out of registers... what to do? . . . . .	126
6.2	Subroutines . . . . .	129
6.3	The Stack . . . . .	131
6.3.1	Subroutines and the stack. . . . .	132
6.3.2	Dirty subroutines. . . . .	135
*6.4	Recursion, stack memory and coroutines . . . . .	135
6.4.1	Recursive subroutines . . . . .	135
6.4.2	Subroutine memory: static vs stack. . . . .	139
6.4.3	Passing a subroutine as a parameter . . . . .	146
6.4.4	Coroutines . . . . .	148
*6.5	Heap . . . . .	153
6.5.1	Requirements . . . . .	153
6.5.2	Buddy allocation algorithm . . . . .	154

<b>7 Virtualisation</b>	<b>163</b>
7.1 Object modules and linking . . . . .	163
7.1.1 Relocatable sections . . . . .	164
7.1.2 Linking object files . . . . .	167
7.1.3 Structure of object file . . . . .	168
7.1.4 Memory image file . . . . .	169
7.2 Instruction aggregation . . . . .	170
<b>8 Down the Rabbit Hole</b>	<b>174</b>
8.1 Platform 3 . . . . .	174
8.1.1 ALU instructions . . . . .	174
8.1.2 Control . . . . .	175
8.1.3Nonce and unique . . . . .	179
8.1.4 Control revisited . . . . .	183
8.1.5 Loops . . . . .	187
8.1.6 Less common control structures . . . . .	189
8.2 Platform 2 . . . . .	189
8.2.1 How does the assembler work? . . . . .	192
<b>9 On the ground climbing up: Platform 0</b>	<b>195</b>
9.1 Representing ones and zeros in Platform 0 . . . . .	196
9.1.1 Some definitions . . . . .	196
9.2 Transistor . . . . .	199
*9.3 Transistor imperfections . . . . .	200
<b>10 Universal Platform 1</b>	<b>202</b>
10.0.1 Logic gates . . . . .	202
10.1 NOT-gate . . . . .	202
10.1.1 A brief note about timing considerations . . . . .	203
10.2 NAND-gate . . . . .	204
10.3 NOR-gate . . . . .	205
10.4 XOR-gate . . . . .	206
*10.5 Pass Transistor Logic (PTL) . . . . .	207
10.6 Platform 0 revisited: Pull resistors . . . . .	208
10.7 Controlled buffer and transmission gate . . . . .	210
10.8 Gate symbols and circuit structuring . . . . .	211
10.8.1 Logisim chips . . . . .	211
10.8.2 Logisim tunnels . . . . .	212

<b>11 Platform 1 circuits</b>	<b>214</b>
11.1 Adders . . . . .	214
11.1.1 Half-adder . . . . .	215
11.1.2 Full adder . . . . .	215
11.2 Wire bunches and gate arrays . . . . .	216
11.3 Combinational circuits: general design . . . . .	218
11.3.1 Implementation of truth tables . . . . .	218
11.4 Decoder . . . . .	220
11.5 Multiplexer . . . . .	220
11.6 Sequential circuits: an RS flip-flop . . . . .	221
11.7 Pulses and edges . . . . .	223
11.8 Latches and the Clock . . . . .	225
11.9 Master-Slave Latch . . . . .	227
*11.10A CMOS alternative . . . . .	228
11.11 Multiported register . . . . .	229
11.12 Memory . . . . .	231
11.12.1 Platform 0 revisited. Capacitors as memory: the story of DRAM . . . . .	231
11.12.2 DRAM operation . . . . .	231
11.12.3 Read-Only Memory (ROM) . . . . .	233
11.12.4 Memory as a Platform 1 building block . . . . .	234
<b>12 Architecture of Platform 2</b>	<b>236</b>
12.1 The (high-altitude) eagle view . . . . .	236
12.2 The register file . . . . .	238
12.3 The ALU . . . . .	239
12.4 Example Data Path and Register Transfer Language . . . . .	240
12.4.1 Data path . . . . .	240
12.4.2 Register Transfer Language . . . . .	241
12.4.3 Adding memory . . . . .	242
12.5 Instruction machine . . . . .	244
12.5.1 The Sequencer . . . . .	244
12.5.2 Primary decoder (PD) . . . . .	247
12.5.3 CdM-8 Data Path . . . . .	248
12.5.4 Secondary Decoder . . . . .	252
<b>13 Level 2 Systems Architecture and Input/Output</b>	<b>255</b>
13.1 Processor . . . . .	255
13.2 System View: von Neumann Architecture . . . . .	256
13.3 System View: Harvard Architecture . . . . .	256
13.3.1 Implementing Harvard architecture . . . . .	257
13.3.2 Programming for split memory: the <code>ldc</code> instruction . . . . .	257

13.4	Memory mapped I/O . . . . .	258
13.4.1	Memory-mapped I/O organisation . . . . .	259
13.4.2	CdM-8 I/O bus . . . . .	260
13.4.3	Peripheral interface: example . . . . .	260
13.4.4	Address sharing. . . . .	263
13.5	Interrupts . . . . .	263
13.5.1	Interrupts: a system-level view . . . . .	265
13.5.2	Interrupts from a program's point of view . . . . .	267
13.5.3	A program example . . . . .	269
13.5.4	Four-phase handshake . . . . .	276
13.5.5	Interrupt Master . . . . .	277
13.5.6	title . . . . .	279
13.5.7	Multiple interrupt sources: an example . . . . .	282
<b>14</b>	<b>Full Platform 2: Operating System</b>	<b>287</b>
14.1	Processes . . . . .	287
14.2	The Coccoe machine: an OS demonstrator . . . . .	290
14.2.1	Memory subsystem . . . . .	290
14.2.2	Extended processor . . . . .	294
14.3	OS <i>cocos</i> . . . . .	298
14.3.1	Scheduler . . . . .	298
14.3.2	Drivers . . . . .	301
14.3.3	Shell . . . . .	307
14.3.4	File system . . . . .	308
14.4	<i>cocos</i> system calls . . . . .	310

## Acknowledgements

The sea coconut (coco de mer) palm tree (photo on the front page) under which the idea of the CdM-8 architecture was conceived by Alex Shafarenko, and which provided critical cooling at the conception stage, is gratefully acknowledged. Alex is indebted to Carl Burch, who wrote the Logisim circuit simulator — a free tool, which was invaluable for his implementation of the Level 2 platform, which in turn enabled his work on the programming system and automated assessment software for an undergraduate module of platform architecture. The influence of the School leadership, who did not wish this to be yet another course in architecture, is also acknowledged with heartfelt thanks; the course has turned out to be broader and better than the original plan.

The key role in converting these disparate thoughts and designs into a product was played by other members of the Computing Platforms team (module 4com1042). We are grateful to Bob Dickerson for advancing what we have dubbed the *Dickerson Principle* (start in the middle of the Tanenbaum hierarchy, go down to the transistor level, and up to the OS user interface), which we have enthusiastically embraced. We are indebted to Dr Mick Walters, an HCI expert, who took it upon himself to develop an Integrated Development Environment, which includes a specialised code editor for CdM-8 macroassembler and a very useful and usable front-end for the CdM-8 system-level emulator. Thanks are due to Dr David Bowes who modified the Java code for Logisim to connect it to the Ecosystem's monitoring and examination facility, AMES, which made it possible to provide continuous assessment including circuit questions and exercises. Finally, we acknowledge the work of Dr Raimund Kirner, who checked many online design exercises with their automatic marking procedures and who proposed new ones.

# About the Book

This book is about computing platforms. They will be treated comprehensively: their structure, principles of operation and the ways of using them in programming will all be discussed, as well as practiced in a problem-solving setting. The book is a technological and conceptual journey across the lowlands of the computing world. As with any such journey it is, of necessity, partial (you don't get to see everything in a single trip and you are not climbing any summits either), but it brings about an understanding of what computing is fundamentally based on, which is useful by itself and as part of a broader computer science and IT education.

## 1.1 Platforms

Our first step is to define the subject:

**Definition 1 :** A *computing platform* is a self-sufficient set of resources (hardware, software or both) intended for manipulation of digital data.

The set is required to be *complete*, in the sense that it is capable of performing a defined range of data manipulation tasks all by itself, but a computing platform is *not* isolated. In other words, a computing platform *interacts* with its *environment*. A common feature of all platforms is that they can be configured and/or programmed. At the very least, a platform allows its users to set it up for a specific task. This can be achieved by manipulating the hardware or by modifying stored data (configuration), or by expanding or modifying the software (programming).

## 1.2 Data manipulation

Whilst data manipulation includes computation in the narrow sense (i.e. performing arithmetic operations on numbers represented as digital data), it also includes the processing of non-numerical data (such as changing and combining strings of characters, searching for patterns in a data set, sorting records into alphabetical order, etc). It further includes handling associative data structures, such as relational databases, working with data structures that represent linked data (such as lists, trees or graphs), interacting with the physical environment by sensing and signalling through peripheral devices: familiar things, such as mice, keyboards, monitors, etc., but also less familiar things such as temperature sensors, robot actuators, etc., as well as sending and receiving data over networks and other forms of communication.

## 1.3 The CdM-8 platform

In this book we use, as a running example, a computing platform that was developed specifically for teaching purposes, (CdM-8). However, the messages conveyed here, and the principles you will learn by studying our chosen platform are universal. Despite the narrow focus of the CdM-8 ecosystem, it is a thoroughly modern and fully functional platform, which can, in fact, be used to solve practical problems. It is not a toy system for whetting a learner's appetite for study: it is a grown-up solution which has been drastically reduced in

complexity in order to help learners focus on the essential parts found in *any* sensible platform of the same level of abstraction.

We begin at the top, with an exposition of the highest level platform being dealt with in this book. We call it Platform 3½ because it provides facilities that are half way between the ‘normal’ assembler level and a classical ‘high level’ programmable virtual machine, such as C, Java or Python. Those are Levels 3 and 4 respectively of our version of Tanenbaum’s layered model of computer systems organisation — hence the name Platform 3½. CdM-8 Platform 3½ is consequently as relevant to an undergraduate course in programming as it is to computer architecture; however, it is designed to be easier to learn than those platforms not specifically intended for use by students.

**Data representation.** We start our journey across Platform 3½ with what appears to be a detour, but which is absolutely essential: a section on information and representation of data. Without an appreciation of the need for effective and efficient data representation, and an understanding of the principles and techniques employed in the representation and manipulation of data, it is impossible to understand the motivation behind the design of computing platforms. We concentrate here on binary representations of data, because that is what is most relevant when dealing with platforms up to Level 4. It is for others to take on the role of explaining data representation at higher levels of abstraction.

**Platform 3½ architecture.** Next we move on to a description of the virtual architecture of Platform 3½ (virtual, because it is implemented in software rather than in hardware, though it is still as real to us as if it were built using silicon and copper). We will discuss various parts of the platform, how they are connected and how they can be brought together to run a computer program.

**Controlling Platform 3½.** Platform 3½ is both configurable *and* programmable. The book is also concerned with the programmability of CdM-8 Platform 3½ without differentiating between the fixed and configurable parts of the platform. For this we will have to learn a small programming language, called the *CdM-8 macro-assembly language*.

## 1.4 Programming CdM-8 Platform 3½

A programming language is effectively a set of rules that enable us to write computer programs in a human-readable, richly commented form so that they may be understood (at least by their authors) when examined days or weeks after being written. Thanks to a platform tool, the CdM-8 macro-assembler `cocas`, CdM-8 macro-assembly language programs can automatically be transformed (compiled) into bit-strings that CdM-8 Platform 2 is able to read and execute (and which are consequently *not* human-readable).

In designing and writing programs for CdM-8 Platform 3½ you will develop skills in *computational thinking*, in expressing solutions in a programming language (also known as *coding*<sup>1</sup>). The process of detecting and correcting the mistakes you have made in writing code is commonly referred to as (*debugging*).

Programming and debugging skills are essential for Computer Scientists and Information Technologists, who need to be able to identify what platform actions are required to solve a problem, and to express the sequence of actions fluently and robustly in a language that can be understood by the platform. The lessons learned and practiced in creating programs for CdM-8 Platform 3½ are applicable to programming in a “real” programming language; nevertheless, we would adopt this approach even if this were not the case, as the best way of learning the workings of a platform is to apply it to solving practical problems.

From now on we will seldom use the phrases CdM-8 macro-assembly language, CdM-8 macro-assembly code, and CdM-8 macro-assembler because they are too long. Instead we will use the terms *assembly language*, *assembly code* and *assembler*, and occasionally the term *macro-assembler*. The reason each of these terms starts with an initial lower-case letter is because the language to which we are referring is *an* assembly language; as with any other assembly language it is specific to its platform.

<sup>1</sup> The term *coding* is used for any activity that involves expressing declarations and algorithms in a language that can be read by a platform. The term *code* is a collective noun used to refer to any set of instructions expressed this way. So a CdM-8 macro-assembly language program is a piece of *CdM-8 macro-assembly code*, and a bit string that can be executed by a CdM-8 Platform 2 machine is said to be a piece of *CdM-8 machine code*

Any other platform even of the same Tanenbaum Level will have a different assembly language. You should also be aware that assembly language programmers typically use the word *assembler* to refer both to the assembler program (a tool for translating assembly language programs into machine code) and to the assembly language. This may be lazy, but it is something you will have to get used to.

## 1.5 Circuits and Universal Platform 1

Computer Science education at undergraduate level tends to veer towards serving the immediate needs of the software industry, which, for all its pragmatic advantage, suffers from placing too much of the technology under the wraps, thus negatively impacting students' understanding. One of the most obvious shortcomings of the "pragmatic" approach is the student's lack of awareness of low-level platforms in general, and especially the very bottom of the hierarchy, the hardware levels. In this day and age when the boundary between software and hardware is blurred (the popularity of FPGA is a testament to that), it is hardly justified to exclude the basic hardware competencies from an introductory course of computing platforms, even if our target audience is not primarily students of Electronic Engineering.

Accordingly, later chapters of the book deal with what we term the Universal Platform 1, the language of circuit diagrams that include gates, memory arrays and what can be build with those, which is pretty much the whole Computer Architecture(CA). Ours being an introductory volume on Computing Platforms, we must not go too far into CA. For example, we do not give even a passing mention to issues such as cache memory and cache hierarchies, not even in the context of memory organisation. Equally, we have only a small discussion of I/O and its associated concurrency requirements. The issue of OS cannot be completely avoided, but again, here we pursue only very general understanding of the concepts despite the fact that, in keeping with the character of the book, everything is illustrated by some minimal specifics, and the utility of each of the solutions is dwelt upon long enough to ensure that the "big picture" is de-mystified.

Wherever possible we touch upon one more thing an introduction to computing platforms is uniquely useful for: the idea of multiple valid solutions that differ by cost. A student of software engineering is rarely aware of cost, concentrating primarily on code production, specifications and validation/testing procedures. Platforms is the only place where cost can be *introduced* as early as the very first year of an undergraduate programme.

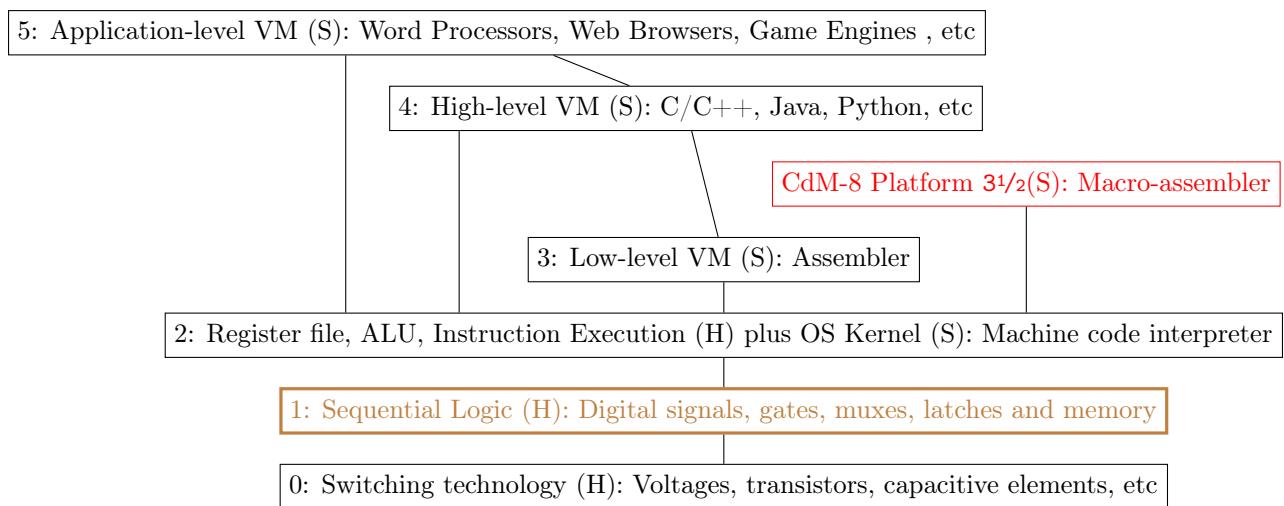
For any convincing discussion of cost, there has to be an equally convincing cost metric. In assembly programming we benefit from continuous exposure to address space and its limitations, and also from seeing a program executing in fixed-duration steps (clock cycles), rather than statements of almost unlimited variation of execution time. In studying the Universal Platform 1 and its applications for Computer Architecture, the cost is less obvious. In order to support cost intuition we opted to introduce a specific silicon solution, CMOS, on top of which our example of the Universal Platform is constructed. As a result transistor counts can be used to argue for or against specific solutions expressed in gates. Another nice feature of our approach is that the wide variety of gates (counting the variation in the number of inputs, optional negation, and optional tristate behaviour) can be accepted without "magic" and reasoned about in the context of various meaningful functional units.

# About the Subject

## 2.1 Tanenbaum's classification of platforms

Tanenbaum remarked in his books on the organisation of computer systems that the resources provided by a platform can be used to solve a specific set of problems, and to support other (higher level) platforms.<sup>1</sup> This creates a hierarchy of platforms, with platforms at higher levels in the hierarchy building upon the facilities provided by those in lower levels. The world of computing is profoundly hierarchical, both in terms of the analysis of problems and the design of machines to solve those problems. You would do well to appreciate hierarchies wherever you find them. They are just as useful as a cognitive tool, as they allow you to hide complexity under the shroud of abstraction. As we move up the hierarchy the platforms we deal with become less and less dependent upon a particular low-level implementation, and more and more understandable by human beings.

Tanenbaum proposed a hierarchical classification system for platforms with numbered Levels, where a higher number indicates that a platform offers a higher level of abstraction, and therefore will need to make use of facilities provided by platforms lower down. At the bottom (Level 0) we find hardware devices, each of which performs a task so tiny that in order to achieve anything significant one requires thousands or even millions of them, but without which nothing would work. At the top (layers 5 and above) we find virtual machines performing specific application tasks. It is convenient to view Tanenbaum's hierarchy diagrammatically and to place any specific platform of interest in the same diagram:



- n: = Level  $n$
- VM = Virtual Machine
- (H) = implemented in hardware, (S) = implemented using software

<sup>1</sup> Andrew Tanenbaum is a Dutch professor, who proposed a hierarchical view of computing platforms in a well-known textbook entitled “Structured Computer Organisation”, which has had six editions since 1975. The version we present here is somewhat simplified, but it is in keeping with the intentions of the original.

## 2.2 Every platform has a language

A platform is distinguished not only by the specific set of resources it provides, but also by the specific means used to describe the tasks that we wish it to perform. In other words, each platform has its own *language*.

The notion of language has to be approached with caution. In software, a (formal) language is defined as a set of rules for combining symbols into valid sequences. Taken together, the set of symbols (its *alphabet*) and rules (its *syntax*) that define a language are referred to as its *grammar*. A sequence of symbols that does not obey the rules of the language *is not a sentence* in that language, and may be said to be *invalid*, or *syntactically incorrect*.

### 2.2.1 A programmable platform has a programming language

We are guaranteed that the platform will “understand” all valid sentences written in its language. If the platform is *programmable* (like CdM-8 Platform 3½) its language is a *programming language*,<sup>2</sup> and valid sentences are called *programs*. A sequence of symbols that does not form a valid sentence in a programming language *is not a program* in that language, so it cannot be “understood” by the platform. However, this is only part of the story, because the grammar rules do not tell us *what the platform will do* when a program is executed (what the program *means*).

There is another set of rules and descriptions that are there to assign meaning to programs. The meaning of a program is known as its *semantics*. A program can be syntactically correct but semantically incorrect: it cannot be executed, or when it is executed it does not do what we intended. The latter is what we mean when we say that a program contains *bugs* (it “works”, in the sense that some processing takes place, but it doesn’t do what we meant it to).

**Aside:** Any running program that provides resources that can be used for solving problems is itself a platform. However, it is not necessarily a platform that can be programmed. For example, Microsoft PowerPoint is a platform for creating, modifying and storing documents, and for giving presentations. Google Chrome is a platform for downloading and rendering web pages. Platforms such as these are so far removed from the hardware that we would say they are at Level 5 in the Tanenbaum classification — or possibly higher, if they implement their own virtual machines that support programs dealing with user experience.

### 2.2.2 A “language” of circuits?

In hardware, we use circuits to create the machines we need to solve problems. Whilst it is pushing the analogy somewhat, it may be helpful to think of each component as a symbol in the “language of circuits”, and the restrictions on how they may be connected as the “syntax rules”; together these play the role of a grammar. The design of a piece of hardware is typically a diagram showing the various components and how they are connected to one another, and the actual machine is obtained by creating a material circuit (using silicon and wires) that matches the design. A circuit that obeys the rules (and therefore is “syntactically correct”) will “work”, in the sense that it will perform some action or actions, but the rules for connecting components don’t tell us what the circuit will do.

The behaviour of each component under various signals and the rules that govern their collective behaviour can be treated as the “semantics” of the circuit. Just as with a piece of software, it is possible to design and build a circuit that is *syntactically correct* (i.e. it performs a set of actions) but *semantically incorrect* (i.e. it is the *wrong* set of actions). A circuit of this kind is said to contain *bugs*, in much the same way that a semantically incorrect program is.

---

<sup>2</sup> A platform may have a control language that is not a programming language. Languages must have certain features (syntactically and semantically) in order to be considered programming languages. It is beyond the scope of this book to discuss what these are.

## 2.3 All platforms ‘compute’

Despite many differences there is one thing in common between all descriptions involved in the definition of a platform: they all define a computing world of a certain *level of abstraction*. Higher levels include artefacts of the information processing milieu: files, databases, multimedia objects, etc. Lower levels include inhabitants of the world of signals: buses, latches, gates, registers, etc.

The actions of a platform correspond to its level: whilst a Level 2 platform may be able to issue signals to other devices, change the state of registers, and write to a prescribed location in memory, a Level 5 platform may provide an instruction that renders a web page or runs a query on a database. High-level instructions of this kind are translated into multiple lower-level instructions, and these translations may go through several layers of abstraction before resulting in the lowest level platform being given a sequence of millions of very simple instructions to execute.

### 2.3.1 Hardware, software and virtual machines

Returning to the Tanenbaum diagram, the lines between boxes represent the support given to platforms further up the hierarchy by platforms in lower levels. The support mechanisms allow us to create higher-level platforms that provide single-step operations which would only be achievable at a lower level by issuing a large number of instructions and making use of many different resources. Higher-level platforms are said to be more *abstract* than the lower-level platforms upon which they rest, because users do not need to concern themselves with the details of how every little piece of work is done by the machine.

When a platform is created by running a program we say that it is a *Virtual Machine*, and we refer to the process of creating such a platform as *virtualisation*. What we mean by this is that a higher-level platform gives the *appearance* of a physical machine that provides facilities and resources that are much more sophisticated (but typically simpler to use) than those provided by the lower-level platform upon which it rests. We call this a virtual machine (VM), because it is implemented in software rather than in hardware.

It is theoretically possible to build any platform at any level directly out of hardware components, with software playing no part at all in its implementation. For example, we could build a word-processing machine out of sequential logic. This would be a ‘real’ word-processor. However, such a machine would be incredibly complex internally, making it expensive to build. It could only be modified by replacing hardware components, making it expensive to upgrade. And it would be usable for one thing only, so we would need another machine, for example, to send and receive emails.

Far better to build a general-purpose hardware platform and to write a program that makes it behave as if it were a word-processor. We get a ‘virtual’ word-processor that does everything the ‘real’ one could at a fraction of the cost. As long as the hardware platform provides the required set of resources, and the language that is used to control it supports the instructions we have used, we can upgrade the hardware without affecting the operation of the VM. We can also upgrade the program that implements the VM without having to change the hardware. Better still, we can write many different programs for *the same* hardware platform, each of which implements a different higher-level platform as a VM, so we can have a ‘virtual’ email client, a ‘virtual’ presentation package, a ‘virtual’ web browser, etc.

## 2.4 Translating between Tanenbaum Levels

There are three major mechanisms for translating the complex operations provided by a higher-level platform into the simpler operations provided by platforms lower down the hierarchy:

### 2.4.1 Aggregation

Aggregation is the only mechanism available for combining hardware components. Aggregation is a process of connecting components together to form a composite that can be used as a component in its own right. This is common practice in all forms of engineering and construction. For example a house is composed of walls, doors, windows, roof, etc; each of these is itself a composite of simpler — less capable — components, such as bricks and mortar. In fact we have become so used to this idea that we frequently think of aggregates as if they were indivisible. For example, we are happy to view an AA battery as a component of a torch, and never think about the technology or the components that are used to build the battery. However, wherever we *want* a battery we must have an aggregate of components that forms one.<sup>3</sup>

In computer hardware transistors are aggregated into silicon chips, the chips can then be aggregated into circuit boards, and circuit boards can be aggregated to form a platform that can run software. When we need a chip that performs a certain set of functions we choose one off the shelf and do not worry about how it was constructed.

Aggregation is also available as a mechanism for combining software components. Software aggregation, i.e. creating composite instructions out of atomic machine instructions and possibly other composite instructions, is usually referred to as defining a macro. A macro is a set of instructions that can act as an instruction in its own right. Each time the macro instruction appears in a program it is replaced by the instructions it has been defined to contain, which are inserted, possibly with some useful alterations, into the program at that point. The process of replacing a macro by the set of instructions it represents is known as *macro-expansion*.

It is important to understand that aggregation is *extensive*: it is based on adding more components every time a composite is required. If a wall is a composite of 1000 bricks we need 1000 bricks to build it, and we need a further 1000 bricks to build another. It's no good building just one wall and expecting the house to use it four times!

Similarly, if we want 16GiB of RAM in a computer we will need twice as many memory chips, containing twice as many transistors, consuming twice as much power, and costing twice as much money, as if we had 8GiB of RAM. And if a macro is mentioned 5 times the resulting program contains 5 copies of its defined content. This is different from the practice of defining a subroutine, which is a single sequence of instructions located at a single point in memory, which can be *executed* many times by using its name as a reference.

### 2.4.2 Interpretation

Interpretation is a much used translation mechanism at the interface between hardware and software and also in software systems themselves. Interpretation is a process whereby each single instruction given to the higher-level platform is translated into a set of instructions that are executable by the lower-level platform and these are then executed before moving on to the next higher-level operation. So each step in a higher-level program is analysed and immediately put into effect using the lower-level platform's resources.

The (interpreted) higher-level platform is not “real” in the sense that a lower-level hardware platform is real: it is a *virtual machine*. In other words it is a *description* of a machine: a container that uses lower-level instructions and data to implement a program that emulates the higher-level machine and gives the appearance that it is real. Real physical events only ever happen on the lower-level platform (unless it is also interpreted, in which case actions are delegated further down until the hardware finally puts them into effect).

Thus a silicon *processor*, i.e. an electronic device that executes machine instructions, *interprets* a program supplied in machine code, and a web browser interprets the HTML it receives from a web server line by line,

---

<sup>3</sup> Or its equivalent. As long as an aggregate of components meets the same specifications and performs the same functions as an AA battery it can be used wherever an AA battery is required.

and constructs a web page as it goes. In a way, interpretation is also extensive: each time the interpreter encounters the same machine-code instruction, or the same line of HTML, it will require the same amount of resources to execute that instruction (unless some clever engineering tricks are employed to make savings, but they do not always exist and are not guaranteed to be effective). However, a crucial difference between this and macro-expansion is that a high-level instruction may be a *reference* to a single stored copy of a sequence of lower-level instructions, so those lower-level instructions need not appear in memory more than once.

### 2.4.3 Compilation

The most powerful translation technique is compilation, understood as the process of analysing the higher platform's program in its entirety *before* attempting to execute any part of it, and translating all of it into lower-level instructions before any part of it is executed. A compiler (i.e. a compiling agent employed for this kind of virtualisation) *translates* the higher-level instructions down to the language that the lower platform supports directly.

For example, a C/C++ program is compiled to machine code or a Java program is compiled to Java byte code, and the resulting lower-level program is stored for later use. The translated version may then be executed by a hardware, software or mixed platform, as many times as we wish without referring back to the higher-level original.

The power of compilation comes from the fact that it is done with knowledge of the whole application (or a large part thereof, which has a degree of autonomy). The process may involve many types of sophisticated analysis and optimisation and result in a finely tuned and trimmed low-level program that takes both functional and physical characteristics of the specific lower platform into account.

## 2.5 The Tanenbaum Levels

We return again to the platform classification to look at the different levels in a little more detail.

### 2.5.1 Level 0

Starting with the **bottom level platform**, we may see it as having the ability to manipulate discrete electrical signals, acting like a multitude of miniature light switches, except that they can be switched about 12 orders of magnitude ( $10^{12}$  times) faster than the electric light in your living room (individual switching events may take a few picoseconds). That is the way that digital (on/off) signals are processed. Switches control, and are controlled by, other switches in a switching network, which is what is typically found on a modern silicon chip. One cannot easily understand what processing is going on, or what data values are being processed, just by looking at these switching networks, nor does one need to.

The purpose of the Level 0 platform is to support meaningful hardware units of digital processing, which form the **Level 1 platform**.

### 2.5.2 Level 1

Level 1 components are constructed by *aggregating* Level 0 components. Switches are combined into units that have some definite functional behaviour: they can perform arithmetic and logic functions familiar from junior school mathematics and elementary logic. The switches may also be combined to form units of memory or data communication infrastructure. Those useful things will be presented in Chapters 9 and 10, when we focus on digital logic. Interestingly, **there is very little diversity at Level 1**.

The functional behaviour of Level 1 components tends to be common to all Level 2 platforms, nearly all of which are realised by aggregating Level 1 components. Interestingly, Level 0 is considerably more varied: gates, memory and interconnect can be constructed in many ways from various types of physical switch.

### 2.5.3 Level 2

The next step up, from Level 1 to **Level 2**, is made by aggregation. There are many different Level 2 platforms, from microcontroller-based systems found in products like washing machines and microwave ovens all the way up to the kind of GPU found in games consoles that has a thousand times more switches in each of its aggregate units (and thousands of quite complex units on a single chip, too). The art of producing Level 2 platforms of considerable power and utility is the domain of digital circuit engineering, something with which we can afford only the most fleeting of encounters in this introductory course.

What is important for us here is that once synthesised (i.e. aggregated from the universal Level 1 units) a Level 2 platform serves as an *interpreter* of an *instruction set*, i.e. a certain set of binary strings. Each of these binary strings represents a different *machine instruction*, and each such instruction causes the machine either to perform a specific arithmetic or logic operation upon one or more operands held in it, or to read or update the content of external memory. **A Level 2 platform is always such an interpreter.**

### 2.5.4 CdM-8 Platform 2

In this book we use as a running example of the Level 2 platform the circuit known as CdM-8 Platform 2, a processor<sup>4</sup> synthesised by hand from Level 1 units by means of aggregation (for which full circuit schematics is provided on the book web site). As with any Level 2 platform, CdM-8 has an associated instruction set (a set of machine code instructions, each of which is encoded as a sequence of bits). Although we lack a physical circuit capable of interpreting that set, we have access to a **different Level 1 platform, called LogiSim**, which is in fact an interpreter that takes a circuit *description* in Level 1 terms as a “program”, interprets that circuit in a virtual world, and shows its signal behaviour via a screen.

---

<sup>4</sup>this term will be properly defined later, but anyone who has ever bought a computer or even a mobile phone already knows it is something on which the running of programs depends for effectiveness and speed; they are not wrong.

LogiSim is itself a program, a piece of software for an ordinary computer, so it runs on its own Level 4 platform (which happens to be Java) on top of your choice of Level 2 platform (which happens to be Mac OS X for the authors of this book). It is quite remarkable that in running CdM-8 programs via a circuit simulator at least three software levels of interpretation are engaged: Java interpreting LogiSim in bytecode, Logisim interpreting the CdM-8 circuit in a graphical language, and the CdM-8 circuit interpreting the CdM-8 instruction set as a simulated circuit, so that eventually a piece of CdM-8 software may run on top of all as if on a real physical CdM-8 computer.

But the magic of multilevel computer organisation does not stop here. We have yet another implementation of Platform 2, this time one that avoids the triple interpretation. A *system level*, i.e. **Level 2** CdM-8 emulator, is available as part of the CdM-8 integrated development environment ([cocoide](#)), which interprets CdM-8 instructions directly, not via interpreting the Level 1 platform and the circuit first. As a result, it is orders of magnitude faster and more convenient to use, but... you don't get to see any signals, nor the workings of the low-level machinery: the ALU, the Sequencer, the data buses, etc. Still, quite a lot of important details are available as opportunities to debug code and examine data in various useful formats. And of course we will descend into Level-1 supported Level 2 in Chapters 12 and 13, so no opportunity will be lost.

### 2.5.5 Level 3

Then we come up to **Level 3**, which is, as mentioned earlier, the assembler level. A pure Level 3 platform provides all instructions that are made available by the underlying Level 2 platform, so the assembly language programmer can achieve anything that can be achieved using the Level 2 platform. A Level 3 platform usually provides extra instructions to make a programmer's life more bearable, but the difference between Levels 2 and 3 is largely one of appearance and programming convenience. Level 2 is defined exclusively in terms of bit-strings. Everything is a bit-string: any instruction, any item of data, the representation of any condition or event. This makes Level 2 programming almost impossible (and at any rate impractical).

Level 3 programming is performed in a human-readable (and writable) language, with each operation, condition or event identified by a short (typically 3 or 4 character) name, often in the form of an acronym or other abbreviation. These short names are called assembly language *mnenomics*. A machine code operation may require one or more operands in order to form a complete instruction, and most assembly language mnemonics need to be accompanied by operands in the form of data values or pointers to the locations where data values can be found. An assembly language will typically provide programmers with the means to refer to built-in resources (such as registers) that can act as operands, to associate names with memory addresses that can be used as operands, and to refer directly to specific numbers, characters, and other kinds of data. This saves the programmer from having to specify the operands as bit-strings, and from having to perform a great many tedious conversions by hand.

### 2.5.6 Level 3<sup>1/2</sup>

An assembler represents the minimum amount of tooling that makes a Level 2 platform human-usuable. A macro-assembler adds features that further raise the level of abstraction above Level 3. These make the programmer's job a lot less unpleasant. In particular, a macro-assembler introduces macros, which pushes the platform up to **Level 3<sup>1/2</sup>**, bringing us back to where we started our journey: CdM-8 Platform 3<sup>1/2</sup>. The CdM-8 Platform 3<sup>1/2</sup> macro-assembler first expands the macros, then compiles the resulting assembly code into CdM-8 Platform 2 machine code.

So the replacement of an assembler's Level 3 language, or a macro-assembler's Level 3<sup>1/2</sup> language by the machine code of a Level 2 platform requires *compilation*. We are going to be faced with compilation for the first time when we use the CdM-8 Platform 3<sup>1/2</sup> macro-assembler. Since many modern programming languages are also compiled, studying the macro-assembly language, and using it to solve programming problems, will help us appreciate what is involved in writing a compiled program. It will give us a glimpse into issues of syntax and semantics, compilation errors, etc.

Since programs written in an assembly language can only solve problems by directly manipulating the resources of a Level 2 platform, we will have only the most basic tools at our disposal, but at least we will not have to write these programs as machine code bit-strings. We will, however, have to understand how our chosen Level 2 platform is organised in order to determine how to solve problems, and we will need to

'look inside' the Level 2 platform to see whether our programs are doing the right things, and to work out how to fix them if they are not. Later on we will discover more joys of compilation, such as the ability to compile parts of a program (sections, libraries, etc) and re-use them with newly written parts.

# Representation of Data

## 3.1 Discontinuity

The world of computing is based on discontinuity. The idea of discontinuity is familiar to us. We distinguish day from night, presence from absence, up from down, truth from falsehood. However, the material world is continuous and in most cases there are intermediate values or phases between otherwise separate items or realities. Out there, there are dawn and dusk that are neither day nor night, there are degrees of presence based on distance, there are directions in between ‘straight up’ and ‘straight down’, and there are various interpretation of a truth.

### 3.1.1 Computers utilise discrete data items

Whilst all of this is of interest to the human users of computers, none of it is of any concern to a computing engine. The world of computing is based on a discrete (not to be confused with *discreet*) model of reality rather than the reality itself in all its uncountable, continuous, fuzzy complexity, so any model that we will build for use with a digital computer can only be an *approximation* of the real world.

The simplest discrete object is a single binary data item. It can take on one of only two values. There are many ways to represent the value of such an item, the most common of which is to use a single symbol (digit) taken from a two-symbol set: {0, 1}.

The symbol 1 may model the presence of something, or the truth of a statement, or a non-vanishing quantity, while 0 may be taken to mean the absence of the same thing, the falsehood of the statement or a vanishing quantity. Or, indeed, the other way around: the engine does not care what humans may use its abstractions for.

There could be thousands of other interpretations of those symbols. The only thing that is important for the engine itself is that it can tell 0 from 1 and 1 from 0, and that it preserves this fundamental discontinuity between them.

### 3.1.2 Discrete models

The content of a computing universe is a projection of the real world onto a discrete model. Several kinds of artefact are available. There are **data items**, **programs**, and **events**. To be useful and usable, a computing engine has to be *implemented*, i.e. realised as a physical device where the model is supported by the device’s signals. Some of these signals may be used to interact with a human being across a *human-computer interface*, making it possible for the computing engine to be used.<sup>1</sup>

---

<sup>1</sup> An *interface* is a point where two systems meet, across which data and instructions may be sent, and where one side may alert another to the occurrence of an event. A *human-computer interface* is a means of sending data, instructions and events between a human being and a computer.

### **Definition 2 : Data items, programs and events**

- **Data Items** reflect properties of objects in the material world, such as measurements, counts, and textual descriptions.
- **Programs** are sequences of discrete instructions that tell a computing engine what to do with data items
- **Events** are discrete real-world occurrences that cause programs and circuits to respond by making other events happen

#### **3.1.3 Discrete values are easy to implement**

Each known physical implementation relies on only a small set of ideas. The discrete nature of the computing universe is reflected in the use of discrete electrical signals: 1 can be represented as a high voltage and 0 as a low voltage or vice versa. Any implementation is based on a technology that excludes the middle: makes sure that the voltage can be categorised as either high or low with certainty at any time that it is examined. To be useful for capturing discrete reflections of the material world, the implementation must be capable of manipulating billions of such signals, which brings us to the following amazing fact. The three kinds of content in the computing universe - the data, the programs and the events - may all be represented using the same implementable concept: a sequence of binary values, also known as a *binary string*.

A binary string (which is often called a bit-string, and we will soon see why) is a fixed-length sequence of binary digits (or bits). For example, each of the sequences 000, 110, 001, and 111 is a binary string of length 3. It is truly astonishing how wide a variety of objects (in fact *any* discrete object with a small enough size, whatever its nature) can be represented using binary strings, whether the objects are atomic or compounds containing a complex, hierarchical or interdependent arrangement of atomic objects. Bit-strings are as fundamental to computing platforms as DNA sequences are to living organisms.

Consequently, before we can see the workings of any discrete computing engine, we must familiarise ourselves with how bit-strings may be used to represent the world in at least a few major areas. Before we start, we should get some idea of how capable bit-strings are of holding information.

## 3.2 Information

In everyday life we use the word information very loosely. We speak of information as something that is synonymous with awareness and knowledge. We could say things such as “according to the information I have it’s going to rain tomorrow”, or “we have no information about how much the audience enjoyed the performance”. In the world of computing, the term *information* can have two possible interpretations:

1. The first interpretation is derived from a branch of mathematics known as *information theory* (not to be confused with information science, which is something quite different). In information theory, to have information means to be able to distinguish representatives of a set from one another with certainty<sup>2</sup>. This ability can be quantified, by counting how many different representatives one can tell apart.
2. The second is broadly similar to that used in everyday life: to have information means to have a context within which to interpret items of data in order to achieve understanding of some aspect of the world. So the number 7 (which is an item of data) conveys no information to human beings about the real world unless they know that it is being used to refer to the number of dwarves known to Snow White, or to the number of stars in the constellation Pleiades, or to some other count or measurement.

It is only the first, narrow interpretation that we are interested in here.

### 3.2.1 Quantifying information

If a class of objects contains  $n$  possible representatives, and one has certainty about which representative a particular object is, then it is said that one has  $\log_2 n$  bits of information about the object. Here  $\log_2$  is a base-2 logarithm, that is, the power to which 2 must be raised in order to equal  $n$ . So if we can identify which one of 8 different categories an object falls into we have 3 bits of information about that object ( $2^3 = 8$ ). If two objects have *identical* representations in our computing universe we cannot tell them apart, so we have 0 bits of information about those two objects. The word “bit” here is the unit of information, which represents the amount of information that creates certainty about whether some binary digit  $x$  is 0 or 1.

#### Definition 3 : Measures of Information Content

- **The Bit.** The basic unit of information is the *bit*, which is the amount of information that creates certainty about whether some binary digit  $x$  is 0 or 1.
- **Quantity of Information.** If we have a set of  $n$  objects, each of which we can immediately identify as different from all the others, we are said to have  $\log_2 n$  bits of information about any object that belongs to the set. Conversely, if we have  $k$  bits of information about an object, it means that the object belongs to a set of  $2^k$  distinct members, and we know which one it is.

### 3.2.2 Binary strings

A sequence of binary values<sup>3</sup> of fixed length is called a *binary string*.

Now imagine that under some interpretation a size- $k$  binary string is associated with a class of objects. For example, each of the four major compass directions (*North*, *East*, *South* and *West*) may be associated with a different bit-string of length 2, so that  $(00 \mapsto \text{North})$ ,  $(01 \mapsto \text{East})$ ,  $(10 \mapsto \text{South})$  and  $(11 \mapsto \text{West})$ . Then the certainty about a compass direction carries  $\log_2(4) = 2$  bits of information. It is not a coincidence that a size-2 binary string carries exactly 2 bits of information. To understand why let us focus on the following observation.

<sup>2</sup> In fact, the full theory of information developed by Claude Shannon, a famous 20th century American mathematician, can deal with probabilistic information where certainty is not required, but that, at present, is more relevant to communication than it is to computing

<sup>3</sup> It doesn’t matter how the two different values are represented, as long as they can be told apart. Conventionally we represent them as 0 and 1.

If we consider a size-2 binary string and add an extra binary digit to it, the resulting size-3 string will have 8 possible combinations: 000, 001, 010, 011, 100, 101, 110 and 111. The first group of four binary strings in our list differ from the second group of four only by the value of their first digit: for the first four the value is 0, and for the others it is 1. So adding an extra digit to a binary string appears to double the number of different strings we can create. If the base-2 logarithm of some number,  $n$  is  $l$ , the base-2 logarithm of twice that number,  $2n$ , is  $l + 1$ . Mathematically,  $\log_2(2n) = \log_2(n) + 1$ . Given that we added exactly one digit, we conclude that each digit carries exactly one bit of information! That explains why binary strings are often referred to as bit-strings, and binary digits as bits.

#### Definition 4 : Properties of Bit-Strings

- **Bit String.** In computing we frequently refer to binary strings as *bit-strings*. A bit-string of length  $k$  is referred to as a  $k$ -bit string.
- **Sets of Bit Strings.** The universal set<sup>a</sup>. The universe of  $k$ -bit strings has  $2^k$  distinct members.

---

<sup>a</sup> The *universal set* - also known as the *universe* - is the set of items of some type, containing all possible items of that type, see Section 3.10

The number  $2^k$  quickly gets very large:  $2^{10} = 1024$ ,  $2^{20} = 1024 \times 1024$  and  $2^{1000}$  is bigger than the number of elementary particles in the universe. If we wished (and if quantum mechanics did not forbid us) to tag every elementary particle in the universe with its own unique binary string, about 250 bits per tag would be quite enough. That is how immensely capable a bit string is of holding information. Of course the bit-strings quickly get longer when we consider *combinations* of objects, and each extra bit we employ *doubles* the number of representations.

#### Example 1 : A student record

Nine bits (512 possible bit-strings) might be sufficient to tag every student in a class with certainty, but if we wished to be certain about each student and their marks in every first-year subject of an undergraduate programme, we would need a 9-bit string to identify the student, plus a further 7-bit string (128 combinations) for each subject to cover the range from 0 to 100 marks in each *at the same time*. So if every student is taking four subjects we will need a bit-string of length 37 for each of them, because of the number of possible combinations of student and marks. A 37-bit string can represent  $2^{37}$  different possibilities (over 137 billion).

#### Example 2 : A photographic image

More dramatically than the student record example, a simple image that has 2 megapixels (produced by a front-facing camera on a mobile phone) might require about 1,000,000 gradations of colour in each pixel, or about 20 bits of information. That's 2 million bit-strings of length 20, so to represent the colour values of all pixels in the image one needs a string 40 million bits in length. And there are  $2^{40,000,000}$  such strings, each of which represents a different image.

Of course a modern computing platform has the ability to hold binary strings of such length (and in fact strings that are a thousand times longer) in memory so that they may be digitally manipulated. However, such manipulation almost never happens on the level of a whole large string, especially when the bit-string can be cut up into portions that hold data for identifiable parts.

### 3.2.3 Manipulating data representations

In the example of a student record, the statement that we have combination number 10,002,175,257 has all the information and represents absolute certainty about both which student we are referring to and that student's marks in each of the four subjects. Yet the information is in a form that makes it hard to process, e.g. to answer a question such as: has this student passed the year? Whereas if the data were available in the form (Student 37, Subject 1 mark 33, Subject 2 mark 52, Subject 3 mark 90, Subject 4 mark 25), the answer could be obtained very quickly.

Consequently for content of any kind (data, program or event) the representation must provide at least two capabilities: *navigation* and *identification*.

#### Definition 5 : Navigation and Identification

- **Navigation.** Any data representation must make it possible to *locate* a specific data item in a combination of items.
- **Identification.** It must also be possible to *identify which* of the possible values of the data item has been located by navigation.

In the student record example, *finding the sequence of bits* within a student's record that represents the student's number is a matter of **navigation**, and *identifying which student* has that number is a matter of **identification**. Another pair of terms that mean the same and have broad currency are *addressing* (finding) and *encoding* (identifying).

This minimum set of capabilities is rarely sufficient. A representation must also support *effective* and *efficient* operations on data. For example, if the set of  $k$ -bit strings represents a set of integer numbers, then in principle any size- $2^k$  set of numbers can be represented with absolute certainty. However, if we wished to use this representation to add numbers, we would have to be assured that for any two numbers represented as bit-strings of length  $k$ , the sum is also representable as a bit-string of length  $k$ . If this is the case we say that addition is *effective*. If addition is effective but requires much work to find which bit-string represents the sum, the representation we have chosen is clearly *not efficient*. Of course our measure of efficiency depends on the meaning of "requires much work" – Compared to what? Measured in what units? – But the principles of effectiveness and efficiency remain key to the representation of data.

#### Definition 6 : Operations.

An **operation** is an action that may be performed by a computing platform upon one or more data items, giving another data item as its result.

- An operation on items belonging to particular universes is said to be *effective* if it can be guaranteed to give a correct result when performed upon *any* items belonging to those universes, no matter what values they represent.
- An operation is said to be *efficient* if a computing platform makes efficient use of available resources when performing it. How this is measured depends upon the operation, the platform, and the broader context.

In the subsequent sections we will see how various objects can be (and are typically) identified (*encoded*). For collections that represent combinations of encoded objects we will explore how they are navigated (*addressed*) in order to find and identify individual components. For all kinds of representations we will also discuss what operations are available and how they are made effective and efficient.

## 3.3 Operations on numbers

We start with the observation that the representation of numbers in a  $k$ -bit string can never be fully effective. Indeed no finite set of numbers is, as mathematicians say, *closed under* addition. In simple language: if you repeatedly select two numbers at random from a finite set and add them together, sooner or later you will

obtain a result that does not belong to the set. This is disappointing. If we have a computing platform that can only deal with bit-strings of finite length (which is every platform there is), it can only handle finite sets of numbers. So there will be some numbers that we cannot represent, and some pairs of numbers that we *can* represent but *cannot add together* to obtain a representable result. Here is a simple demonstration.

### **Example 3 : Addition of 8-bit binary numbers**

Suppose we start with the set of whole numbers between 0 and 255 (each of which can be represented uniquely by a different bit-string of length 8). If we select the number 200 and the number 100, the result we get is 300, which is larger than the largest number in the set, and so definitely does not belong to it.

No matter how we use the set of 8-bit strings to represent a set of numbers, there would only be  $2^8$  different bit-strings in the set (that's 256). The very meaning of representation dictates that each different number must be encoded by a different bit-string (the principle of identification). Because there are only 256 different sequences of 8 bits we can only encode 256 different numbers. By the argument given above, the operation  $+$  on a set of 256 numbers is ineffective.

It doesn't take a genius to see that the same argument holds true for any set of numerical values represented by bit-strings of length  $k$ , no matter how big we make  $k$ .

#### **3.3.1 Addition of $k$ -bit binary numbers is ineffective, and potentially unsafe**

For any finite set of numbers<sup>4</sup>, it will always be possible to find at least one pair of numbers that, when we add them together, gives a result that is not in the set. Only an infinite set of numbers can be closed under addition, and to represent an infinite set we need an infinite length bit-string. So it is not possible to make addition effective unless we have a machine with infinite resources.

Worse still, if we add two  $k$ -bit numbers together and their sum is a number that cannot be represented, what should the computing engine produce as the result? Assuming that each of the  $2^k$  different bit-strings has been allocated to a numeric quantity, the only choice is to produce an incorrect number<sup>5</sup>. If this is reported without being accompanied by a warning the program will act on the (incorrect) result and do the wrong thing. This state of affairs is said to be *unsafe*, because the program is no longer behaving as it should but *we do not know*. At best the program will give incorrect outputs. At worst it could damage property or even kill somebody, if it is used in a safety-critical system.

In practice we are happy to accept a solution that works for *most* pairs of numbers, as long as we can tell when it isn't going to work (or, at least, detect when it has failed to work). So, if the set of numbers being represented is really large, and if only a small subset is typically used for any calculation,

- we can assume that a program would very seldom need to use any of those pairs of numbers for which the operation  $+$  gives a result that cannot be represented (the same with any other operations that are not effective on finite sets of numbers).
- we need the platform to detect when a program uses an operation that gives an unrepresentable result, and we need it to have a means of proactively informing the program that this has happened.

In other words, we must impose a *restriction* on our data and demand a *reaction* of the platform to any abnormal situation with operations. Under these restrictions the arithmetic on bit-string representation becomes effective, and if the platform provides the needed reaction, it is also safe. However, the issue of efficiency still remains.

---

<sup>4</sup> Except the empty set, and the set containing just the number zero

<sup>5</sup> The engine cannot “refuse” to compute when the computation is mandated by a program, all it can do is generate sufficient data for the program to be adequately informed

### 3.3.2 Using a table to implement an operation

One possible solution would be to choose a bit-string representation for each number in our finite set, and then make up a table that shows the bit-string representations of the results obtained upon performing an operation. We will take the addition operation as our example.

There are many possible ways to represent the numbers in a finite set as bit-strings, and if we are using a table it really does not matter much which of them we choose. Here is one possible way to use 3-bit strings to encode each of the whole numbers in the range 0 to 7:

0	000
1	001
2	011
3	010
4	110
5	111
6	101
7	100

The above encoding has a name: the *Gray code*. This set of bit-strings has a special property: for any  $n$  within the set, the bit-string representing the number  $n$  differs from the bit string representing the number  $n + 1$  (if it belongs to the set) by a single bit flip<sup>6</sup>. However, the only way to know which bit needs to be flipped is to know the code. For example, changing the bit-string 110 that encodes 4 to the bit-string 111 that encodes 5 involves flipping the right-most bit of the string, and changing from 1 to 2 (or vice versa) involves flipping the middle bit.

To be able to do addition effectively (on a restricted set), we need an addition table, such as this:

+	000	001	011	010	110	111	101	100
000	000	001	011	010	110	111	101	100
001	001	011	010	110	111	101	100	U
011	011	010	110	111	101	100	U	U
010	010	110	111	101	100	U	U	U
110	110	111	101	100	U	U	U	U
111	111	101	100	U	U	U	U	U
101	101	100	U	U	U	U	U	U
100	100	U	U	U	U	U	U	U

where each cell in the table contains the sum of the row and column headings in the chosen encoding. The cells marked as U indicate that the sum is unrepresentable (it is a number outside the range 0 .. 7, so we have no encoding for it).

### 3.3.3 Pros and cons of tables

The advantage of the table method is that it works no matter what representation we choose: the encoding can be completely arbitrary provided that each number in the set has its own unique bit-string representation. Notice that we can represent the table itself as a bit-string: scan the table row by row from top left, skipping the formatting lines and the U's, and copy out the digits one after another into a single string. We just need to make sure that we write the algorithm that can *navigate* the table to find the answer.<sup>7</sup> Consequently a platform capable of supporting bit-strings (and that means any sufficiently large computing platform) would be able to support this approach.

One great disadvantage (and one of the reasons why an arbitrary representation is not necessarily good for a real platform) is that the table is potentially huge. For a representation that uses bit-strings of length  $k$ ,

<sup>6</sup> Flipping a bit means replacing it by its complement: 0 is replaced by 1, and 1 is replaced by 0. For example, if we flip all bits in the bit-string 011001 we get 100110.

<sup>7</sup> There are several ways to do this. Possibly the most straightforward is to use the fact that each row ends with the bit-string '100'. See if you can write an algorithm that will do the job.

the number of columns required to capture all possible bit-strings will be  $2^k$ , and the number of rows will be  $2^k$ . So the number of cells in the table will be  $2^k \times 2^k = 2^{2k}$ .

Almost half of the information in the table is redundant since for any pair of values  $x, y$ , the sum  $x + y$  has the same value as the sum  $y + x$ . Consequently the exact answer is closer to  $2^{2k-1}$ , even a little less since the (bottom-left to top-right) diagonal need not be included, as every entry is 100, so we end up with  $2^{2k-1} - 2^k$  table entries. In practice that makes very little difference: for a reasonable  $k$ , say 32 for a modern computer, the table would contain roughly  $2^{63}$  entries, each of which is 4 bytes<sup>8</sup> (32 bits) in length.  $4 \times 2^{63} = 2^{65}$  bytes. That's well over 30 million terabytes. Clearly, it is completely impractical to keep that much information on a platform just to be able to add together two integer numbers.

What we really need is an efficient  $k$ -bit representation and a completely mechanical *procedure* for working out the result from the operands<sup>9</sup> without using a table, i.e. a simple enough summation algorithm. Which is where *positional number systems* come in.

---

<sup>8</sup> A *byte* is a unit of size for bit strings. A byte-size bit-string has exactly 8 bits in it. In the world of platforms larger units are also common: a kilobyte is 1024 ( $2^{10}$ ) bytes or 8192 bits, a megabyte is 1024 kilobytes, etc.

<sup>9</sup> In any arithmetic operation we call the values that are being combined the *operands* and the way in which they are combined the *operator*. So, in the sum  $x + y$  the operands are  $x$  and  $y$  and the operator is  $+$

## 3.4 Positional number systems

As the world of human-conceptualised reality has grown larger with the march of civilisation, mankind has come to appreciate the need for efficient representations of numbers. Even the Babylonians had a number system that supported mechanical addition and the use of an algorithm to perform multiplication<sup>10</sup>, but those were skills that only sages had a chance to develop.

Numeracy in human society started to improve dramatically with the invention of the Hindu-Arabic positional decimal system. Like the ancient Babylonian one, it had an effective representation for addition and subtraction in an unlimited range of integers, but unlike the Babylonian system it also allowed efficient multiplication and division. The Hindu-Arabic positional decimal system forms the basis for the number system you learnt in primary school. Let's look at the principles laid into its foundation.

### 3.4.1 Base ten numbers (Decimal)

Numbers are written using strings of symbols (called *digits*<sup>11</sup>), with each different digit representing a different whole-number quantity in the range zero to nine. The term *decimal* also comes from Latin and stems from the fact that there are ten different digits.<sup>12</sup> In most countries people use the symbols 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. When a *string* of digits is formed, it represents the *weighted sum* of the contributions made by individual digits. For example, the number 8635 denotes 8 lots of a thousand, plus 6 lots of a hundred, plus 3 lots of ten plus 5 lots of 1, or

$$8 \times 1000 + 6 \times 100 + 3 \times 10 + 5$$

The weighting of a digit in a positional system depends on its *position* within the digit-string. The lowest-weighted digit is the one at the far right-hand end: it is called the *least significant digit*, because it is the one with the smallest weighting. The highest-weighted digit is the one at the far-left end of the string: it is called the *most significant digit*. We number the positions in the string starting with 0 for the right-most (least significant) and working up through 1, 2, 3, etc until we reach the left-most (most significant). The position number is the *logarithm to the base ten* of the digit's weighting.<sup>13</sup> The digit in position 0 is given the weight  $10^0$  (=1), the digit in position 1 has the weight  $10^1$  (=10), the digit in position 2 the weight  $10^2$  (=100), etc.

In our example the least significant (right-most) digit is 5 so it contributes  $5 \times 10^0$  (=5) to the total. The digit to its left acquires the weight  $10^1$  (=10), so in this case it contributes  $3 \times 10^1$  (=30) to the total. The digit in position 2 has the weight  $10^2$  (=100), so contributes  $6 \times 10^2$  (=600), and the digit in position 3 has a weight of  $10^3$  (=1000), contributing  $8 \times 10^3$  (=8000) to the total. This is the most significant digit, so the string represents the quantity eight thousand six hundred and thirty five.

### 3.4.2 Positional weighting and string length

In a positional number system we can have digit strings of any length we choose. It is easy to see that the longer we allow strings of digits to be, the larger the quantities that we can represent. Also observe that we can prefix a digit string with as many 0s as we wish without changing its value, because  $0 \times 10^n$  will always contribute 0 to the total, no matter how big  $n$  happens to be. So, whilst it may be unconventional, we know that the numbers 8635, 08635, 008635 and 0008635 all represent the same quantity. This is another useful feature of positional number systems, because it is easier to mechanise arithmetic if all digit strings the computing engine is working with are of the same length.

It is very easy to get distracted by the use of ten digits, but this is not the most important innovation of the system. What is really important is the positional weighting principle, and the invention of the idea that a symbol may be used to represent the quantity *zero*. This means that we can write large numbers in a small space, and perform mechanical calculations upon them. Positional weighting of digits is more general than the Hindu-Arabic decimal system. As long as we have  $n$  symbols, one of which is reserved for representing the quantity zero, we can devise a *base n* representation that allows us to write any whole number<sup>14</sup>.

<sup>10</sup> The Babylonians had a base-sixty number system that employed complicated digits drawn from a collection of sixty cuneiform symbols, and used an abacus and half-square tables for calculations thousands of years ago

<sup>11</sup> The word digit comes from the Latin for *finger*

<sup>12</sup> The word decimal is derived from the Latin for the number *ten*

<sup>13</sup>  $\log_{10}1 = 0$ ,  $\log_{10}10 = 1$ ,  $\log_{10}100 = 2$ ,  $\log_{10}1000 = 3$  etc. So  $10^0 = 1$ ,  $10^1 = 10$ ,  $10^2 = 100$ ,  $10^3 = 1000$ , etc

<sup>14</sup> Indeed the Babylonians with their 60-digit set representing quantities from 0 to 59 directly had the very same positional

### 3.4.3 Base two numbers (Binary)

It should come as no surprise that the positional weighting principle is the basis of an efficient number representation in computing platforms. As those platforms are based on, and are reducible to, the fundamental Platform 1 that can only deal with binary values, we only have 2 symbols at our disposal. So we will devise a *base n* positional number system where  $n = 2$ , and we must choose one of the two symbols we have available to represent the quantity zero.

Given that we typically use the symbols 0 and 1 to represent the two different binary values the obvious symbol to choose to represent zero is 0. Each whole number quantity will now be represented using a string of binary digits (a bit-string), with the least significant bit having the weighting  $2^0$  (which is 1), the next left digit having the weighting  $2^1 (= 2)$ , the next left having the weighting  $2^2 (= 4)$ , the next left  $2^3 (= 8)$ , etc. Just as with decimal numbers, we may write binary numbers of arbitrary length and leading zeros do not change the quantity represented by a binary number.

We do not have an unlimited number of bits available for representing numbers, and machine operations can be made more efficient if different numbers are represented using the same length of bit-string. So we choose the *length*,  $k$ , of the bit-string that will be used to represent numbers, giving us  $2^k$  different whole number quantities we can represent. We now know that a string of  $k$  bits needs to be read into the arithmetic device to retrieve a whole number, and  $k$  bits must be set aside to store each whole number in memory. We also know that our arithmetic device must be able to perform operations on bit-strings of length  $k$ . We say that we are using  $k$ -bit numbers and the platform is performing  $k$ -bit arithmetic.

For efficiency reasons, the designer of a platform should use the whole set of size- $k$  bit-strings to represent numbers (otherwise information would be wasted), which leads to another property of binary number systems. A  $k$ -bit number system will represent  $2^k$  different quantities. The simplest number system is based on a bit-string of length 1, which we can use to represent the two quantities zero and one.<sup>15</sup> The next simplest is a bit-string of length 2, which we use to represent the four quantities zero, one, two, and three, etc.

The rest happens automatically: since the positional principle assigns weights as progressive powers of the base, the rightmost digit will, as before, have a weighting of 1 ( $= 2^0$ ), the one to the left of it will have the weighting of 2 ( $= 2^1$ ), which is the number-base of the representation (the number of digits in the set of digits), the one to the left of that will have the weighting 4 ( $= 2^2$ ), then 8 ( $= 2^3$ ), etc. For example the string 11011 will represent the quantity

$$1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 = 16 + 8 + 2 + 1 = 27$$

### 3.4.4 Quantities, numbers and digit strings

The multifaceted nature of digits, quantities, symbol strings and numbers can be a little confusing at first, as can the notation used to write them down, but you will get used to it over time. In this text we will try to be consistent in calling the abstract idea of number a *quantity*, as in *thirty-four* oranges or *seven* pears, the record of a quantity a *number*, as in 34 or 7, the symbols used in the record *digits*, such as 3, 4, and 7, and the sequence of digits (irrespective of its meaning) a *digit-string*, e.g. 0049800. If the digits are binary digits 0 and 1, we will continue, as before, to call them bits, and the string of bits a binary string or a bit-string. Whenever a bit-string represents a quantity, i.e. it is a number, it will be called a *binary number*.

A quantity is an *idea*, whereas a number is a *representation* of that idea. When referring to a quantity (as opposed to a number that represents that quantity) we must use some language to identify it irrespective of how it is represented. For example, the abstract idea of “three” is an idea based on equivalence rather than counting. When we say that a set of objects has three members we mean that we have a reference set of objects (which could be three distinct marks on a sheet of paper) that *defines* the concept “three”. Any collection of objects that can be tagged with those marks fully (in the sense of having enough marks to assign one to each object with none remaining) can then be accepted as a collection of **three** objects. That is all that is required to have the notion of quantity; there is no need to have a fixed representation, no need to be able to count, and no need to be able to do arithmetic operations.

---

system only it was one based on the weight 60. The rightmost digit was taken at its face value, the one to the left of it with the weight 60, the next one to the left with the weight  $3600 = 60^2$ , etc. Again there was no limit to the size of integers that the string of Babylonian digits could represent as it grew longer and longer.

<sup>15</sup> Or any other pair of quantities.

Because of the equivalence nature of quantity we can, in particular, use some convenient representation to make it easy to communicate to other people the quantity to which we are referring. It is not important which representation we choose to do this, as they are all equivalent to each other in terms of the quantity they are representing, but we need to *know* which representation is being used so that we can identify which quantity is being represented. In this text we will use the decimal positional system and English words such as twenty-seven, in referring both to quantities and decimal numbers, and we will rely on the context wherever the distinction between numbers and quantities may be important.

To summarise, we have just introduced a binary positional system for representing quantities as bit-strings in the hope that the system will prove to be as efficient for arithmetic as its big sister the decimal positional system is. We will use the binary positional system at the Platform 1 and Platform 2 levels, and will write and manipulate binary numbers in this text, but we will continue to use the decimal positional system as the language for referring to quantities. When working at Levels 3 and above it is normal to use the decimal positional system for quantities, but to use the hexadecimal system as a shorthand way of representing bit-strings

### 3.4.5 Conversion between decimal and binary number systems

How do we recognise the quantity represented by a number in one positional system and how do we express the same quantity as a number in another system? We humans have been irrevocably decimalised by culture and by education. So much so that most of us have difficulty differentiating between abstract quantities and their decimal representations. The conversion question effectively begs for a recipe (an algorithm) that we can use to convert between decimal representations and representations in whatever other number system we might be interested in.

### 3.4.6 From decimal to binary

We start with an observation. When you first learned how to perform division in primary school you were taught that one natural number can be divided by another. For example, Sally has 26 sweets that she wants to share with her 5 friends, so she makes 6 equal-sized piles of 4 sweets, and has 2 left over. We call the first of these two numbers the *quotient* and the second the *remainder*. So  $26 \div 6$  gives a quotient of 4 and a remainder of 2. We rely on this idea when performing conversions between positional representations.

Let us, for example, divide 547 by 10. The result is obviously 54 (the *quotient*) with 7 left over (the *remainder*). Notice that the remainder (7) gives us the right-most digit (position 0) of the decimal representation of the quantity five hundred and forty-seven. The quotient (54) gives the rest of the digits in the decimal representation. We put aside the remainder (7), then divide the quotient (54) by 10, giving quotient 5 remainder 4 (digit position 1). We put aside this remainder, and divide the new quotient (5) by 10, giving quotient 0 remainder 5 (digit position 2). We now stop, because if we were to repeat the division on the new quotient we would get 0 remainder 0.

This is not an accident; consider an arbitrary decimal number  $\dots d_2 d_1 d_0$  representing the quantity:

$$\dots d_2 \times 10^2 + d_1 \times 10^1 + d_0 \times 10^0.$$

If we divide that quantity by 10, we get

$$\dots d_2 \times 10^1 + d_1 \times 10^0,$$

and the remainder  $d_0$ . Clearly if we divide the quotient by 10 again, we will get

$$\dots d_2 \times 10^0,$$

and the remainder  $d_1$ .

In general, when one natural number,  $n$ , is divided by another natural number,  $d$ , we get a quotient,  $q$ , and a remainder  $r$ , such that  $r < d$ . This procedure can clearly be followed for any natural number greater than 1. Thus, if we divide a quantity repeatedly by the base of a positional number system, until we are left with a quotient of 0, we obtain as remainders all the digits of the representation of that quantity in the number system.

Suppose we wish to represent the quantity 46 as a binary number. We start with 46, and we divide by 2 over and over again until we end up with a quotient of 0:

Quotient	Remainder
46	
23	0
11	1
5	1
2	1
1	0
0	1

The binary representation is captured in the column marked *remainder*. If we continue dividing by 2 the table will continue to grow downwards, and any further rows of the table are bound to contain only 0s (because 0 divided by 2 is 0 remainder 0). If we now read the *remainder* column from the bottom upwards (remembering that the first division produced the rightmost digit as the remainder) we obtain 101110 as the binary representation of 46.

### 3.4.7 Binary to decimal

Converting from a binary to a decimal representation, presents even less challenge. By definition of the base- $b$  positional number system, a digit in position  $k$  has the corresponding weight  $b^k$ , in the case of binary this is  $2^k$ . To obtain the total quantity represented by the number, we multiply the digits by their weights and total up the results. For the binary number 101110 the result is  $1 \times 32 + 0 \times 16 + 1 \times 8 + 1 \times 4 + 1 \times 2 + 0 \times 1 = 46$ . Simpler still, because  $b = 2$ , the digits are either 0 or 1. We can ignore the 0's, so for the binary number 101110 the result is  $32 + 8 + 4 + 2 = 46$ .

### 3.4.8 Base eight numbers (Octal)

It is already noticeable that binary numbers tend to be long even when they express small numbers, such as 46. That is due to the fact that the base is so small, so it takes high powers of the base to represent even moderate quantities. To remedy this, we can use a representation in a higher number base. But what we want is a representation where it is easy to see the correspondence between its written form and the bit-string representation that we will find in Platform 1. For this reason we choose a number base that is a power of 2. For example, the octal (i.e. base 8) system. We can use the familiar numerals 0,1,2,3,4,5,6,7 as octal digits (but note that we no longer have the digits 8 and 9).

We can convert the quantity forty-six, which serves as the running example in this section, from the decimal number 46 into an octal number using repeated division by 8.  $46 \div 8 = 5$  remainder 6, and  $5 \div 8 = 0$  remainder 5, so 46 (decimal) is represented in octal as 56.

The conversion between octal and binary numbers does not involve any division. Indeed, we can simply replace octal digits by their binary values, so

$$\begin{aligned} 56 & (\text{octal}) \\ &= 5 \times 8 + 6 \times 1 \\ &= 5 \times 2^3 + 6 \times 2^0 \\ &= (1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0) \times 2^3 + (1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0) \times 2^0 \end{aligned}$$

We then multiply out the brackets, to give

$$= 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0,$$

which is a perfectly valid expansion of the binary number 101110. This works for any octal number, and it works both ways: to convert from octal to binary we replace each octal digit by the corresponding 3-bit binary string and to convert from binary to octal we start from the least significant bit, dividing the string into 3-bit groups, and replace each group by an octal digit<sup>16</sup>. The reason why this is possible between base 2 and base 8, and not possible between base 2 and base 10 is that  $8 = 2^3$ , whereas 10 is not a power of 2.

Octal numbers can be used as a more concise way to write bit-strings, and need not necessarily be interpreted as numbers. However, to achieve a one-to-one correspondence between bit-strings and octal numbers, the bit-strings have to have a size divisible by 3, which turns out to be inconvenient.

---

<sup>16</sup> It may be necessary to pad the left-hand end of the binary string with zeros to increase its length to a multiple of 3, which is a perfectly correct thing to do if the string is a number

### 3.4.9 Base sixteen numbers (Hexadecimal)

Computing platforms are not just binary, their every dimension (such as the length of the bit-strings representing memory, registers, persistent storage, or the operands of an arithmetic operation) tends to be a power of 2. Clearly no power of 2 is going to be divisible by 3. So if we adopted the octal system for describing platforms we would have to deal with incomplete octal digits. This involves having to split some octal digits down the middle, which is possible (as the example of the Digital PDP-11, a landmark in the history of computing platforms, convincingly demonstrates), but it is far from being ideal.

There are two ways of alleviating this difficulty. We could employ base-4 representations ( $4 = 2^2$ , all perfectly binary), as we observe in the encoding of information in DNA <sup>17</sup>. That would be a fitting contribution to culture, but unfortunately not very efficient. It would only reduce the length of digit-strings by half. Anything to do with a modern 32 bit platform would still require 16 quaternary (base 4) digits to express.

The other alternative is to seek the next power of two above 8, rather than below. After all, 8 is almost 10, so it would be helpful to employ octal from the point of view of human convenience, if it were not for the fact that each digit is represented by a 3 bit string. The power of 2 closest to 10 on the other side of it is 16, and it is common for human beings to employ a base-16 positional system when describing platforms. This is called *hexadecimal* or hex for short.

### 3.4.10 Hexadecimal digit strings

Base-16 presents an additional challenge over that of base-8. There are sixteen basic values, each of which must be represented by a different symbol. So we require sixteen distinct digits before we are able to write any numbers in hexadecimal notation.<sup>18</sup> We already have the conventional ten digit symbols, 0-9 and we choose to reuse them with their associated quantities. Rather than invent new symbols (which probably wouldn't be on a keyboard) we borrow the remaining six symbols from the beginning of the Roman alphabet: A-F or a-f (either uppercase or lowercase is acceptable). The basic quantities associated with these symbols are: ten (A), eleven (B), twelve (C), thirteen (D), fourteen (E) and fifteen (F).

### 3.4.11 Conversions

The conversion between binary and hexadecimal is just as easy as between binary and octal. To convert from binary number *to* a hexadecimal number, replace each consecutive group of four binary digits by one hex digit starting from the right-most four. For example, the binary 10010011 is 93 in hex, and 00111011 works out as 3B. Similar to octals, any size binary *number* can be represented as a hex number.

The length of a hexadecimal number is only about a quarter of the length of the equivalent binary number, which is quite convenient in practice. However, if we wish to use hex as a shorthand for a *bit-string* which is not a number, we must demand that the string is of a length which is a multiple of four: 4 bits, 8 bits, 12 bits, etc. Importantly 8, 16, 32 and 64 are all multiples of 4; these numbers correspond to fundamental bit-string sizes in modern computing platforms, and it is easy to create hex representations that correspond directly to bit-strings of these lengths. As a result, hex is used very broadly, and we shall use it in this text a great deal.

The main use we will make of hex will be *non-numerical*. Even though hex numbers are what the word suggests, numbers, we will rarely use them for arithmetic operations, such as addition, subtraction etc. We are however, going to use them routinely as a shorthand for the corresponding bit-strings, when the latter represent anything at all, not necessarily numbers. So whilst it is important to be able to convert between hexadecimal and decimal (and vice versa), it is *more important* that you can convert between hex and binary.

### 3.4.12 Avoiding confusion between number bases

We are so used to reading and writing decimal numbers that we automatically assume that every digit string is a number represented using the decimal positional system. But this isn't so. Because octal employs the

---

<sup>17</sup> That's a topic for a whole different book!

<sup>18</sup> This is *still* better than the 60 digit symbols required by the Babylonians

symbols 0..7, and hex employs the symbols 0..9 it is easy for a human being to confuse an octal number or a hex number for a decimal number, and vice versa.

The hex number 42, the octal number 42, and the decimal number 42 all represent different quantities. In most cases in this text, when we employ a string of conventional digits (such as 42) as a number we mean the *decimal* number. But how should we deal with those situations where there may be confusion as to which of them we mean?

Where there is a possibility of confusion mathematicians add a subscript to the right-hand end of the digit string to indicate the number base in use. So the quantity twenty seven may be represented as  $27_{10}$  (base ten), or as  $11011_2$  (base two), or as  $33_8$  (base eight), or as  $1B_{16}$  (base sixteen), or even as  $10_{27}$  (base twenty seven).

This is very flexible, but not very easy to reproduce in the simple text editors that are typically employed by programmers. So in computing we introduce a set of conventions for writing numbers in the number bases that we find most useful. There is no set standard, but there are a number of conventions that are well understood. We will be adopting such a set of conventions in this text.

When we use a digit string and do not say otherwise it will represent a decimal number, so the string 42 should be taken to represent the quantity forty two. However, there will be cases where a digit string may have more than one possible interpretation. For example, the string 1001 could be a decimal representation of the quantity one thousand and one, or a binary representation of the quantity nine (what quantity would this string represent if it were an octal number, or a hex number?).

In order to avoid confusion we will add a prefix to those digit-strings that represent non-decimal numbers, unless it is clear from the context that a certain non-decimal representation is being used.

- **Binary.** For binary numbers the prefix will be ‘0b’ (zero *b*). Because of this we can assume that 1001 (a digit string without a prefix) represents one thousand and one, whilst 0b1001 represents nine. 37 has no possible interpretation as a binary string, because the digits 3 and 7 are not available in binary. We will write the string 0b100101 when we mean the binary number representation of the quantity thirty seven.
- **Hexadecimal.** For hexadecimal numbers we will use the prefix ‘0x’ (zero *x*). The digit string 0x1001 represents the quantity four thousand and ninety seven, the digit string 0x37 represents the quantity fifty five, and the quantity thirty seven is represented by the string 0x25.

There are other ways to identify which number system is in use, but we have chosen these conventions because they are used in many programming languages, including CdM-8 Platform 3<sup>1/2</sup> assembly language.

## 3.5 Addition of non-negative numbers

When human beings use a positional number system they need not place any restriction on how many digits a number may have. We can add together any two numbers that we can write down, even if they are of different lengths. However, the binary positional system used by a platform has to be limited due to the finiteness of the platform. There are a number of problems with variable-length number representations:

1. If there is no limit to the length of a number we need an infinite platform.
2. Programs may need to set aside memory space to store each number, and if a number can be of any length we cannot know how much space needs to be set aside until the number is entered by a user.
3. To store a sequence of numbers we use a bit string made up of their bit strings. If each number can be of different length how does the machine know where one number ends and the next begins?
4. It is more difficult to build hardware that performs arithmetic efficiently on numbers of different lengths.

### 3.5.1 Digit strings of fixed length

In practice we restrict ourselves to using binary number representations of fixed length, and binary numbers are commonly referred to using that restriction explicitly: an 8-bit number, a 16-bit number, etc. Clearly, leading zeros (zeros at the left-hand end) do not change the interpretation of a number as a quantity, so the fact that we *can* represent the quantity 27 using a bit-string of length 5 (11011) does not stop us from representing it using a bit-string of length 8 (00011011). This cuts both ways, and the decimal number 0027 can *still* be represented using the 8-bit string 00011011.

More generally, if we are employing  $k$ -bit binary numbers, and we number the bits from the right-hand end, the most significant bit will be in position  $k - 1$ , with weight  $2^{(k-1)}$ , and the range of whole number quantities that can be represented is  $0 \dots 2^k - 1$ . Any number that is within this range would have zeros in all positions from  $k$  upwards, and there is absolutely no point in attempting to capture this fact, as these zeros have no effect on its value. However, a number that is less than  $2^{(k-1)}$  *must* have a 0 in position  $[k - 1]$  (the left-most position), because we are using a  $k$ -bit representation, and all bits in the string must have a value. Similarly, the quantity 1 will be represented by a bit-string of length  $k$ , which has a 1 in the right-most position (position 0), and 0s in all the other positions ( $1..(k - 1)$ ), because no part of the bit-string may be empty. Thus we end up with a fixed-length ( $k$ -bit) representation for each quantity.

CdM-8 employs 8-bit fixed-length representations of numbers, which makes life much simpler for those designing and implementing the platform. This limits us to the whole numbers in the range 0..255, which seems to be a ridiculously small set, but there is a great deal that can be done with just this set, and we can write programs that deal with larger sets of numbers by using more than one 8-bit string to represent a number.

### 3.5.2 Adding together positional numbers

We promised earlier that the positional system will make addition (and later subtraction) efficient. Now we are going to demonstrate how it can be done. Let us first remember how two decimal numbers of the same length are added together:

$$\begin{array}{r} 217 \\ + 535 \\ \hline 01 \\ \hline 752 \end{array}$$

The 1 above the middle digit of the result (5) represents the ten carried over from units (column 0) to tens (column 1). In column 0 we have  $7 + 5 = 12$ , so we write 2 at position 0 in the result and carry 10. The 1 (taken as a unit) is added to the second digit (column 1) of the sum:  $1 + 3 = 4$ , add 1, the answer is 5.

The 0 above the left-most digit of the result represents the amount carried over from tens (column 1) to hundreds (column 2)

The addition algorithm has the following remarkable properties:

- Each pair of digits in the same column is added independently, except:
- 1 can be received from the neighbour on the right, which needs to be added to the result, and
- if the result proves greater than the base (ten in this case), the excess over the base is carried left to the next column (in the form of a 1).

In the standard platform terminology the 1 or 0 received from/passed to a neighbour is called a *carry*. There are consequently two carries to a column under addition: the *carry-in*, which is the carry received from the neighbouring column on the right and the *carry-out*, which is the carry passed to the neighbouring column on the left. Both can only be 0 or 1 no matter what base we use (isn't it fortunate for a binary system, which has just those two?). A carry of 0 from a column means that the total of all digits in the column plus possibly the carry-in is less than ten.

### 3.5.3 Adding together binary numbers

We can add together two binary numbers in *exactly* the same way as two decimals. In doing so we should remember that in binary  $0 + 0 = 0$  and  $0 + 1 = 1 + 0 = 1$  as usual, but  $1 + 1 = 10$ , which means that the result is 0 and the carry-out is 1. Of course in each column we will need to add the digits from the two numbers and the carry-in value for that column. When the carry-in is 0 it makes no difference to the column total, but when the carry-in is 1, we could be adding  $1 + 1 + 1 = 11$ , in which case the result is 1 and the carry-out is 1 also.

Carry-in	Digits	Sum	Carry-out
0	0+0	0	0
0	0+1	1	0
0	1+0	1	0
0	1+1	0	1
1	0+0	1	0
1	0+1	0	1
1	1+0	0	1
1	1+1	1	1

For instance:

$$\begin{array}{r}
 0\ 1\ 1 \\
 +\ 0\ 1\ 1 \\
 \hline
 1\ 1 \\
 \hline
 1\ 1\ 0
 \end{array}$$

which is how  $3 + 3$  makes 6 in 3-bit binary.

### 3.5.4 Overflow events

There is, in fact, a carry-out from the leftmost digit in binary addition, which, if it is equal to one, indicates that the sum is larger than any number representable as a bit-string of the chosen size. If there is carry-out of 1 from the most significant bit it causes an event called an *overflow*. Here is an example in 4-bit binary:

$$\begin{array}{r}
 1\ 0\ 1\ 1 \\
 +\ 0\ 1\ 1\ 1 \\
 \hline
 1\ 1\ 1\ 1 \\
 \hline
 0\ 0\ 1\ 0
 \end{array}$$

Clearly, when an overflow occurs it means that the result that the addition algorithm produced is incorrect. In this example the result is 0010, which represents the quantity two. However, the numbers added together were 1011 (eleven) and 0111 (seven), and  $11 + 7 = 18$ , rather than 2. Detection of an overflow event is easy for the platform: all it needs to do is note the carry out of the left-most digit in any computation of the sum. For natural numbers a non-zero carry-out means that an overflow event has occurred.

### 3.5.5 Bit slicing

Before we move on to look at subtraction, we make a short detour. Imagine a platform that has the ability to add 1-bit numbers. Could two consecutive additions be used to somehow add together two 2-bit numbers? The answer is yes, provided that the operation  $+$  could somehow take not just the two numbers (called *operands*) but also a carry-in and then add all three together. This principle scales up too, so a platform that can add together two 2-bit numbers can be used to add together two 4-bit numbers,etc. Suppose we had a platform that can add together 2-bit numbers, we could organise the above example as follows:

$$\begin{array}{r} 11 \\ + 11 \\ \hline 1 \ 1 \\ \hline 10 \end{array}$$
  

$$\begin{array}{r} 10 \\ + 01 \\ \hline 1 \ 1 \\ \hline 00 \end{array}$$

The first 2-bit addition represents the two least-significant columns (right-most two bits) in the 4-bit addition sum, and the second 2-bit addition represents the next two columns to the left. The carry-out of the first addition is taken as the carry-in of the second.

Such organisation of a binary operation on longer strings is known in the platform world as *bit slicing*: all the numbers are split into slices of a smaller size and the addition operations on them are chained by equating the carry-out of one addition to the carry-in of another. Using bit slicing, a platform is able to add binary numbers of any size as long as it has the resources to do so (memory to keep the result and time for it to be computed) while still remaining a proper platform 2 with a fixed size bit-string as the only kind of data available for immediate manipulation.

### 3.5.6 Big- and little-endianess

These terms go back to Jonathan Swift, an English novelist, and refer to the fruitless debate in a fictional society where some members believed a soft-boiled egg should only be cracked at the big end, and others believed that the little end was the one to use <sup>19</sup>. The history of it is not very well remembered in the computer science community, where debates of this kind rage for decades, but the particular egg of discord that causes the term to continue to be used in computing is to do with a very specific situation that arises when number-bearing bit-strings occupy multiple chunks of memory.

Imagine a 32-bit string accommodated on a platform that can only handle chunks of 8 bits at a time (such as our own Platform 3<sup>1/2</sup>). The string will occupy 4 chunks, and bit-sliced operations will have to be applied to process it. The chunks themselves are usually numbered (this is called addressing), so the 32-bit string will have chunks 0, 1, 2, and 3 covering it. As soon as each chunk is numbered, it is necessary to choose which chunk gets which part of the 32-bit number. One approach is to make chunk 0 (i.e., that with the offset zero from the beginning of the string) the “least significant” and chunk 3 the “most significant” – as bit positions would be in a number. However, if we were to write a 32-bit string out in full we would expect its most significant digits to be at the left, in positions 24–31, i.e., in the *first* chunk, which we’ve just decided to call chunk 0:

31-24	23-16	15-8	7-0	Big Endian
0	1	2	3	

The arrangement above is one of the two contiguous ones, and it is called *big-endian* since the number is laid out across the chunks big end first. The other arrangement is called *little-endian* and it is the opposite:

7-0	15-8	23-16	31-24	Little Endian
0	1	2	3	

so the number is laid out little end first. Each chunk contains a bit-string which is a number in its own right, so positions in it are enumerated right-to-left, from 0 to 7. In a way the big-endian solution seems more logical and consistent with bit-position enumeration, but the little-endian approach is more consistent with chunk addresses: junior bits of the 32-bit number are accommodated in junior addresses.

This appears even more striking when hexadecimal representations are used for the different bit-strings. For example, if a 2-byte big-endian representation is used, the decimal number 5376 is stored as 00010101 00000000 in binary, which can be converted to 1500 in hexadecimal, whereas if a little-endian representation is employed the same decimal number is stored as 00000000 00010101 in binary, which can be converted to 0015 in hexadecimal. It is important not to confuse the order in which a machine numbers the bytes in a multi-byte representation of quantities with the way we represent numbers on paper (or in programs). In the latter case the number is written in a standard positional system, which represents a whole bit-string, rather than one split into chunks; consequently the issue of endianness does not arise.

Unfortunately the big-endian/little-endian debate continues to this day. Some platforms are little-endian, while others are big-endian. Even when a platform is capable of taking the whole 32-bit (or even 64-bit) number in *at once* and processing it, it is usually the case that it still splits its data space into 8-bit chunks for processing data types other than numbers (characters, for instance, which will be discussed later in this chapter, and which can take as little as one chunk each). Consequently the platform has to provide addressing for those small chunks as a basis, and so endianness is as important now as it was in the 1970’s when IBM mainframes were big-endian, and PDP minicomputers were not. Nowadays endianness can even be switchable (as is the case with ARM processors found in most smartphones and tablets), so that the platform can easily handle data items that were created elsewhere.

---

<sup>19</sup>This comes from Swift’s most famous novel: Gulliver’s Travels. The people of Lilliput are required by royal decree to crack open their soft-boiled eggs at the little end, and the inhabitants of the rival kingdom of Blefuscu crack open their eggs at the big end

## 3.6 Subtraction and negative numbers

Multiplication and division of positive whole numbers can be achieved by repeated addition, but arithmetic without subtraction is fundamentally incomplete. A particular complication arises when we are working exclusively with natural numbers: it is impossible to define even an infinite set of natural numbers on which subtraction is effective. Pick any two natural numbers  $x$  and  $y$ . If  $x - y$  is a natural number, and if  $x \neq y$ , then  $x > y$ , so  $y - x$  cannot be a natural number because it must be negative.

It quickly becomes clear that we will have difficulty with subtraction if we have no way of representing negative numbers in binary strings. By itself this is not a problem: since a bit is perfectly capable of representing the presence or absence of a minus before the number, all we need to do is pick a bit-position within a given length of binary string that will be used to indicate the sign of a number. That way we will not attempt to interpret the bit in this position as one of the value digits in the positional system.

### 3.6.1 Signed-magnitude representations

We can assign the left-most position in a bit-string to represent the sign of the number, with a 1 in this position indicating that the binary number is preceded by a minus sign and a 0 that it is not. For example, if we have only 3-bit strings to represent numbers, a 0 in position 2 would mean that a number is *positive* and a 1 in that position would mean that it is *negative*. We call this the *sign bit*. So we interpret 111 as  $-3$  on the basis that the sign bit is 1 (the number is *negative*), and the magnitude of the number is represented by 11 (which stands for the quantity three).

If we adopt this approach we obtain a set of binary strings that correspond to the quantities  $-3, -2, -1, -0, +0, +1, +2$  and  $+3$ . A slight imperfection here is the fact that the single quantity zero has two different binary representations: 000 and 100, corresponding to the decimal representations  $+0$  and  $-0$ . So this form of binary representation is inefficient. However, if the size of string were chosen to be large enough, say 16-bit or 32-bit, the loss of efficiency associated with the aforementioned information redundancy would be so tiny that it would matter very little.

This means of representing *signed integers* (as opposed to the unsigned integers we had before) is in fact quite standard and is known as a “signed-magnitude” representation. Here the quantity represented by the bit-string with the sign bit removed is called the *magnitude* of the number.

The problem with signed-magnitude representations is that the platform would need two *different* algorithms to add numbers together. To calculate  $x + y$  the engine would first have to determine the signs of the two operands,  $x$  and  $y$ . If they have the same sign, the result has that sign too, and the magnitude of the result is the sum of their two magnitudes,  $|x| + |y|$ . If the two operands have different signs the engine needs to subtract the smaller magnitude from the larger and the sign of the result will be the same as the sign of the number with the larger magnitude.

Subtraction,  $x - y$ , is possibly even more complicated. If both operands are positive subtract  $|y|$  from  $|x|$  to get both sign and magnitude, if both are negative subtract  $|x|$  from  $|y|$  to get both sign and magnitude. If they are of different signs, add the two magnitudes together,  $|x| + |y|$ , and take the sign from the first operand,  $x$ . Phew!!

By using a different representation, however, it is possible to create a single simple algorithm to perform both addition *and* subtraction. In this representation it is possible to add operands blindly, using the same procedure no matter what sign each of them has. Better still, the procedure is exactly the same as that used for the addition of unsigned numbers. Such a drastic simplification has led to the broad adoption of the representation in question by all low-level computing platforms. This representation scheme is known as *two's complement*.

### 3.6.2 Complement representations of signed numbers

First we will illustrate the idea of complement representation of signed numbers in the decimal system. Let us imagine that we have to represent signed decimal numbers positionally in a string of fixed length 3.

The standard way to do this would be to introduce two new symbols ('+' and '-') and adopt a signed-magnitude representation, so that +19 represents positive nineteen and -19 represents negative nineteen.

This is pretty inefficient. We have 12 symbols at our disposal (0,1,2,3,4,5,6,7,8,9,+,-), so there are  $12^3$  ( $=1728$ ) different digit strings available in representation, but only 200 are ‘legal’ (-00 .. -99 and +00 .. +99).

Now imagine that we have no positive or negative signs at our disposal: just the ten decimal digits. There are  $10^3$  ( $=1000$ ) digit strings available in representations, but how do we capture the fact that some of them will encode positive quantities, and others will encode negative quantities? So we devise a 3-digit number representation that encodes both positive and negative.

The **key idea** of complement representations is that each  $n$ -digit string that represents a *positive* number has a complementary  $n$ -digit string that represents a *negative* number with the same magnitude, and the conversion between the two is performed by simple arithmetic operations. We set aside half of the  $n$ -digit strings for the non-negative numbers, and the other half for the negatives. All we need then is a simple means of obtaining negative number representations from positives, and vice versa. One of the simplest ways to do this is to use “standard” number representations for non-negative values less than  $10^n/2$ , and to replace each digit by its *complement* when representing a negative value: each 0 is replaced by 9, 1 is replaced by 8, 2 by 7, 3 by 6, 4 by 5,etc. This gives us the *nine’s complement* of the original. Notice that the nine’s complement of the nine’s complement of a number is that numbers, just as  $-(-x) = x$  for any  $x$ .

For example, under a 3-digit nine’s complement representation scheme the strings 000 - 499 represent the decimal numbers 0 to 499 as we might expect, but the strings 500 - 999 represent their negatives. The two 3-digit numbers 499 and 500 are each other’s nine’s complement (representing the quantities 499 and -499 respectively), as are the two 3-digit numbers 941 and 058 (representing -58 and 58). As with signed magnitude there are two representations for zero (000 and 999), which is not ideal. Nine’s complement representation schemes work, but there’s a better way to do the job. It’s known as *ten’s complement*, for reasons that will become clear later.

### 3.6.3 Ten’s complement decimal numbers

First we assign the bottom half of the number range  $0 \dots 10^n - 1$  to non-negative quantities under the standard interpretation, just as we did for nine’s complement. And once again we use the top half of the range to encode negatives. But this time we use a different technique to obtain complementary representations.

Any  $n$ -digit ten’s complement number may be negated by reading it as if it were an unsigned number and subtracting it from  $10^n$ . We call this the ten’s complement of the number. So if  $x$  is an  $n$ -digit number, the ten’s complement of  $x$  is  $10^n - x$ .

For example, in a 3-digit ten’s complement scheme the string 027 encodes the quantity twenty seven, just as it does when we are working with unsigned numbers. To find the encoding of the quantity -27 we perform the subtraction  $1000 - 27$ , giving the 3-digit number 973. Notice that we can get back to positive twenty seven by performing the same operation on the encoding of -27, because  $1000 - 973 = 027$ . This is very useful: in this representation  $-(-x) = x$  in all cases,<sup>20</sup> as it was under nine’s complement, since  $1000 - (1000 - x) = x$  and both  $x$  and  $(1000 - x)$  are representable in three digits without using a sign.

Clearly the *identification* of quantities in this representation presents no problem either way: a string  $nnn$  where  $nnn$  falls within the range from 500 to 999 represents a negative number, so we subtract  $nnn$  from 1000 to get the magnitude of the number and use the negative sign, whereas a string in the range 000 to 499 is interpreted as written, according to the unsigned decimal rules.

The representation thus covers the full range of quantities from *negative five hundred* (represented as the string 500) to *positive four hundred and ninety nine* (represented as the string 499), and there is only one encoding of zero. This is known as the 3-digit ten’s complement representation scheme, and it can be seen in figure 3.1.

---

<sup>20</sup> Except one. The number -500 has a 3-digit ten’s complement representation, but the number 500 is outside the range, and so it does not.

Quantity	Representation
499	499
498	498
...	...
2	002
1	001
0	000
-1	999
-2	998
...	...
-499	501
-500	500

Figure 3.1: Ten's complement representation

### 3.6.4 Ten's complement addition and subtraction

In this representation we are able to use the addition algorithm to perform both addition and subtraction. Suppose we want to compute the sum of the two 3-digit ten's complement numbers  $x$  and  $y$

- **If both quantities  $x$  and  $y$  are positive**, their sum ( $x + y$ ) will be correct in the new representation using the old algorithm (unless it is greater than 499).

For instance, if  $x = 023$  and  $y = 114$ , we get  $023 + 114 = 147$ , which works for 3-digit ten's complement as well as standard unsigned decimal. How would the platform detect an overflow?

- **If the quantities  $x$  and  $y$  have different signs**, they are represented by numbers in the two different ranges (0..499 and 500..999), and can be added together in the same way as two positive numbers.

When the sum of the two quantities is negative ( $x + y < 0$ ) we get the correct answer directly. So when  $x = 53$  and  $y = -99$  we add 053 to 901 ( $= 1000 - 99$ ), to give 954, which is the 3-digit ten's complement representation of -46.

However, when the sum of the two quantities is non-negative ( $x + y \geq 0$ ) the result may be greater than 999, and we subtract 1000 to get the right answer. For example, when  $x = 28$  and  $y = -13$  the 3-digit ten's complement representations are 028 and 987, and  $028 + 987 = 1015$ . The extra thousand is the carry-out from the left-most column, which can apparently be ignored.

- **If both quantities  $x$  and  $y$  are negative**, we perform the addition  $(1000 - |x|) + (1000 - |y|)$ :
- $$\begin{aligned} &= 1000 + (-|x|) + 1000 + (-|y|) \\ &= 1000 + (1000 - |x| - |y|) \end{aligned}$$

The result we get is the ten's complement representation of  $x + y$  (which is  $1000 - |x| - |y|$ ), plus a carry-out from the left-most column, which we can now see should be ignored.

For example, if  $x = -13$  and  $y = -100$ , we get  $987 + 900 = 1887$ . We throw away the carry-out, so the result is 887, which represents -113 (because  $1000 - 887 = 113$ ). However, we know that no finite representation of integers is effective; an overflow *must* occur under certain combinations of the operands. How might the platform detect an overflow? (This will be answered shortly.)

We conclude that the old algorithm (the cascaded addition of decimal numbers via a shared carry) works in all cases of the new representation. The carry-out of the most significant digit (left-most column, position 2) is no longer an indication that an overflow event has occurred, so it should simply be ignored.

Consequently we get subtraction for free. If  $x + y$  works no matter what signs  $x$  and  $y$  have, we can calculate  $x - y$  by negating  $y$  and adding it to  $x$ , because  $x - y = x + (-y)$ . So all we need to do is to change the sign of the second operand (by finding its complement to 1000) and to add the two operands in 3 digits as normal. However, it appears that we *still* need subtraction for computing  $1000 - x$ , to perform the ten's complement change of sign which we need to do when computing  $x - y$ .

### 3.6.5 Why ten's complement?

In order to change the sign of a number it appears that we need to subtract it from a power of ten. However, all we really need are a *complement* operation and an *increment* (+1) operation.

Observe that  $1000 - x = (999 - x) + 1$ . This observation makes the ten's complement of a number much easier to compute.  $999 - x$  is actually the three-digit nine's complement of  $x$ , which is obtained by replacing each of the three digits in  $x$  with its complement as we saw in the previous section. This is an easily mechanised procedure based on a tiny table. We obtain the nine's complement of a number and add 1, which is why it's called *ten's complement*.

This is a very special case of addition, which is more easily mechanised than 3-digit addition. In column 0, and in each column to its left where there is a carry-in of 1, any digit less than 9 will be replaced by the next digit up, and 9 will be replaced by 0 with a carry-out of 1. As soon as a position is reached where the carry-in is 0 the process stops, as it does when all positions have been visited. This procedure works just as well when we want to find the positive equivalent of a negative number as the other way around. For example, to negate 310 we first find its nine's complement, which is 689, then add 1 to get 690. This is the same as calculating  $1000 - 310$ .

### 3.6.6 Overflow events in ten's complement

We know there *ought to* be overflow events in any finite set of numbers under addition. However, in this representation it is no longer associated with the carry-out from the leftmost digit (when the sum of two numbers exceeds 999). It now happens for a much lower sum.

Say we add 400 and 400. The number 400 represents the quantity four hundred in ten's complement, but the sum 800 encodes the quantity minus two hundred, so if we are using a 3-digit ten's complement representation  $400 + 400 = -200$ . So when the result of adding together two positive numbers is negative we know that an overflow has occurred. How do we spot an overflow when two negative numbers are added together?

For obvious reasons an overflow event can only happen when addition is performed on two numbers of the same sign (or when subtraction is performed on two numbers of opposite signs, which is essentially the same thing). If the result represents a quantity of the opposite sign, an overflow event has occurred. In 3-digit ten's complement we compare the result with 500 to determine the sign, so detection is readily available.

### 3.6.7 Two's complement

All the above also applies in the case of binary numbers, which are more important to us here. In binary, nine's and ten's complement become one's and two's complement respectively, and the technique simplifies dramatically. In the one's complement of a binary number we replace all 1's by 0's and all 0's by 1's. To get two's complement we add 1 to the result. In fact the “obtain the two's complement” operation can be implemented using very simple hardware. Here is a table of 4-bit two's complement representations:

Quantity	2's complement (4 bits)	Quantity	2's comp
-8	1000	7	0111
-7	1001	6	0110
-6	1010	5	0101
-5	1011	4	0100
-4	1100	3	0011
-3	1101	2	0010
-2	1110	1	0001
-1	1111	0	0000

Notice that the most significant bit (position 3) can actually be used to tell us the sign of the number. When this bit is 1 the quantity being represented is negative, otherwise it is non-negative. So the most significant bit of a two's complement number is generally referred to as its *sign bit*. This is doubly convenient: we can use it to determine a number's sign, and overflow can be detected by examining changes to the sign bit.

**Subtraction.** Consequently there is no need for a special subtraction unit for 2's complement numbers. The most efficient implementation of subtraction  $x - y$  is to compute 2's complement of  $y$ ,  $\hat{y}$  and add it to  $x$  using the platform's adder unit, possibly the same unit as it contains for adding numbers. It is quite easy (and fast) to do the first part of 2's complement (i.e. bit flip) in hardware “on the fly”, as we will see in chapter 10. What might seriously slow down the operation is the second part of the 2's complement transformation: adding 1. In the case when bit-sliced *subtraction* is not supported, the most efficient way of supporting subtraction is to set the adder carry-in bit to 1, flip the bits of the second operand as they are fed to the adder and do a normal (non-bit-sliced) addition. The adder then does the addition and the increment at the same time, in one operation. Notice that even though carry-out of the most significant bit is not important for signed addition/subtraction, it is used for comparing unsigned numbers. We will touch on this in the next section, but for now let us finish the analysis of 2's complement arithmetic with the issue of overflow detection.

**Independent subtraction.** The question of bit-slicing subtraction (as opposed to addition of unsigned numbers) is not as straightforward as it might seem. The problem is that the cost saving of having a single adder that supports both addition and subtraction is not without an additional cost. The operation of sign change is in principle bit-sliceable, since the bit-flip is strictly local to the chunk and since the increment is bit-sliceable via carry propagation. However, this means that bit-sliced subtraction requires *two* distinct stages. It could be faster, albeit more expensive in terms of circuitry, to implement subtraction as an independent operation in one stage only. This is based on the following 1-bit subtraction table:

$x$	$y$	$b_{\text{in}}$	$d$	$b_{\text{out}}$
0	0	0	0	0
0	1	0	1	1
1	0	0	1	0
1	1	0	0	0
0	0	1	1	1
0	1	1	0	1
1	0	1	0	0
1	1	1	1	1

The key concept of independent subtraction is *borrow*. It is similar to carry except borrow is the amount that we take from the left (borrow-out) or which is taken from us on the right (borrow-in). Consequently, if we are to subtract 1 from 0 and we have a borrow-in of 0, the result is a borrow-out of 1 (giving us the additional quantity 2) and a difference of 1 ( $2 - 1 - 0$ ). The same with a borrow-in of 1 would give the identical borrow-out, but the difference would be 0 ( $2 - 1 - 1$ ). Independent subtraction is fairly similar to addition, and allows an equally effective hardware implementation; the main disadvantage compared to subtraction-as-sum is that we cannot reuse the adder, requiring instead a separate subtraction unit. The main advantage is that independent subtraction can be slightly faster. Subtracting two  $n$ -bit numbers may result in a borrow of 1 out of the most significant bit, which is, clearly, an unsigned overflow.

Can independent subtraction be used on 2's complement numbers thus saving the change of sign step for the second operand? Clearly independent subtraction is valid with respect to unsigned addition, i.e. for any  $n$ -bit  $x, y$  and  $z$ , if  $x - y = z$  by  $n$ -bit independent subtraction with 0 borrow-in to the least significant bit, then  $z + y = x$  by standard addition. It follows that 2's complement numbers will be subtracted correctly as well.

In this introductory book we will assume that subtraction is *never* independent. From now on, any subtraction operation will affect a carry (not borrow) and will be implemented as addition after 2's complement change of sign.

**2's complement overflow.** As we saw in the section on 10's complement, signed overflow happens only when two numbers of the same sign are added (or of opposite signs subtracted) resulting in a value of the wrong sign. A hardware implementation of (signed) overflow may just focus on this, taking two operand signs and the result sign into a logical rule. There is a simpler way, however, of detecting overflow *at the same time* as adding signed numbers, as carry cascades right-to-left through the adder. Let us consider the two cases of overflow:

- Operands are both positive and the result is negative. In this case the sign bits of both operands are 0 and of the result 1. This means that there was a carry-in of 1 into the sign bit, and there is always a carry-out of 0 from the sign bit when two non-negatives are added. If the carry-in were of 0, there would be no overflow.
- Operands are both negative and the result is positive. In this case the sign bits of both operands are 1 and of the result 0. This means that there was a carry-in of 0 into the sign bit and there is always a carry-out of 1 from the sign bit when two negatives are added. If the carry-in were of 1, there would be no overflow.

We are beginning to think that

**the *mismatch* between the carry-in into, and the carry-out from, the sign bit of the result indicates overflow.**

Let us check this hypothesis on the remaining case: operands of opposite signs. Clearly when the operand signs are opposite the only thing that can cause the result sign bit to produce a carry-out of 1 is a carry-in of 1, since the operand signs always add up to 1. So in this case the carry-in and carry-out are always the same and, as we know, overflow does not occur when adding opposite-sign numbers. This proves our hypothesis.

The significance of the above result is in the fact that the -in and -out carry values are available from the adder anyway, and the only comparison needed here is of them, not the two-stage comparison of first the operand signs and then the operand and result signs.

### 3.6.8 Comparing numbers

In many situations we need to compare two unequal numbers to establish which one is the greater. While equality is never subject to interpretation (the requirements that the representation uniquely identifies the object and the object is uniquely represented by it take care of that), the greater/less relation does indeed depend on the class of numbers. As we see above the same bit-string that represents  $-6$  in 4-bit 2's complement also represents  $+10$  if the 4-bit string is read as an unsigned number. By itself this causes no problem, since the representation of a number is always made certain before any operation, but there is a peculiar problem that only arises when we compare numbers.

Comparison has a non-numeric result: true or false. Furthermore, it is genetically linked with subtraction: if  $a > b$  then  $a - b > 0$ , so all we really have to do for comparison is compare with 0, which is the whole lot easier than comparing two arbitrary numbers with each other. In any meaningful machine implementation of arithmetic, the operation addition will be available, and 2's complement gives us both signed numbers and an easy change of sign; it seems that the problem of comparison as such should never arise!

Not so simple, unfortunately. First of all, as we said earlier, the result of comparison is not a number. Consequently, comparison does not have to be as *effective* as subtraction; interestingly it is always *more* effective. The result of comparison is well defined even when the difference between the numbers is unrepresentable. For example, using the table from the previous section observe that  $1001 < 0111$  in 4-bit 2's complement as the bit-strings represent  $-7$  and  $7$ , respectively. However, their difference  $(-7) - 7 = -14$  is unrepresentable in 4-bit two's complement and so the subtraction would cause an overflow. It would be unreasonable to limit comparison to only pairs of numbers for which the difference is representable, since this would exclude pairs such as  $7$  and  $-7$ , where the sign of each number alone would give an immediate answer.

There exists, however, a simple solution. Observe that 2's complement overflow results in the completely wrong absolute value of the result (in this example it yields  $(-7) - 7 = 2$  instead of the correct  $-14$ , but the sign of the result is not at all arbitrary. It is always the opposite to what it would have been if the engine had not overflowed. And that is due to the fact that what actually *causes* the overflow is the carry into the sign bit, or the lack thereof (when two negative numbers are added). In either case the sign bit turns out to be the opposite to what it should be according to the correct mathematical result.

Consequently, the following is the correct method of comparing two signed numbers  $a$  and  $b$ :

1. compute  $-b$  by 2's complement
2. add  $a + (-b)$

3. if the result is positive and no overflow, **or** if the result is negative and there was an overflow, then  $a > b$ , else  $a \leq b$

Mathematically speaking  $a > b$  if and only if  $N \oplus V$ , where  $N$  is the sign bit of the result (interpreted as true/false) and  $V$  is the overflow event (true, if overflow happened, false if it did not). The symbol  $\oplus$  is exclusive-or, i.e. only one is true but not the other.

**Unsigned comparison.** We could end the section right at this point if it were not for an even more esoteric question: how to compare *unsigned* numbers. The above method is inapplicable, since the most significant bit is not the sign, and the number representation used does not have space for negatives. Yet building a separate solution in hardware for this operation alone would be undesirable.

Again, there is a simple solution here, which requires little effort to understand. Observe that if we mechanically perform 2's complement change of sign on an  $n$ -bit unsigned number  $y > 0$ , we will obtain a bit string  $\hat{y}$ , which will be meaningless given an unsigned  $x$ , but which would nevertheless add up to 0 if added to the original  $y$  using standard  $n$ -bit addition. Indeed addition does not "know" the interpretation of the bit strings it is adding. It always works according to  $y + \hat{y} = 0$ . For a nonzero  $y$  this would also result in a carry-out of 1 from the most significant bit; in a sense the result is not a 0, but  $2^n$ , and its  $n$  least significant bits represent the value 0. This means that the "2's complement" of an unsigned  $y$  is the lower-order  $n$  bits of  $2^n - y$ .

So for two  $n$ -bit unsigned numbers: an arbitrary  $x$  and some  $y \neq 0$ ,  $x + \hat{y} = 2^n + x - y$ , as far as the  $n$  least significant bits. Now if  $x \geq y$ ,  $x + \hat{y} \geq 2^n$  and the addition will produce a carry of 1 out of the most significant bit. But if  $x < y$  then  $x + \hat{y} < 2^n$ , and the addition will complete without producing a carry-out. The only exceptional case is  $y = 0$ , since in this case  $\hat{y} = y = 0$  and adding  $x + \hat{y}$  fails to produce a carry-out of 1 for all  $x > y$ , but then the process of 2's complementing  $y$ , when  $y = 0$  uniquely produces a carry-out of 1 itself, which will be picked up on by the adder (see previous section). So in all cases of two unsigned numbers  $x - y$  in 2's complement sense produces a carry-out of 1 if and only if  $x > y$ .

This remarkable result allows the platform to use the same plain adder (one that is designed to add two unsigned  $n$ -bit numbers and a carry-in, producing an  $n$ -bit unsigned result and a carry-out) for as many as five purposes:

1. unsigned addition (main function)
2. unsigned comparison (by blindly applying 2's complement to the unsigned second operand and watching the carry-out)
3. 2's complement addition (due to the commonality of the addition method with unsigned numbers)
4. 2's complement subtraction (due to the fact that signed subtraction boils down to signed addition)
5. 2's complement comparison (by combining the sign of the result and the overflow event)

The above is a remarkable practical achievement of number representation theory.

### \*3.7 Binary-Coded Decimal

The issue of *cost* underpins any decision the designer of a platform makes in selecting an appropriate representation of data. However, in this area of technology, the abstract cost of a solution very much depends on its usage. The 2's complement representation is ideal for addition and subtraction, but as we will see in the next section, multiplication is less costly in the signed-magnitude format. Let us consider an unusual situation when the cost of conversion between binary and decimal dominates computations in a program or device. This is easily the case when very few arithmetic operations are performed on numbers most of the cost is associated with exchanging data with a human. Indeed, conversion from decimal to binary is not a fast operation however implemented (usually by repeatedly multiplying by 10 in binary with subsequent addition). Binary to decimal involves division by 10, which is not very fast either, especially if there is no support for fast hardware multiplication. If all that is required is a couple of additions/subtractions per data item, as is often the case, for example, in processing accounting spreadsheets, the platform spends most of its

computing resources doing the conversions, rather than the arithmetic the converted numbers are required to be processed with.

Last century the language Cobol was introduced for those types of computations. Cobol programs were able to utilise strings of decimal digits, each encoded as a 4-bit binary string. Such a number is a decimal number from the point of view of the positional system, so the digit preceding a given one on the digit-string has 10 times the weight, but the fact that each *decimal* digit is represented as a 4-digit *binary* string makes it possible to represent the whole  $n$ -digit decimal number as a  $4n$ -bit string. This representation was called Binary-Coded Decimal or BCD.

Clearly it is very easy to convert between BCD and decimal, since all that changes is the representation of individual digits rather than the quantity under a positional system. The question is: is it possible to do arithmetic directly in BCD?

While every operation that can be implemented in binary can also be done in BCD, we will limit ourselves to just unsigned addition, which nicely illustrates the kind of peculiarities the use of BCD involves, which is sufficient for our introductory study.

Let us start with a single-digit addition. Two BCD digits,  $x$  and  $y$  can be added as 4-bit binary numbers with subsequent *BCD correction*:

- if the result is the quantity 9 or less, and the carry-out is 0, the result is valid, with no correction being necessary. Indeed this is only possible if the result quantity corresponds to a single-digit decimal, in which case there is no difference between BCD and binary addition.
- if the result is the quantity 10 or above and the carry-out is 0, subtract 10 from the result to make it valid and raise the carry-out. This is the situation when larger 1-digit decimal values are added, with the result being a 2-digit decimal less than 16. Alternatively (and more efficiently), add 6 to the 4-bit group; Since  $6 = 16 - 10$ , this will take care of both subtracting 10 and producing a carry-out of 1.
- if the binary addition has produced a carry-out of 1, the result should be corrected by adding 6 to account for the fact that we carried the quantity 16, rather than 10, over to the next decimal digit. Indeed, by adding 6 we counterbalance the subtraction of 16 (which is the effect of carrying 1 from the most significant bit in 4-bit binary) by increasing by 6 the value in the current 4-bit group. For example, if we add, 9 and 9, the result is going to be 12 in hex, with the leading 1 represented as a carry-out (weight 16), but the sum should be  $18_{10}$ , last digit 8 or  $2 + 6$ .

Combining the last two cases we come up with a simple single rule: If there is a carry-out of 1 or the sum is greater than 9 then add 6 (it may also produce a carry of 1 if it was not produced already in the course of the first addition). Given this rule, the addition operation will boil down to binary addition of 4-bit chunks (called *nibbles* in the context of BCD), and the usual cascade of carry propagation.

BCD subtraction can be achieved via either 10's complement or an independent subtraction operation; we will not discuss this in detail, leaving it to the reader as an exercise.

## 3.8 Multiplication

### 3.8.1 Decimal example

As before, we begin the analysis of binary multiplication with a decimal analogy. Here is an example of how two equal size (2-digit) decimals may be multiplied:

$$\begin{array}{r}
 & 7 3 \\
 \times & 6 5 \\
 \hline
 & 0 3 6 5 \\
 & 4 3 8 0 \\
 \hline
 & 4 7 4 5
 \end{array}$$

What we have here is a three step procedure:

**Partial multiplication.** We take each digit of the second operand in turn, starting with the least significant (right-most), and multiply it by the first operand, and place the results of each of these partial multiplications (the *partial products*) on successive lines. There will be one partial product line for each digit in the second operand, starting with the right-most (least significant) digit.

**Shift** Each partial product line is shifted to the left by a distance that increases as we work down, and the right-hand end of each string is filled in with zeros. The number of positions a partial product line is shifted to the left is the column number of the digit that produced that partial product, so the first line is shifted 0 positions to the left, the second line is shifted left by 1 position and a single zero is appended to the right-hand end, the third line (if it were there) would be shifted by 2 positions and 2 zeros would be appended,etc. Finally the left-hand ends of the strings are padded with zeros to ensure that all partial product strings are of the same length. Notice that all partial products are now twice the operand size.

**Summation** The partial product lines are totalled up using addition (in this example we need to use 4-digit addition).

### 3.8.2 Unsigned binary multiplication

Binary multiplication uses the same algorithm, but it is greatly simplified by the fact that all digits are 0's and 1's. Multiplication by 1 means copying, and by 0 means nullifying. Here is an example of multiplication of two 4-bit strings, representing the (unsigned) quantities thirteen and fourteen:

$$\begin{array}{r}
 1101 \\
 \times 1110 \\
 \hline
 0000 \\
 1101 \\
 1101 \\
 \hline
 10110110
 \end{array}$$

Once again, all calculations are done in strings of the same size. When we multiply a 4-bit number by another 4-bit number we will get an 8-bit result, so we must use 8-bit strings for each of the partial products. Each partial product should be shifted to the correct position in an 8-bit string, and padded with zeros as appropriate:

$$\begin{array}{r}
 1101 \\
 \times 1110 \\
 \hline
 00000000 \\
 00011010 \\
 00110100 \\
 01101000 \\
 \hline
 10110110
 \end{array}$$

Multiplication of unsigned numbers is pretty straightforward. When we calculate the product of two  $n$ -digit binary numbers we generate  $n$  partial products, each of which is padded with zeros to give a  $2n$ -digit number. These are then added together to give a  $2n$ -bit result. The result in this case is the 8-bit binary number 10110110, which represents the unsigned value one hundred and eighty two ... the *correct* result.

### 3.8.3 Multiplication of signed numbers

Things change significantly when we switch to signed numbers. If a signed-magnitude representation is being used we first have to remove the sign bit from each number. Then we can multiply the two unsigned numbers using the partial product method. But what sign should we give to the result? Simple: the result will be negative whenever the two operands have different signs (one negative one positive), and it will be positive whenever the two operands have the same sign (both positive or both negative).

Suppose our two bit-strings 1101 and 1110 are 4-bit signed-magnitude binary numbers. The decimal equivalents are  $-5$  and  $-6$  respectively. We remove the sign bit in each case, giving the two bit-strings 101 and 110 (representing 5 and 6), which we multiply together using the same technique as before:

$$\begin{array}{r} 101 \\ \times 110 \\ \hline 000000 \\ 001010 \\ 010100 \\ \hline 011110 \end{array}$$

In order to get the correct 8-bit signed-magnitude result we need to pad this number to the left with zeros, and then set the sign bit to 0 or 1 as appropriate. In this case the sign bit is 0, so we get the result 00011110, which represents +30.

What happens when a *complement representation* is used for signed numbers? It's really easy to see the problem when we try multiplying ten's complement numbers together. Suppose we have the two quantities 5 and  $-2$ . The two operands are represented in 3-digit ten's complement as 005 and 998, and the result of multiplying these two is 004990. This is the six-digit ten's complement representation of the number 4990, rather than  $-10$ , which is represented by 999990. And suppose we use this technique to multiply  $-2$  by itself. The result should be 000004, but when we calculate  $998 \times 998$  we get 996004, which is the 6-digit ten's complement representation of  $-3996$ . So how can we deal with this?

The problem is that a negative number  $(-|x|)$  is represented as  $1000 - |x|$  in 3-digit ten's complement form.

$$(-|x|) \times |y| = -|xy| \quad \text{but} \quad (1000 - |x|) \times |y| = 1000 \times |y| - |xy|$$

Even worse,

$$-|x| \times -|y| = |xy| \quad \text{but} \quad (1000 - |x|) \times (1000 - |y|) = 1000000 - 1000 \times |x| - 1000 \times |y| + |xy|$$

It doesn't take a genius to see that the result of multiplying two ten's complement numbers together will usually be wrong when one or both of them represents a negative quantity.

### 3.8.4 Sign extension

We can solve this problem by employing a technique known as *sign extension*. We know that the result will be a six-digit ten's complement number, so we pad both operands to the left to make each of them into a six-digit ten's complement representation. Where the number represents a positive quantity we obviously extend it by padding with leading zeros, but where the number is negative we extend it by padding with leading nines.<sup>21</sup> Then we perform unsigned multiplication on the two numbers, and ignore any digits that appear to the left of position 5. Try it for yourself with the two examples we gave above ( $5 \times -2$  and  $-2 \times -2$ ).

Sign extension works fine, but it means calculating twice as many partial products as before, and performing more additions. There is actually a simpler solution, which is to make a note of the signs of the two operands, negate any negative operands to give their positive equivalents, perform an unsigned multiplication, and then correct the sign of the result, just like we did in the signed-magnitude case. If the two operands have the same sign we do not need to do anything because the result is positive. If the two operands have different signs we work out the ten's complement of the result to get the correct answer.

### 3.8.5 Two's complement multiplication

Multiplication of two's complement numbers presents similar problems. Returning to our example of 4-bit binary operands, a negative  $x$ ,  $x = -|x|$ , would be represented as a complement, i.e.  $2^4 - |x|$ . If we are expecting an 8-bit result from this, and if we assume as we should that the result is also in the two's

<sup>21</sup> Can you see why this works? Imagine the quantity is  $-8$ . The 3-digit ten's complement representation of this is  $1000 - 8 = 992$ , the 4-digit representation is  $10000 - 8 = 9992$ , the 5-digit representation is  $100000 - 8 = 99992$ , etc.

complement representation (all be it an 8-bit two's complement) we are bound to be disappointed. Indeed, for a positive  $y$ ,  $y = |y|$ ,

$$(2^4 - |x|) \times y = 2^4 \times y - |x| \times y,$$

which is not at all the same as

$$2^8 - |x| \times y,$$

which we expect from an 8-bit two's complement result. Just as in the ten's complement case, because the addition and the shifting left (multiplication by powers of two) are all done in the 8-bit format, the two operands need to be transformed to 8-bit representations so that the magic of complement arithmetic works. Let's assume we wish to multiply  $-3$  by  $-2$ , and these quantities are represented using 4-bit two's complement, we get:

$$\begin{array}{r} 11111101 \\ \times 11111110 \\ \hline 00000000 \\ 11111010 \\ 11110100 \\ 11101000 \\ 11010000 \\ 10100000 \\ 10000000 \\ \hline 00000110 \end{array}$$

The operands are converted to 8-bit representations by *extending* the sign bit (most significant bit) to the left, and the answer we get is the 8-bit two's complement representation of the quantity six. In this case each operand is negative so the 4 extra spaces to the left are filled with 1's, but if they were positive the spaces would be filled with 0's. This is obviously correct for positive numbers (padding to the left with zeros adds nothing), but is not so obvious that it will work for negative numbers.

If we look at the two negative operands we see that each of them can be negated again to obtain a positive number with the same magnitude, just by obtaining its two's complement. This works just as well for an 8-bit representation as it does in 4 bits. So when we pad a negative  $n$ -bit two's complement number to the left with a string of  $k$  1's we get an  $(n+k)$ -bit two's complement number that represents the same quantity.<sup>22</sup>

Now let's look at the multiplication process. Not only do we have 8-bit partial products as before, there are now going to be 8 of them, which means two's complement multiplication requires much more work as unsigned binary multiplication.

So, just as in the decimal case, it is better to note the signs of the two operands to determine whether the result should be positive or negative<sup>23</sup>, and then convert each of them to a positive number of the same magnitude and use unsigned multiplication to work out their product. Then we use the signs of the two operands to determine the correct sign for the result, and obtain the two's complement of the product if the result should be negative.

---

<sup>22</sup> The correct positive number is the one we would get by left-padding the  $n$ -bit positive version of the number to  $(n+k)$  bits with 0's.

<sup>23</sup> A simple way to do this is to take the logical *exclusive-or* (xor) of the two sign bits ( $A \text{ xor } B$  is 1 if  $A$  and  $B$  are the same, and 0 if they are different). If it is 0 the result sign is positive and if it is 1 the result sign is negative.

## 3.9 Division

Note that we are only interested in *whole number* division. When one whole number is divided by another we will always get a whole number *quotient* and a whole number *remainder* (just like division at primary school):

### Definition 7 : Terms used in division

In *any* division calculation

- the **dividend** is the number to be divided
- the **divisor** is the number the dividend is divided by
- the **quotient** is the main result of division,  
though see below.

In the formula  $q = a \div b$ , the dividend is  $a$ , the divisor is  $b$ , and the quotient is  $q$ .

A **whole number** division calculation gives two results:

- a **quotient**, which is the *whole number* of times the divisor ‘goes into’ the dividend. In other words, the quotient is the maximum integer that if multiplied by the divisor gives the result not exceeding the dividend.
- a **remainder**, which is the quantity left over, i.e. the difference between the dividend and product of the quotient and the divisor.

The whole number division calculation  $a \div b$ , gives a quotient,  $q$ , and a remainder,  $r$ , such that  $a = qb + r$  (*dividend* = *quotient*  $\times$  *divisor* + *remainder*), where a valid remainder is required to be strictly less than the divisor:  $r < b$  (if it isn’t, we can always increase the quotient and reduce the remainder until it is).

So far we have been able to use two’s complement representations for arithmetic operations. Multiplication worked, but it turned out more efficient to make the operands positive before multiplying, losing the benefits of two’s complement that we gained for addition. Unfortunately two’s complement is completely unsuitable for division.

When adding or multiplying fixed-size two’s complement negative numbers any large power of 2 that is generated is shifted away to the left and disappears. Addition preserves the magnitude of the extra power of 2, and multiplication amplifies it. In both cases the representation works because we can ‘throw away’ powers of 2 that are beyond the limits of the number representation. With division this doesn’t work.

### 3.9.1 Division of signed numbers

Calculating  $a \div b$  seems pretty straightforward when  $a$  and  $b$  are both positive. But when  $a$  is negative its two’s complement representation is  $(2^n - |a|)$ , and when we divide this by a positive number  $b$  we get  $2^n \div b - (|a| \div b)$ . The additional contribution,  $2^n \div b$ , has an arbitrary number as its quotient and an equally unpredictable remainder.

Getting back to  $-|a| \div b$  from  $(2^n - |a|) \div b$  is far more difficult than simply noting the signs of  $a$  and  $b$ , comparing them, etc, as we did for multiplication. So all division calculations must be performed on unsigned binary numbers.

### 3.9.2 The division algorithm

The algorithm used for binary division is exactly the same as the long division of decimals, which should be familiar to you from primary school:

$$\begin{array}{r}
 \text{STAGE 1:} \quad \begin{array}{r} 1 \\ 25 \overline{) 365} \\ -25 \\ \hline 11 \end{array} \\
 \text{STAGE 2:} \quad \begin{array}{r} 1 \\ 25 \overline{) 365} \\ -25 \\ \hline 115 \end{array} \\
 \text{STAGE 3:} \quad \begin{array}{r} 14 \\ 25 \overline{) 365} \\ -25 \\ \hline 115 \\ -100 \\ \hline 15 \end{array}
 \end{array}$$

Once we reach the end of Stage 3 we have a remainder (15) which is smaller than the divisor, so we stop dividing because we have the result:  $365 \div 25 = 14 \text{ rem } 15$

**Binary division** is simpler than decimal. Long division of  $a$  by  $b$  includes a step where you must work out how many times  $b$  “goes into” (some part of)  $a$ , but in the case of binary,  $b$  either “goes into”  $a$  once (because  $b \leq a$ ), or not at all (because  $b > a$ ). The result is 1 or 0 respectively, and that result is shifted into the quotient in the correct position. Here is an example, in which the dividend is eleven (1011) and the divisor is three (11):

$$\begin{array}{r}
 \text{STAGE 1:} \quad \begin{array}{r} 1 \\ 11 \overline{) 1011} \\ -11 \\ \hline 10 \end{array} \\
 \text{STAGE 2:} \quad \begin{array}{r} 1 \\ 11 \overline{) 1011} \\ -11 \\ \hline 101 \end{array} \\
 \text{STAGE 3:} \quad \begin{array}{r} 11 \\ 11 \overline{) 1011} \\ -11 \\ \hline 101 \\ -11 \\ \hline 10 \end{array}
 \end{array}$$

**At Stage 1** the divisor (11) is aligned with the furthest-left column of the dividend (1011) that makes the subtraction possible (while ensuring that the result is not negative). 11 cannot be subtracted from 1 (the 1-bit number starting at the left-most end of the dividend), or from 10 (the left-most 2-bit number), but it *can* be subtracted from 101 (the left-most 3-bit number). So it is aligned with the 3-bit number 101, and the subtraction is performed:  $101 - 11 = 10$ . In binary this is the same as  $101 \div 11 = 1 \text{ rem } 10$ .

The alignment is for efficiency only. We could in fact start off by aligning the left-most bit of the divisor with the left-most bit of the dividend, then shift the divisor to the right to make the subtraction possible. This shift generates a leading zero for the quotient and we start subtracting at our present starting position. We build the quotient *left-to-right*, and it is not important how many leading zeros we place on the left-hand end because they do not change the quotient’s value.

As long as we start off with a dividend that is greater than or equal to the divisor a starting position will be found after zero or one shift. If the divisor is greater than the dividend we will not be able to shift it to the right without a digit ‘falling off’ its right-hand end. In this case the dividend becomes the remainder, and the quotient is 0. In our example, at Stage 1 we found the initial position and did the subtraction, placing a 1 in the quotient area and 10 as the current remainder.

**At Stage 2** we expand the remainder by bringing down the next digit to the right in the dividend and appending it to the right-hand end of the current remainder. If the expanded remainder is greater than or equal to the divisor we move on to Stage 3. Otherwise we must place a 0 in the quotient and bring down the next digit from the dividend to expand the remainder further.

We keep this process going until we either end up with a remainder from which the divisor can successfully be subtracted (in which case we move on to Stage 3) or else there are no more digits to bring down from the dividend (in which case we have finished the division and the current quotient and remainder are the end results).

**At Stage 3** we do exactly the same as we did at Stage 1, except now the remainder cannot be expanded and so this is the last stage. We find, as we should, that  $11 \div 3 = 3 \text{ rem } 2$ .

In a bigger example, Stage 1 and Stage 2 will need to be applied one after another many times in exactly the same manner. Of course in a computing platform (as opposed to a human exercise on a piece of paper), the divisor will be represented in a fixed-length bit-string, as will the dividend.

Subtraction of the divisor presents no problem as it involves working out its two’s complement and doing addition. A further optimisation may be applied by observing that during division we always subtract the

same number from the changing current remainder, so the divisor can be subjected to the two's complement change of sign once and the result added every time the algorithm calls for subtraction.

However it is implemented, division of binary numbers is a complex and lengthy procedure. The platform that we will study in this text has no built-in facilities for multiplication, let alone division; both operations can only be supported by writing programs that make use of addition, subtraction and other simple operations. This is typical for small systems, such as microcontrollers operating in fridges or washing machines, and for other systems where division and multiplication will be performed very rarely, so the cost of adding circuitry to perform those tasks outweighs the benefit of providing the operations.

## 3.10 Sets

Having dealt with whole numbers we will make a short pit stop in the corner occupied by sets. You should be familiar with elementary set theory from either secondary school or an appropriate undergraduate module. Here is a reminder, which we include for ease of reference.

### Definition 8 : Sets and Universes.

- A **set** is an unordered collection of items, no two of which are the same. In mathematics, sets may be finite or infinite. In computing we can only deal with finite sets.
- An **element**,  $e$ , of a set,  $S$ , is an item that is in the collection. The element  $e$  is said to be a *member* of the set  $S$ . Mathematically this is written  $e \in S$ .
- A **subset**,  $s$ , of a set,  $S$ , is a (possibly smaller) set whose elements are members of  $S$ . Mathematically this is written  $s \subseteq S$ .
- A **universe** (or *universal set*) is the set of all items of some type. For example, the set of all possible natural numbers. In mathematics universes may be finite or infinite. In computing we can only deal with finite universes, so whilst we might represent a universe of natural numbers, it will always be a finite subset of the mathematical universe of natural numbers.

We will only be considering sets as *finite* collections of elements which are subsets of some given *finite* universe  $U$ . Members of such a universe are *countable*. In other words there exists a method of assigning consecutive natural numbers to members, starting from 0 and ending up with the number that corresponds to the size  $|U|$  of the universe less 1:

$$U = \{m_0, m_1, \dots, m_{|U|-1}\}$$

The manner in which we will represent any set takes nothing from the universe of values other than the number of members it has. This means that we can represent any subset of any finite universe, as long as we have sufficient memory to do so.

### 3.10.1 Set representation

To make sets data objects that can be manipulated by a platform, we must be able to represent them as bit-strings. The number of bits in the string corresponds to the number of items in the universe of which our set is a subset. For example, if we are dealing with a universe of 93 items we need a 93-bit string to represent sets that are subsets of that universe.

So we choose to represent a set  $S \subseteq U$  as a size- $|U|$  bit string:

$$b_0, b_1, \dots, b_{|U|-1}$$

where  $b_i = 1$  if  $m_i \in S$  and 0 otherwise.

For example, if the universe  $U = \{\text{homer}, \text{marge}, \text{bart}, \text{lisa}\}$  then each subset of this universe may be represented by a 4-bit string. We may choose any enumeration of the universe<sup>24</sup>, but a specific one needs to be chosen.

We could, for instance assign position 0 to marge, 1 to lisa, 2 to bart and 3 to homer (numbering from left to right). Then the set  $\{\text{homer}, \text{lisa}\}$  would be represented using the bit-string 0101,  $\{\text{bart}\}$  using 0010 and the empty set  $\emptyset$  using 0000. It is important to understand that neither the bit-strings nor the members of the universe are numbers.

All bit-strings that represent subsets of a particular universe must be of the same length (in this case 4), “leading zeros” cannot be omitted, and arithmetic operations such as addition or subtraction are meaningless, giving results that are ambiguous and that depend as much on the enumeration we choose as they do on the characteristics of the sets we are trying to represent.

---

<sup>24</sup> The association of members of the universe with positions in the bit-string is called its *enumeration*.

### 3.10.2 Set operations

We know from elementary set theory that sets can be transformed and combined to produce a new set. Apart from *set complement*, which has only one operand, it is only meaningful to perform set operations on two subsets of the same universe, which will necessarily be represented as bit-strings of the same length. We define four basic set operations, each of which can be computed in our chosen representation:

**Set complement** For any set  $A \subseteq U$ , the complement,  $A'$  may be defined as follows

$$A' = \{x \in U \mid x \notin A\}.$$

That is, the complement of a set,  $A$ , is the set of all members of the universe that do *not* belong to  $A$ . If  $A$  is represented as the bit-string:

$$a_0, a_1, \dots, a_{|U|-1},$$

the set  $A'$  is represented as

$$\overline{a_0}, \overline{a_1}, \dots, \overline{a_{|U|-1}},$$

where the bar above a formula indicates logical negation, or *logical not*:  $\overline{1} = 0$  and  $\overline{0} = 1$ .

**Set union** For two sets  $A, B \subseteq U$ , the union,  $A \cup B$ , may be defined as follows

$$A \cup B = \{x \in U \mid x \in A \text{ or } x \in B\},$$

So the set  $A \cup B$  contains all members of  $U$  that belong to  $A$  or to  $B$  (including those that belong to both). If  $A$ ,  $B$  and  $A \cup B$  are represented by the three bit-strings:

$$a_0, a_1, \dots, a_{|U|-1} \quad b_0, b_1, \dots, b_{|U|-1} \quad c_0, c_1, \dots, c_{|U|-1},$$

then for all  $i = 0, 1, \dots, (|U| - 1)$ ,

$$c_i = a_i \vee b_i,$$

where the operation  $\vee$  is called *disjunction*, or *logical-or*:

$x$	$y$	$x \vee y$
0	0	0
0	1	1
1	0	1
1	1	1

Informally, to compute the union of two sets represented as bit-strings one needs to apply the logical-or *bitwise*. This means ‘or-ing’ together the individual bits of the two operands pair-wise.

The two strings are aligned, and pairs of operand bits in the same columns are ‘or-ed’ together, one column at a time, with the result being placed in the same column in the result string. This is known as a *bitwise logical-or* operation. It gives a bit-string of the same length as each of the operands, but with each position occupied by a ‘1’ if one or both of them had a ‘1’ in that position, and a ‘0’ otherwise.

**Set intersection** For two sets  $A, B \subseteq U$ , the intersection,  $A \cap B$ , may be defined as follows

$$A \cap B = \{x \in U \mid x \in A \text{ and } x \in B\},$$

i.e. the intersection contains all members of  $U$  that belong to both  $A$  and  $B$  at the same time. If  $A$ ,  $B$  and  $A \cap B$  are represented as above, then for all  $i = 0, 1, \dots, (|U| - 1)$ ,

$$c_i = a_i \wedge b_i,$$

where the operation  $\wedge$  is called *conjunction*, or *logical-and*:

$x$	$y$	$x \wedge y$
0	0	0
0	1	0
1	0	0
1	1	1

The logical-and is applied bitwise, in the same manner as the logical-or above.

**Symmetric difference** For two sets  $A, B \subseteq U$ , the symmetric difference,  $A \Delta B$  may be defined as follows

$$A \Delta B = \{x \in U \mid \text{either } x \in A \text{ or } x \in B\},$$

i.e. the symmetric difference contains all members of  $U$  that belong either to  $A$  or to  $B$  but not to both. If  $A, B$  and  $A \Delta B$  are represented as above, then for all  $i = 0, 1, \dots (|U| - 1)$ ,

$$c_i = a_i \oplus b_i,$$

where the operation  $\oplus$  is called *exclusive-or* (or, sometimes, *modulo 2 addition*<sup>25</sup>):

$x$	$y$	$x \oplus y$
0	0	0
0	1	1
1	0	1
1	1	0

The exclusive-or is applied bitwise, in the same manner as the logical-or and the logical-and above.

**Membership test** We may wish to determine whether a particular element is a member of a set ( $e \in S$ ).

We do this by *creating* a set that contains only that element, then finding the intersection between this and the set we are testing. If the intersection is empty the element is not a member of the set. The empty set is easy to detect, because it is represented as a bit-string with all zeros, which is also the representation of the number 0. So the presence of any element  $e$  in a set  $S$  can be determined by finding  $\{e\} \cap S$  and checking whether the result is the empty set.

**Set cardinality** Determining the cardinality of a finite set is a matter of counting the number of members it has. In our representation this is the number of ones in the bit-string that represents the set.

Let us look at an example of a set expression. We have three sets,  $P, Q$  and  $R$ , each of which is a subset of a three-member universe, where  $P \mapsto 010$ ,  $Q \mapsto 100$  and  $R \mapsto 001$  (the sign  $\mapsto$  should be read “is represented as”). the result of the set expression

$$(P' \Delta Q) \cap (Q \cup R)'$$

can be worked out as follows:

$$\begin{aligned} P' &\mapsto 101 \\ P' \Delta Q &\mapsto 001 \\ Q \cup R &\mapsto 101 \\ (Q \cup R)' &\mapsto 010 \\ (P' \Delta Q) \cap (Q \cup R)' &\mapsto 000 \end{aligned}$$

---

<sup>25</sup> The term *modulo 2 addition* is sometimes used in recognition of the fact that the sum of two one-digit numbers modulo 2 has the same representation as the result of the exclusive-or of two logical values.

### 3.10.3 Bit-slicing sets

Sets are even easier to bit-slice than numbers. If the platform can only process bit-string data in “chunks” of some fixed size then this may be achieved as follows:

1. Use a collection of the same number of memory chunks to represent each set. The total number of bits in a collection of chunks must be at least as many as the number of elements in the universe.
  - Each element is allocated a bit-position in one of the chunks.
  - If the universe contains fewer elements than the available number of bit positions set the ‘spares’ to be always zero, because they will never be used. Setting them to zero guarantees that the representation of the empty set in any universe results in an all-zero bit-string, which makes the membership test efficient.

Note that it doesn’t matter which bit positions are considered ‘spare’, as long as they are the same for every set: the choice will not affect how the set operations are implemented.

2. In order to apply an operation to whole sets made up of several chunks we apply that same operation to corresponding chunks of the operands in turn. The result is the bit-string formed by the corresponding result chunks. This works because there is no set operation in which the value in one bit position is affected by the value in another, so each bit position (and therefore each chunk) can be considered independently of all the others.
3. Care should be exercised in computing set complement when ‘spare’ bits are introduced. Although the spare bits do not matter for the rest of the set operations, they will be flipped by set complement and will then become nonzero. As a result, the membership test will not work on such a representation, since a chunk that has a spare bit will not contain all zeros even if the rest of the bits are zero. Consequently, the set complement operation must additionally nullify any spare bits.

## \*3.11 Sparse sets

The representation discussed in the previous section is unquestionably effective. Whether it is efficient depends on the circumstances. If a universe is large but the sets we are going to work with will all have a tiny proportion of members, the bit string required to represent each of them will consist of mostly zeros. The time required to apply an operation to two such sets will be just as large as if each of them contained many members, since every member of the universe (as opposed to the members of the sets that are involved in the computation) will have to be visited.

For example, if a platform has primary facilities for manipulating 32-bit numbers (which any physical computer these days will have of necessity), it is capable of enumerating a universe with more than 4 billion members. Each set that is a subset of this universe will be represented by a bit string that is over 4 billion bits in length. This is pretty excessive in any case, but even more so if we know that programs will only ever have to deal with sets containing a few hundred (or even a few thousand) members. We may end up with bit-strings that contain four billion zeros and a hundred or so ones.

### Definition 9 : Sparse set

In computing, sets that are known to be much smaller than the universe,  $U$ , from which they are drawn – and will therefore contain vastly more zeros than ones in a simple  $|U|$ -bit representation – are called **sparse sets**.

#### 3.11.1 Representing sparse sets

For sparse sets a *list representation* is more efficient than a single bit-string. In the simplest case a set is represented as a list of its members under the assumed enumeration:  $X \mapsto [x_0, x_1, \dots, x_{|X|-1}]$ . So what we have is a list of numbers, each of which *identifies* one item that is a member of the set, instead of a bit-string where the presence of a 1 in a particular position indicates that a specific item is a member of the set.

When we use a list representation the count of unique identifiers in the list is the same as the count of members in the set. However, the size of the universe also influences the size of the bit-string representing the list. Each member is represented by its identifying number, and those numbers have to be in the list.

We require a bit string of length 10 to provide sufficient unique identifying numbers for a universe that has 1024 members ( $2^{10} = 1024$ ). For a universe of about a million members, we require 20 bits to represent each identifying number,etc. However, we only need to add a single bit to each number in a list in order to be able to handle a universe with twice as many members, as opposed to the need to double the number of bits in the dense representation discussed in the previous section.

At this point you may be wondering whether it is appropriate to represent an object (such as a sparse set) as a list at all, given that we have stated quite categorically that the only kind of representation a computing platform can deal with is a bit-string. The reason we can use lists of identifying numbers is that such a list may be trivially represented as a single bit string. Each number is represented as a bit-string, and we can join together these bit strings in an end-to-end fashion.<sup>26</sup> Thus a list may be thought of as a single continuous bit-string.

We need to know two things: the length of each bit-string that represents an identifying number, and how many numbers there are in the list:

- The universe from which members are drawn is of a known size, the length of each identifying number is governed by the size of the universe, and each identifying number is represented by a bit-string of the same length. This takes care of the first.
- The second is most easily achieved by finding a simple way to mark the end of the list, making it possible to count how many numbers are in the list. One of the simplest is to adopt the convention that the number zero is not used as the ID of any member of the universe. Then all we have to do is

---

<sup>26</sup> Joining strings together end-to-end to create a new, longer, string like this is known as *concatenation*.

place an all-zeros bit-string of the same length as a member ID at the end of each list. So every list (even a list representing the empty set) will have the number zero as its last element.<sup>27</sup>

Here is an example. Assume that the universe,  $U$ , consists of compass directions:

$$U = \{N, NE, E, SE, S, SW, W, NW\}.$$

this gives us 8 members of the universe. We must enumerate them from 1, because we are going to use 0 as an end-of-list marker, so we need sufficient bits to capture 9 different bit-strings. Thus we require 4 bits to accommodate the identifying numbers of the members:

N	1	0001	NE	2	0010
E	3	0011	SE	4	0100
S	5	0101	SW	6	0110
W	7	0111	NW	8	1000

Here we show the identifiers in both decimal and binary in order to use them in further representations. Under the above enumeration, the set  $\{W, NW\}$  may be represented as the decimal number list [7, 4], which would be encoded as the bit-string 01111000, and after appending the ‘end-of-list’ marker this becomes 011110000000.

In order to aid humans in reading these representations we will add spaces between the 4-bit strings that represent individual member numbers, even though they will not be there in a machine-level representation. So the set  $\{W, NW\}$  may be represented by the bit-string 0111 1000 0000; the set  $\{S, N, W, E\}$  may be represented by 0101 0001 0111 0011 0000,etc.

Sets are unordered collections, so the order in which identifiers appear in the bit-string is not important (nor was the order in which they occurred in the set written in extension). So  $\{W, NW\}$  could be represented as 1000 0111 0000 without a problem.

Moreover, it is possible (but less efficient) to use a representation of sets in which members can be listed more than once – as long as it remains the case that numbers representing *all* members of the set are included, and *only* numbers representing members of the set occur in the string that represents it. The only exception is that the number zero *must* occur once and *only* once, because that marks the end of the list.

Consequently we may distinguish 3 types of list-based representation of sets:

**Representation R1** Members need not be in order, repetition allowed

**Representation R2** Members need not be in order, repetition not allowed

**Representation R3** Members listed in a specific order, repetition not allowed

Note that the condition for a list to be considered of type R1 does not say that it *must* contain duplicates, or that the members *must not* be in order. So any list that satisfies the condition for R2 also satisfies the condition for R1. Similarly, any list that satisfies the condition for R3 also satisfies the condition for R2, and therefore satisfies the condition for R1.

---

<sup>27</sup>The termination requirement may seem artificial, since a list appears to be well defined by itself when it is written down, without needing a special termination marker. In reality, there *is* a special marker at the end of a list, at the very least the lack of a comma, which requires encoding just as every other sign or symbol used as data.

### 3.11.2 Efficiency of list representations

We have established that there are (at least) three different strategies we can adopt for representing sparse sets as lists. But which should we choose?

**Representation R3** is the most efficient one: each member is included only once, and we would normally ensure that members occur in the list in the ascending order of their identifying numbers.<sup>28</sup>

If we choose R3 each set has one - and only one - representation. This makes comparing and combining sets much more straightforward in many cases.

In representation R3 (and R2) the length of the list of numbers is guaranteed to be  $n + 1$  (where  $n$  is the number of members in the set), and the length of the bit string representing the set will be  $(n+1) \times \log_2(N+1)$ , where  $N$  is the number of members in the universe.<sup>29</sup>

**Representation R2** uses the same length of list - and hence the same length of bit-string - as R3. However, the list (and hence the string) is not unique, because there is no restriction on the order in which members appear.

As a matter of fact, there may be very many different lists that represent the same set in this way. This can make representation R2 inefficient. Suppose, for example, we need to compare two lists to find out whether or not they represent the same set.

**Representation R1** is neither unique nor size-bounded. It is consequently the least efficient of the three.

### 3.11.3 Operations on sparse sets

What can we do with a sparse set represented by a list? The answer is: all the standard set operations except complement, since the complement of a sparse set is bound to be a dense set: it will be almost the size of the universe (for a large universe).

**Union** To compute  $A \cup B$  using lists that satisfy the R1 condition, all that is required is to concatenate the two bit-strings (obviously after removing the terminating zeros from one). If we do this we are guaranteed that we will end up with a list representation of the set  $A \cup B$  that satisfies the condition for R1.

To do the same and end up with a result that is in R2 requires a scan of the list, number by number, in order to find and remove possible repeated members.

If we want the result to be in R3, the result also needs to be sorted so that the numbers representing members appear in ascending order. Fortunately this can be avoided by working exclusively with R3 representations, because the lists we are combining are already sorted into order.

In R3 we can scan through the lists representing  $A$  and  $B$  side-by-side, and work through them looking for matches by repeatedly comparing the first member of list  $A$  with the first member of list  $B$ , making sure we include the lower of the two before moving on.

A procedure for doing this is described algorithmically on the next page.

---

<sup>28</sup> purely for certainty; any fixed ordering would do as long as it is known in advance.

<sup>29</sup> Where a universe is of a size that is anything other than a power of two minus one,  $2^k - 1$  for some  $k$ , we round up so that there are sufficient bit-strings available to represent each member with a distinct bit pattern.

```

*** Union of A and B in R3 ***
begin
    set Pa to point at the head of list A
    set Pb to point at the head of list B
    repeat while Pa or Pb is pointing at a number that identifies a set member
    <<<<<
        compare the member pointed to by Pa with the member pointed to by Pb
        if the member of A < the member of B
            insert a copy of the member of A into the result list
            make Pa point to the next member of A
        if the member of A > the member of B
            insert a copy of the member of B into the result list
            make Pb point to the next member of B
        if the member of A = the member of B
            insert a copy of that member into the result list
            make Pa point to the next member of A
            make Pb point to the next member of B
    >>>>
*** at this stage one or both of the two lists has been examined in full ***
if only one list has been examined in full
    add the remaining members of the other list onto the end of the result list
*** now both lists have been 'used up' ***
add an 'end-of-list' marker to the end of the result list
end

```

**Intersection** To compute  $A \cap B$  in R1, for each member of  $A$  it is necessary to check whether it also occurs in  $B$ . Only then will it be included in the result.

The check is no more efficient if we say the list representing  $B$  cannot contain duplicates (i.e. if  $B$  meets the conditions for R2 as well as for R1), but it will generally be much quicker if  $B$  meets the conditions for R3 (no duplicates **and** members sorted into order).

If both  $A$  and  $B$  are in R3, we can once again obtain a fast solution by merging two sorted lists, except that this time we only include those members that appear in both lists.

```

*** intersection of A and B in R3 ***
begin
    set Pa to point at the head of A
    set Pb to point at the head of B
    repeat while Pa or Pb is pointing at a number that identifies a set member
        compare the member of A at position Pa with the member of B at position Pb
        if the member of A < the member of B
            make Pa point to the next member of A
        if the member of A > the member of B
            make Pb point to the next member of B
        if the member of A = the member of B
            insert a copy of that member into the result list
            make Pa point to the next member of A
            make Pb point to the next member of B
    *** at this stage one or both of the two lists has been examined in full ***
    add an 'end-of-list' marker to the end of the result list
end

```

**Set subtraction** Since the complement operation is not available in a list representation, we cannot implement  $A \setminus B$  as  $A \cap B'$ . Consequently, a distinct set-subtraction operation is required. This has similarities with intersection. Everything said about intersection applies to set subtraction, except that all members of the result list are drawn from  $A$ , and it only includes those that are *not* also in  $B$ .

```

*** set difference A \ B in R3 ***
begin
    set Pa to point at the head of A
    set Pb to point at the head of B
    repeat while Pa or Pb is pointing at a number that identifies a set member
        compare the member of A at position Pa with the member of B at position Pb
        if the member of A < the member of B
            insert a copy of that member into the result list
            make Pa point to the next member of A
        if the member of A > the member of B
            make Pb point to the next member of B
        if the member of A = the member of B
            make Pa point to the next member of A
            make Pb point to the next member of B

    *** at this stage one or both of the two lists has been examined in full ***
    if any members of A have not been examined
        add the remaining members of A onto the end of the result list
    add an 'end-of-list' marker to the end of the result list
end

```

**Symmetric difference** This operation has a similar efficiency to intersection in each of the three representations. There is no difference in search, the difference is in what we must do if we find a member in both lists: for intersection it is members that appear in both lists that go through to the result, for symmetric difference the only members that go through to the result are those that appear in one of the lists but not in both .

```

*** symmetric difference A / \ B in R3 ***
begin
    set Pa to point at the head of A
    set Pb to point at the head of B
    repeat while Pa or Pb is pointing at a number that identifies a set member
        compare the member of A at position Pa with the member of B at position Pb
        if the member of A < the member of B
            insert a copy of that member into the result list
            make Pa point to the next member of A
        if the member of A > the member of B
            insert a copy of that member into the result list
            make Pb point to the next member of B
        if the member of A = the member of B
            make Pa point to the next member of A
            make Pb point to the next member of B

    *** at this stage one or both of the two lists has been examined in full ***
    if only one list has been examined in full
        add the remaining members of the other list onto the end of the result list
    add an 'end-of-list' marker to the end of the result list
end

```

**Equality** Under representations R1 and R2 there are many different ways to represent a single set. Comparing two bit-strings to determine whether or not they represent the same set becomes much more complex than it was when we were dealing with dense sets, where whatever enumeration we chose for a universe each unique set had its own unique representation. Nevertheless, it is possible to compute the set difference  $A \setminus B$  and compare the result with the empty set, whose representation is always unique no matter what method we use. However, we can test the equality of two sets represented using R3 by literally comparing their bit-strings by the method we use to test the equality of two bit-strings representing dense sets.

**Example:**

We continue with our chosen universe of compass directions. Let us compute the union of two sets

$$\{W, NW\} \cup \{S, N, W, E\}.$$

Let us first try working with bit-string representations of R1 lists. Recall that  $\{W, NW\}$  may be represented as 0111 1000 0000 and  $\{S, N, W, E\}$  as 0101 0001 0111 0011 0000. We can get a correct result by concatenating the two R1 bit-strings, giving

0111 1000 0101 0001 0111 0011 0000

or

0101 0001 0111 0011 0111 1000 0000

Notice that member 0111 appears twice in each of these two representations of the set, and the numbers follow in no particular order. The two operands are good enough to use in R2 as well (they have no repeated members), but for the result to also be in R2 we must scan it again and remove the duplicate, giving

0111 1000 0101 0001 0011 0000

or

1000 0101 0001 0111 0011 0000

or

0101 0001 0111 0011 1000 0000

or

0101 0001 0011 0111 1000 0000

Now let us do it again in R3:

It turns out that representing  $\{W, NW\}$  as 0111 1000 0000 satisfies the condition for R3, since the members are listed in ascending order and there are no duplicates. But our previous representation of  $\{S, N, W, E\}$  does not satisfy the R3 condition, so we need to use a different bit-string: 0001 0011 0101 0111 0000, in which the two members that were out of place are moved to create a list that is ordered. Now we can merge the lists:

A	0111	1000	0000	
B	0001	0011	0101	0111 0000
result				

The initial step is to compare the first member in one list with the first member in the other. In this case the number of the first member in list  $B$  is lower than the one in  $A$ . Since both lists have been sorted into ascending order, that number is also lower than any number in the rest of either of the two lists, so it comes through as the first member in the result list:

A	0111	1000	0000	
B	0001	0011	0101	0111 0000
result	0001			

Now we take the first un-tested member of  $B$  again, as it is still represented by a lower number than the

first member of  $A$ :

$A$	0111	1000	0000		
$B$	0001	0011	0101	0111	0000
result	0001	0011			

and again:

$A$	0111	1000	0000		
$B$	0001	0011	0101	0111	0000
result	0001	0011	0101		

Now the members of both lists are the same, so a copy is inserted into the result, and both lists are moved on by one:

$A$	0111	1000	0000		
$B$	0001	0011	0101	0111	0000
result	0001	0011	0101	0111	

Now we reach the ‘end-of-list’ marker for  $B$ , so the remaining members of the result list come from  $A$ , giving:

$A$	0111	1000	0000		
$B$	0001	0011	0101	0111	0000
result	0001	0011	0101	0111	1000

Finally we add an ‘end-of-list’ marker to the end of the result list:

$A$	0111	1000	0000		
$B$	0001	0011	0101	0111	0000
result	0001	0011	0101	0111	1000

Notice that at every step we perform one comparison (we compare the next unused member of one list with the next unused member of the other) after which we perform one insertion into the result list, and move on to the next member in one or both of the lists. Altogether the number of operations required may be as few as 3 or 4 times the number of items in the longest list. That is very fast indeed. In R1 we would simply concatenate the bit-strings after removing the terminating zeros from the first one. Which would be even faster but would potentially increase the number of duplicates in the result compared with the operands. In R2 we could also concatenate the strings but then we would have to remove potential duplicates which may take many scans of the string and is consequently quite expensive.

As another example let us compute the R3 representation of the intersection of the same two sets. We start with sets  $A$  and  $B$  represented in R3 form:

$A$	0111	1000	0000		
$B$	0001	0011	0101	0111	0000
result					

The intersection of two sets contains only those elements that are in *both* of them. Using an R3 representation makes finding the list that represents this a straightforward proposition. Observe:

Align the members at heads of the two lists

Compare the current member of  $A$  with the current member of  $B$

$A$	0111	1000	0000		
$B$	0001	0101	0111	0000	
result					

The current member of  $B$  is represented by a lower number, so ignore it, and move the comparison point on to the next member of  $B$ ,

then compare the current member of  $A$  with the current member of  $B$

$A$	0111	1000	0000	
$B$	0011	0101	0111	0000
result				

The current member of  $B$  is represented by a lower number, so ignore it, and move the comparison point on to the next member of  $B$ ,

then compare the current number in  $A$  with the current number in  $B$

$A$	0111	1000	0000	
$B$	0011	0101	0111	0000
result				

The two are the same, so insert a copy into the result list, and move the comparison point on by one member the the next member of both  $A$  and  $B$

then compare the current member of  $A$  with the current member of  $B$

$A$	0111	1000	0000	
$B$	0011	0101	0111	0000
result	0111			

List  $B$  is now empty, so ignore the rest of list  $A$

$A$	0111	1000	0000	
$B$	0011	0101	0111	0000
result	0111			

Now add the ‘end-of-list’ marker to the result list

$A$	0111	1000	0000	
$B$	0011	0101	0111	0000
result	0111	0000		

Again, we observe that merging lists in R3 gives us the amount of work proportional to the (now shortest rather than longest) list length. If we were using the representation R1 we would have to traverse the second list every time we consider a member of the first as a potential member of the intersection. That requires tens of times more work on sets sized in the hundreds.

## 3.12 Representing text

E.W.Dijkstra<sup>30</sup> is credited with the maxim: “The easiest machine applications are the technical/scientific computations.” In other words, the simplest thing one can do with a computing platform is to use it for crunching numbers. Whether or not this is true in general, it is certainly true that from the point of view of IT and computer science, arithmetic is hardly a challenging topic. Much more interesting applications arise when data of a different nature is manipulated by a computing platform. Of those other forms of data, text is probably the most important.

### Definition 10 : Text

**Text** is an arrangement of characters that conveys information. The characters are drawn from a finite (though potentially very large) collection, which for a Western text typically contains letters among other characters, hence the term *text*. Each character has an **identity** independent of the details of its graphical representation, i.e. what dots/lines are used and its colour/intensity. For example, we recognise what a lower-case **t** character and a comma character stand for no matter whether they are printed or hand-written, in a large or small size, whether they are italicised or bold, and no matter what colour they are. We recognise that one signifies a particular letter of the alphabet with a particular pronunciation, and the other signifies a punctuation mark that has a particular interpretation. In other words each character is an example of a symbol that “means” something. One could say that the character is an *abstract idea* of a certain smallest unit of text, rather than any particular representation of it, of which there can be more than one anyway. The collection of characters used in the text is an important *text attribute* as characters need to be represented and identified for correct interpretation of text.

We also derive information from other text attributes, of which we identify the following three that are important for computing platforms:

**The order in which characters appear in the text.** An ordered sequence of characters (a character string) can represent a word, a sentence, a mathematical formula, or some other information-carrying chunk. In fact an entire text document may be represented as a character string.

**The formatting that's applied to characters and character strings.** The choice of layout, such as the use of spaces and tabs, page and line breaks, line and paragraph indents, and the choice of character attributes such as typeface, point size, and index position (sub-/superscript), all influence what information a reader derives from a text.

**The structure of a text document.** Documents may be organised into chapters, sections, subsections, etc. They may contain numbered lists, bullet points, tables and other structures. This structuring is part of the information content of the document.

It is worth noting these four attributes are not independent of one another. Structure and content are intimately related to one another, and the interpretation of each is influenced by formatting and the character set being used.

Text has a long history and we do not have the space here to review even a significant part of it. The period of that history relevant to our needs starts with the invention of electrically controlled typing machines, which could be operated at a considerable distance over electric wires. Those machines, known as teletypewriters, or *teletypes*, were invented for transmitting and receiving *messages*. A message is a piece of text represented by an ordered sequence of “characters”, which is to be transmitted by one machine and later received by another, so it has both a spatial and a temporal existence (a message takes up both space and time).

Each teletype had a keyboard and a type head, ink ribbon and paper roll (for printing alphanumeric symbols), so it could be used as an electric typewriter. A teletype also had the facilities for sending information to distant teletypes about each key that the operator pressed, so that a distant machine could print out the same text as was being typed locally. The unit of transmission for a teletype was a single character, so messages were sent, one character at a time, along the wire from the transmitting to receiving teletypes.

---

<sup>30</sup> One of the founding fathers of computer science, see archive item EWD498

Since the only thing a teletype could send or receive was a character it was necessary for some of them to be interpreted by the receiver as invisible “control” inputs, rather than as letters, digits, or other marks that could be printed on paper. These control characters were used to indicate things like the start and end of a message, when to move the print head to the left-hand end of the line on the printout, when to move the paper up one line, and when to attract the operator’s attention by ringing a bell. Other control characters carried signals that changed the mode or manner of printing at the receiving end.

What teletypes did to text was to reduce the above four attributes of it to just two: identification of characters (including control ones) and their sequencing. Identification was done in the form of binary digital encoding (which preceded the era of computing by a century, appearing as it did in 1849 as technology for a communication line between New York and Philadelphia). Sequencing was based on the fact that the order in which character codes were sent from one end was the same as the order in which they arrived at the other.<sup>31</sup>

### 3.12.1 Character encoding

Fundamental to both the teletype and modern computing is the concept of text as a sequence of *encoded* characters, some of which are *printable* and others of which are used to *control* the output device. Fortunately each character can be encoded as a short bit-string.

At one time each different computer manufacturer was free to use whatever encoding was most convenient for their equipment, but this meant that text created for one manufacturer’s machines could not be used on another manufacturer’s machines without being run through a program that converted every character from one encoding to the other. So a receiving user needed to know what kind of machine had been used by the person who sent the text. This led to the adoption of standard representations of characters in the 1960s, with the ASCII<sup>32</sup> standard being defined in 1963.

The ASCII standard (which was actually only one of several encoding standards available at the time) was rapidly adopted by the majority of computer manufacturers. It encoded the reduced character set found on a ‘standard’ US English teletype keyboard of the 1960s using 7 bits per character. There were  $2^7$  different bit patterns, so 128 different characters could be encoded. The use of the ASCII encoding system became so widespread that it remains unchanged to this day, in order to maintain backwards-compatibility.

Later standards have been grafted on top of it by lengthening the bit-string and adding further representations, but the original 7-bit character encoding has formed the first 128 characters of each newer standard. Currently the Unicode standard - which can encode tens of thousands (and potentially millions) of different characters - includes the original ASCII character codes as its first 128 characters. The ASCII encoding system is presented in its original table form in figure 3.2.

Because each ASCII character is encoded as a 7-bit string there are exactly 128 potential characters, all of which are present in the table. The table is structured into 4 columns, with 32 characters in each. Since a bit-string can also be interpreted as a binary number, it is common to refer to ASCII codes as numbers, and to represent them in decimal or hexadecimal format.

---

<sup>31</sup> Modern digital communications do not rely on this principle. Characters are sent in numbered batches - called packets. Packets can be *sent* and *arrive* in any order because the whole message can still be obtained by sorting them into numerical order before concatenating them.

<sup>32</sup> ASCII = American Standard Code for Information Interchange

## ASCII CODE CHART

BITS				0 0 0 0 0 0 0	0 0 1 0 1 0 1	0 1 0 1 0 0 1	1 0 0 1 0 0 1	1 0 0 1 0 1 1	1 0 1 1 0 1 1	1 0 1 1 0 1 1		
b7	b6	b5	b4 b3 b2 b1	CONTROL			SYMBOLS NUMBERS		UPPER CASE		LOWER CASE	
0 0 0 0 0 0 0	0 0 1 0 1 0 1	0 0 1 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127 128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143 144 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159 160 161 162 163 164 165 166 167 168 169 170 171 172 173 174 175 176 177	NUL DLE SOH DC1 STX DC2 ETX DC3 EOT ENQ ACK SYN BEL ETB BS CAN HT EM LF SUB VT ESC FF FS CR GS SO RS SI US	16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127 128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143 144 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159 160 161 162 163 164 165 166 167 168 169 170 171 172 173 174 175 176 177	SP ! " # \$ % & ' ( ) : ; ,— . / ?	0 1 2 3 4 5 6 7 8 9 A B C D E F G H I J K L M N O P Q R S T U V W X Y Z ` [ ] ^ _ ` ~ DEL	64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127 128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143 144 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159 160 161 162 163 164 165 166 167 168 169 170 171 172 173 174 175 176 177	96 97 98 99 100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127 128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143 144 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159 160 161 162 163 164 165 166 167 168 169 170 171 172 173 174 175 176 177				

LEGEND:

dec	CHAR
hex	oct

Victor Eijkhout  
Dept. of Comp. Sci.  
University of Tennessee  
Knoxville TN 37996, USA

Figure 3.2: The original ASCII Code Chart (courtesy of Prof. Eijkhout, U.Tennessee)

### 3.12.2 Control characters

The **zeroth column** in the ASCII table, bit-strings 0b00000000 - 0b00011111 (hexadecimal 0x00 - 0x1F), contains exclusively control characters. Their meaning - with a few exceptions - is irrelevant to present-day technology, since it reflects the workings of a mechanical printing device. For instance, code 0b0000111 (0x07), character BEL, caused the teletype to ring the bell in order to attract the operator's attention, and code 0b0001011 (0x0B), character VT, would cause the teletype to feed paper to the next vertical tab stop on the rotating wheel. Still, some of those characters are in use to this day, and are present on computer keyboards:

**HT, Horizontal Tab** . The electric and mechanical typewriters of the time (and a teletype is essentially a typewriter with telecommunication capabilities) had a number of resting places where the printing head could be halted, called horizontal *tab stops*. The stops were placed at regular intervals, but their locations could also be adjusted. Pressing the tab key on a typewriter would send the print head moving rapidly to the right until it reached the next tab stop. This greatly facilitated the alignment of columns in tables. Computer keyboards still have a tab key, which generally causes a virtual print head (indicated by an on-screen cursor) to move to the right. However, the precise way a tab character is interpreted may differ from one document to another. Some text editing programs employ only fixed-width characters (like an old-style typewriter) and interpret a tab character in the same way as a teletype would have done, with tab stops placed every 8 character positions along the horizontal line. Other programs insert a fixed number of spaces every time a tab character is encountered. Word processors allow users to define tab stops using the *distance* from the beginning of the line (as opposed to the number of characters), because when variable width fonts and different point sizes are in use it is impossible to predict how many characters can be placed between consecutive tab stops. Still, it is convenient to use tabs for tabulation with fixed-width fonts, and they are used quite often in programming for marking structural indents, for example in a Python program.

**CR, Carriage Return** was used together with LF, Line Feed, to move the print head to the beginning of the next line in a piece of text. On the 1960s teletype the print head moved to the right each time a character was printed, until the print head reached the right-hand end of the carriage. Any characters printed after that point would simply be printed over the top of those that were already there. If a piece of text was to appear on more than one line it was necessary to send control signals to move the print head back to the left-hand end and down the paper.

The CR character caused the print head to fly back to the left-hand end of the carriage. On its own this wasn't enough to start a new line, because the print head also needed to be moved down the paper. The LF character would cause the teletype to feed a single line's worth of paper through so that the print head was moved one line down on the sheet without changing its horizontal position. The combined effect of CRLF (usually in this order) was to move the print head to the start of a new line. Of course it wasn't necessary to wait until the print head reached the far right end before sending the CRLF sequence, so a new line could be started at any point in a message.

Modern systems no longer work in quite the same way, though it is still necessary to signal the point at which a new line of text should be started. So whilst we no longer need control signals to move the print head we still use one or more characters to say "start a new line". This is referred to as a *newline* marker, and it is generally inserted into a piece of text when the 'Enter' key is pressed on a computer keyboard. For the sake of continuity and consistency it is normal in ASCII-encoded text to use one or both of CR and LF to mark the point at which a new line should begin; however, different newline markers may be employed on different platforms because there is no common standard. For example, Microsoft Windows platforms employ CRLF, Linux and Unix platforms use just the LF, and older versions of the Macintosh platform use just CR.

**ESC, Escape** is a special control character that is used in conjunction with other characters. The original meaning of it was as a character that causes an "escape" from printing, in order to use a sequence of otherwise printable characters for control purposes until a special termination character is sent. This was (and still is) known as an "escape sequence". Neither the ESC nor the other characters in the escape sequence would be visible to the user, though the effect of sending the escape sequence to the device most probably would.

The definition and use of escape sequences allowed manufacturers of displays and printers to add functions that were not available on a 1960s teletype without the need to add new control characters

to the ASCII encoding scheme. For example the iconic VT100 video display terminal (whose legacy is still seen in UNIX configuration files) employed escape sequences that started with ESC] and contained one or two ordinary characters, the combined effect of which was to move the cursor around the screen, to turn the screen on and off, to change the style of the printing characters that followed,etc. Similar escape sequences were defined for other displays and for output devices such as printers, and the technique is still in use today.

Of course an escape sequence could be generated by a user typing ESC followed by the appropriate printing characters, but it was frequently better to provide special keys on the keyboard (cursor keys, ‘function’ keys,etc), each of which caused an escape sequence to be generated and sent to a computer, a display, or a printer.<sup>33</sup> It was also possible to have these character sequences stored in data files or generated by programs.

### 3.12.3 Modifier keys

The Shift key and the Control key (CTRL) are *modifier* keys. Most keyboards are too small to accommodate a separate key for every different character in the ASCII set. This would mean having two keys for each letter of the alphabet because the lowercase and uppercase forms of a letter are encoded differently, a special key dedicated to each punctuation mark or other non-alphabetic symbol, and a special key for each (non-printing) control character. Not only is this impractical, but it would render keyboards much more difficult to learn and use. So there are a number of modifier keys on a computer keyboard, none of which generates a code of its own when pressed, but each of which modifies the codes generated by other keys.

All keyboard users are familiar with the Shift key. Each key that is assigned to a printing character generates one code when it is pressed on its own, and a different code when it is pressed with the Shift key held down. The Control key has a similar effect, except that the codes generated are usually for non-printing characters.

These two modifier keys were on 1960s teletype keyboards too. An uppercase letter, or a special symbol such as \*, could be generated by holding down the Shift key and pressing a key for one of the printing characters. A control character could be generated by holding down the Control key whilst pressing a key from the third column in the ASCII code chart. The combined effect of these keys was to generate the ASCII code that corresponds to the second column character minus 0x40. Thus CR could be generated by pressing Ctrl/M and HT by pressing Ctrl/I,etc.

Modern keyboards are still capable of this, though the way keyboard output is encoded and interpreted has changed over the years, and it is unusual to find a keyboard that generates ASCII codes, even though these codes may be in use within the computer.<sup>34</sup> However, nearly all ASCII characters can be made available via a modern keyboard, if not by using single keys, then by combining a modifier key with a character key.

### 3.12.4 8 bit character codes

Although each ASCII character may be encoded in 7 bits this is inconvenient from a technological point of view. A string of ASCII characters will be a multiple of 7 bits in length, so working up through the characters in a character string, or finding a specific character, requires multiplication by 7. In binary it is much more straightforward - and much faster - to multiply and divide by powers of 2 than by any other numbers, so it makes sense to use 8 bits per character rather than 7. Also, the standardisation of the byte at 8 bits means that the natural ‘chunk size’ for data transmission is 8, rather than 7, bits. So it has become commonplace to employ an 8-bit representation for characters, with the right-most 7 bits of each byte carrying the ASCII code. This means ‘wasting’ the left-most bit, but the benefit far outweighs the cost.<sup>35</sup>

<sup>33</sup> Whilst escape sequences are still in use today, there are now alternative techniques available for achieving the same effects.

<sup>34</sup> In the 1960s the keyboard was an integral part of the teletype, but now it is a separate device. Manufacturers can design keyboards that generate key codes that have nothing to do with ASCII, which are then converted to ASCII by a program (a *keyboard driver*) that runs on the platform. This allows them to manufacture a single type of keyboard that can be used with many different key layouts or alphabets by putting different labels on the keys, and providing different driver software (so that the same key codes are converted to different ASCII/Unicode characters). If you want to see this in action try using a British English keyboard with a French keyboard driver, and typing “the quick brown fox jumps over the lazy dog”.

<sup>35</sup> In actual fact the left-most bit need not be wasted. When it is 0 the left-most 7 bits are interpreted as ASCII codes, but when it is 1 the left-most 7 bits can be used to encode characters that are not in the ASCII code table. Unfortunately this has been done differently on different platforms, meaning that there is no single standardised definition of ‘the’ 8-bit *extended ASCII* character set. The Unicode system, which allows each character to be encoded in 1, 2, or more *bytes* introduced a standard interpretation of 8-bit, 16-bit, and longer character codes.

### 3.12.5 ASCII objects

**Sequence of lines.** Having dealt with single characters, we now turn to the issue of how characters are collected together to make a piece of text. A single character is taken from a fixed collection of characters, so it is representable by a bit-string of fixed size. However, a piece of text does not have a natural size. It could be 8 characters in the case of a login password, or over 5 million characters in the case of The Complete Works of William Shakespeare. Nevertheless, when character data is used in data processing (such as database queries or Google searches) the unit of information is usually the *line*, which is, of course, of variable length. You may wonder how one uncertainty (the length of a line) is better than another (the length of a complete piece of text, e.g. a book). The answer is that a line can be defined in a way that makes it easy to manipulate:

- The length of a line can reasonably be limited to something of the order of a few tens or a few hundreds of characters.
- A line can be restricted not to contain control codes other than what is known as white space (i.e. spaces and tabs).
- The end of a line can be clearly identified using a control character.

Even though the line continues to be of variable length, it is of limited variation and hence requires a limited amount of platform resources for processing.

We could then slightly restrict the representation of text and only consider such text objects that consist exclusively of a sequence of newline-separated lines. Most computer programs (in their source code form) are of this kind. A web page, even though it looks enriched with screen positioning information and all sorts of layout and font specifics, is defined using text in a sequence-of-lines representation. A line can contain special instructions in printable form to instruct the web browser how to format text, images and other content, but it is still a line of text in the above narrow sense.

**Null-terminated strings.** If we consider a character string as data for a computer program, we will need some indication of its size in order to be able to manipulate it. Since the size is not part of the definition of a character string (unlike numbers that are defined as, for example, 8-bit, 12-bit, etc.) what is necessary is a means of marking where the string begins and ends. The beginning is easy: we just point at the location in memory where the first character may be found. However, if we don't know the length of the string in advance it is difficult to know where it will end. What we do is to reserve a special control character for use as an 'end-of-character-string' marker, and make sure that every character string ends with that character. It could be any of the non-printing characters of the ASCII collection, except HT or SP (which are used to put white space into the string). The standard choice of terminator is NULL (which is represented as 0), because it is the easiest to detect: the typical computing engine includes facilities for recognising 0 already in its arithmetic circuitry. Assuming that we are employing an 8-bit character representation (see above), an  $n$  character NULL-terminated string (i.e. a string that consists of  $n$  printable characters followed by the 'end-of-character-string' marker, NULL) we get an  $((n+1) \times 8)$ -bit string (an  $n+1$  byte string), representing the whole character string.

**Operations on strings.** The concept of text *processing* presupposes a certain set of operations that are performed on strings of characters, just like arithmetic is based on operations performed on numbers. Unlike numbers, text is a much more complex substance which supports a large variety of operations. Most of them are not primary in the sense of representing an algorithm, which is defined using basic operations on NULL-terminated strings. It is the latter that we are interested to explore as they must be supported by low-level platforms if text processing is to have any chance higher up the Tannenbaum hierarchy. Here they are:

**empty(s)** This is a predicate that checks whether the first character of the string  $s$  is NULL, and consequently whether the string is empty. The result is a logical value: true or false.

**firstchar(s)** This operation takes a string  $s$  yields its first **character**  $c$  as the result. If the string is empty, the operation is invalid. This is similar to the overflow event we discussed when dealing with numbers.

**rest(s)** This operation yields a **string** that is identical to  $s$  with the first character removed. If  $s$  is empty, the operation is invalid.

**extend(s,L,c)** This operation lengthens string  $s$  of length  $L$  by 1 by appending the character  $c$  to it at the end<sup>36</sup>. If the length of the string  $s$  is not equal to  $L$  the operation is invalid.

The names we use to identify string operations are not necessarily standard (there is more than one standard set of notations in this area), but their effect is quite common anyway. What is interesting about the first three is that the cost (the amount of time, memory space, etc) of each of them is independent of the length of the string  $s$ . This is true of the last operation, too, except to ascertain its validity (if that cannot be done by controlling the operands) takes a time proportional to the length of the string  $s$ .

The most common (but not the only) implementation of character strings is based on byte memory, which will be discussed in section 3.13 later. Strings are placed in a contiguous range of memory cells, which are consecutively numbered byte-size containers. The cell numbers are referred to as memory addresses. A string can be identified merely by the address of the first cell it occupies. The cell at the beginning of the range contains the first character or NULL, and consequently its inspection is all that is required for the operations **empty** and **firstchar**. The operation **rest** simply adjusts the range to start at the second cell of the original range, and the operation **extend** overwrites the last cell in the range, which contains NULL, and which it locates by adding  $L$  to the address of the first cell, with the character  $c$ . Then it writes NULL to the next cell to finish the job. Finding the length of a string, if it cannot be obtained from other data, can be determined by a program that repeatedly applies **empty** and **rest** counting until **empty** yields true.

Naturally, all sorts of complex operations on text, such as search of a pattern in a string, rearrangement of delimited text segments, etc. can be programmed using the above operations. Due to their constant cost irrespective of the underlying representation, meaningful assumptions can be made as to the operation efficiency, which allows for important optimisations. Text processing of this kind is typically found at levels 4 and above; it is covered by function or class libraries of high-level programming languages and are outside the scope of this book. It is important, however, to see what assumptions are made about the cost of the basic operations to be able to design code at a high level of abstraction without unexpected losing performance.

### 3.12.6 Unicode

ASCII characters are conveniently sized in representation and logically organised in the code space; they are easy to work with. Importantly, most programming languages are based on the character set found on a conventional keyboard, which is covered almost completely (with some unimportant exceptions, such as the  $\pm$  character found on some Mac keyboards) by the code. The only problem is ... the *human* world is seriously multilingual. Even West-European languages rely on accented letters of the Roman alphabet, such as é, umlauts such as ü, etc, not found in ASCII, not to mention completely different alphabets such as Cyrillic. For some time “quick and dirty” solutions were used, based on extending the 7-bit ASCII to include the remaining 128 codes in a byte, from 0x80 to 0xFF, but this could only accommodate a small extended alphabet of a single language. Any attempt in the past at standardisation of at least the characters used by the European nations only, which jointly amount to no more than 10% of the world population, but which

---

<sup>36</sup>all string operations receive and produce NULL-terminated strings so the NULL will be preserved during extension

require Latin, Greek, Cyrillic, Armenian, Hebrew, Arabic, and Georgian characters in several versions for everyday use<sup>37</sup>, resulted in multiple encoding tables. A specific table would be used by a subset of the users and different subsets would not be able to communicate with each other without specifying which code table is used to represent which specific character groups.

Of course if we embrace the rest of the world, especially users of the so called CJK (Chinese, Japanese and Korean) writing systems, the code space immediately explodes. To aspire to any degree of completeness, a set of Chinese Han characters alone has to exceed 30,000 members! The problem was appreciated as far back as 1987 when the corporations Xerox and Apple collaborated to count characters in all world alphabets and to design a code space to include them all. The year 1991 saw the birth of a new code, called Unicode, which had a 16-bit code space where up to 64K international characters could be accommodated. Yet it was discovered later that even such a huge number of “letters” was not large enough to include scripts that are rarely used, characters that are needed from time to time to represent historic names, or toponymy, etc.

**Code points.** Fast forward to present day, and we find that the World Wide Web, which is the planet’s largest repository of text has adopted Unicode as the de-facto standard representation of text characters. As mentioned before, ASCII is still in use and is the principal character set for programming languages (except when character strings are used in programs as text constants, in which case there is no reason why they should be limited to just the ASCII). For the rest, the Unicode standard defines 1,114,112 so-called *code points*, i.e. integer numbers that correspond to individual characters. The code points are abstract because, by contrast to ASCII codes, they are not immediately usable as bit-strings, but instead require a *secondary* representation. It works as follows: the world’s characters are represented by code points, which are numbers in the range 0 to 1,114,111, and the code points are *encoded* in a certain way as bit-strings. In fact there are several standardised encodings each preferred for its own reason. The universally accepted one is called `utf-8`, which works best for documents where the majority of characters are ASCII or other European/non-CJK, and which account for more than 80% of the Web, but there is also `utf-16` used as the main encoding by Microsoft Windows, and which is better suited for CJK-dominated texts. There are other encodings as well with more esoteric properties.

The need for a secondary encoding is not, strictly speaking, limited to Unicode. We could treat ASCII similarly, interpreting its codes as code-points and declaring the 8-bit format (leading 0, 7-bit representation as a binary number) as an encoding. Or we could encode ASCII strings by packing each group of 8 characters into 7 bytes. Or any other method of achieving correspondence between a sequence of codes and a sequence of bytes. However, for a code that deals with only 128 characters, the benefits of alternative encodings would be marginal at best.

Not so for Unicode. From the information theory point of view, the information content of each Unicode character is 21 bits (based on  $\log_2 1114112$ ). That is three times the amount of information contained in an ASCII character! If you use Unicode for writing programs, where ASCII with its 7 bits per character is a largely sufficient character set, and if the encoding used for this purpose were *representation of the code-point as a 21-bit binary*, it does not take more than a moment to realise that about 2/3 of the storage space would be wasted. The 14 bits difference is the information content of the knowledge that the character is not outside the ASCII-subset of Unicode!

Code points have their own syntax, which is convenient for distinguishing characters and general binary numbers apart.

**Secondary encoding** Limiting ourselves to a specific secondary encoding known as `utf-8`, let us see how this problem can be alleviated by clever design (figure 3.3). As you can see, the `utf-8` encoding is an example of a variable-size representation. This reflects the stage-wise access to information, which is a fairly universal encoding technique. We will ourselves use it later, in designing the instruction set of our example Platform 2 (see section 8.2), so it is worth taking an early look early at how such things may work.

The idea is that a choice of 1 out of  $N$  only represents  $\log_2 N$  bits of information if all choices are equally probable (i.e. we do not have *any a priori* information about which member of the set may be chosen. Since we expect most practical choices to choose from a subset, in our case ASCII, let us introduce an *ancillary choice*, or selector, which tells us whether the character is an ASCII character or not. This is a choice of one out of two, and it only requires a single bit to make. Specifically, let us nominate bit 7 of the first byte of

---

<sup>37</sup>including social, business and religious practices

Number of bytes	Bits for code point	First code point	Last code point	Byte 1	Byte 2	Byte 3	Byte 4
1	7	U+0000	U+007F	0xxxxxx			
2	11	U+0080	U+07FF	110xxxx	10xxxxxx		
3	16	U+0800	U+FFFF	1110xxxx	10xxxxxx	10xxxxxx	
4	21	U+10000	U+10FFFF	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx

Figure 3.3: Unicode secondary encoding: utf-8

the encoding as such a selector, and if that bit is 0, then the remaining 7 bits are used to select one ASCII character out of 128.

So far so good, what do we do if the character is a non-ASCII code point of Unicode? Obviously, bit 7 of the first byte goes up to indicate a non-ASCII choice, at which point the minimum number of bytes taken for the character goes up to 2. Of the 15 bits allocated, the most significant bits of the first byte indicate how many bytes will be used to accommodate all the significant bits of the code point. Specifically, the prefix 110 indicates two bytes, 1110 three bytes, and 11110 four bytes. Notice that the prefixes are mutually distinct irrespective of the subsequent less significant digits in the byte. Indeed, they are simply a string of 1's terminated by a 0. The number of 1's before the first zero is the total number of bytes taken to represent the character. The subsequent, *continuation* bytes are marked with the prefix 10, which again is distinct from any prefix of the first byte as well as from all ASCII characters.

It is easy to see that in a string of Unicode characters encoded according to `utf-8` one can immediately identify each byte as either

- a solitary ASCII character
- the first byte of a non-ASCII character
- a continuation byte of a non-ASCII character

This is convenient for scanning Unicode text, which can be done with a small penalty from any byte in the middle of the text: one has to check whether the byte is the first or the only byte of a character, and if not, move to the next continuation byte until the first/only byte of the next character is encountered — no further than three bytes away. This is important because for an arbitrary multibyte encoding it might be necessary to scan the text from the beginning in order to determine where each character is located.

As we have seen the most distinctive feature of `utf-8` is its compact representation of ASCII: 8 bits despite the 21-bit information capacity of a Unicode character. From the information theory point of view, information cannot be gained at no cost in space, so we must ask ourselves what has been sacrificed in order to achieve the saving. If some characters take less bits in representation than they should, it is to be expected that some other characters will take more. Indeed they do. The four-byte characters take 32 bits to represent a 21 bit code point, a loss of 11 bits! If all characters were equally frequent in a set of text files, `utf-8` would be *very* inefficient, utilising as it does only about 2/3 of the storage. However, the underutilisation of space by the high-code characters has to be weighed with their expected frequency and for a Western and even CJK text would be infinitesimal. Notice that three-byte characters also represent with a loss (of 5 bits), and for CJK text they would be most frequent. On the other hand, Two-byte characters are still a gain of 10 bits and there are 2048 such, which makes `utf-8` very efficient for non-CJK, non-English users.

**Text processing in Unicode** In section 3.12.5 above we discussed the basic text processing toolkit which has a very simple implementation if the representation of text is as a sequence of equal size characters, bytes in the case of ASCII. If unicode is to serve as a basis for text processing we must demonstrate that the same toolkit is feasible here. The operations of the toolkit still work on a *sequence*, but it is primarily a sequence of bytes, rather than characters. The characters of the text string also form a sequence, but members of that sequence take from 1 to 4 bytes according to the secondary encoding in figure 3.3.

Let us consider each of the operations in turn:

**empty(s)** Since unicode under `utf-8` includes 8-bit ASCII with the most significant bit equal 0, clearly this operation is identical to the ASCII version. It simply checks whether the first character in the sequence is NULL.

**firstchar(s)** This is slightly different from the ASCII version. The operation must take the first byte of the string, count the number of 1's before the first 0 starting at the most significant bit to obtain the size of the character in bytes. Then it should read that many bytes and extract the code point by coalescing the code bits in them according to the format in figure 3.3 to produce a single 21-bit number. That number is yielded, possibly after padding with 0's on the left to reach a machine word size (e.g., 32).

**rest(s)** Slightly simpler than the above; the operation will read one byte off the sequence, determine how many more bytes belonging to the current character there are and skip them before it yields the rest of the string.

**extend(s,L,c)** is in a way opposite to **firstchar**. The character **c** is supplied as a code point; it is converted into a sequence of bytes and inserted before the NULL terminator in the sequence. Notice, that the value of **L** is the length of the sequence in *bytes* rather than characters; there was no difference between the former and the latter for ASCII, but now there is.

## 3.13 Addresses

The Central Processing Unit (CPU) of a computer can only hold a limited amount of data at any time. Typically the data held within the CPU is stored in a set of internal locations that can be accessed very quickly, and that can be used for manipulating the data values. These locations are known as *registers*. In a small processor the number of registers is very limited indeed. In the case of CdM-8 there are only 4 “general purpose” registers that can be used in this way, and each of them is capable of holding a single bit-string of length 8. The bit-strings representing the data we wish to manipulate must be stored elsewhere, copied into the CPU (one chunk at a time) so that operations can be performed, and the results of performing those operations must be copied out (also one chunk at a time) for storage away from the CPU, so that they can be used in future. We call the place where bit-strings are stored the *memory* of the computer. In order to store and retrieve an item the machine has to know where to find it, or its memory *address*. Supplying an address in order to retrieve or store a data item is called *addressing* and it can be done in one of the following two ways:

1. *Addressing by location.* In this case we have to specify the location of the data item typically by providing a single-number address.
2. *Addressing by content.* Addressing by content involves associating a ‘tag’ to an item, which acts not as a location, but rather as the item’s distinguishing characteristic. For example if the item is an employee record (name, age, position, etc.), the full name (assuming that all full names in the organisation are unique) can be used as the tag, with the rest of the information constituting the data item proper.

In this text we will concentrate on addressing by location, because that is what is supported directly by the technology at Levels 1, 2, 3, and 3½. Addressing by content can be achieved programmatically, but that is beyond what we want to consider at this stage<sup>38</sup>.

### 3.13.1 Representing the location of a data item in memory

We have only one tool at our disposal for specifying a memory address: the bit-string. As you have already seen, the length of the bit-string that is employed dictates how many different addresses are available. So the length of the bit-strings used to hold addresses dictates the *maximum* number of memory locations a computer may address. A different data item may be stored at - and retrieved from - each of these different addresses. In the case of CdM-8, addresses are 8 bits in length, so we say the CdM-8 CPU has an 8 bit *address space*. This means that the maximum number of uniquely identifiable memory locations the CdM-8 CPU can cope with is  $2^8 = 256$ , and we may attach it to a memory that has up to 256 different locations in which data items may be stored.

### 3.13.2 How much data at each location?

Whilst it is, in theory, possible to give every bit in a computer’s memory its own unique address, this is expensive, impractical, and unnecessary. Instead we divide the memory into  $n$ -bit “chunks”, each of which has its own unique address. The question, then, is how long is a bit-string that is referred to by a single address? This is what is known historically as a *byte*. The original definition of a byte was “the smallest unit of independently addressable memory” for a platform, and different platforms had different sizes of byte. With the widespread adoption of 8-bit microprocessors (which could address and manipulate data in 8-bit chunks) the byte became standardised as a bit-string of length 8. So a byte is *still* “the smallest unit of independently addressable memory” for a platform (in most cases, at least), but now the length of a byte has been standardised at 8 bits. When a CPU has an  $n$ -bit address space this generally means that the maximum size of memory it can support is  $2^n$  bytes.

---

<sup>38</sup> Strictly speaking, Level 2 machinery can include some content-addressable memory (often used as cache-memory, i.e. fast short-term storage), but it is expensive and not always efficient. Location-addressable memory is by far the most common type.

### 3.13.3 Addresses as numbers

It is convenient to think of the bit-strings representing memory addresses as unsigned integers. Treating an address as a number means we can perform *address arithmetic*. This is useful when the bit-string representation of a single data item is of a length greater than 8, because such an item will need to be stored across several consecutive bytes (and hence several memory locations). Such a data item may be stored or retrieved by specifying its *start address*,  $s$ , and the number of bytes it occupies,  $b$ . Retrieving the item is then achieved by fetching the first byte from location  $s$ , the second from location  $s + 1$ , etc.<sup>39</sup>

The CdM-8 CPU has an 8-bit address space, so it can address memory locations in the range 0b00000000 to 0b11111111. These addresses may also be referred to as decimal numbers in the range 0 to 255, or as hexadecimal numbers in the range 0x00 to 0xFF. CdM-8 Platform 3<sup>1/2</sup> provides a 256-byte memory, each byte of which is uniquely identified by its own address.

### 3.13.4 Referencing and de-referencing

Addresses are numerical data items that are used for pointing at the locations where other data items are stored. One data item that *points at* another data item is known as a *reference* to the second data item. A stored reference is often called a *pointer*, but the two terms are generally interchangeable. Suppose  $s$  is the (start) address of some data item  $x$ . So  $s$  points to the location where  $x$  is stored.

- We say that the address  $s$  is *a reference to*  $x$ , and when we make  $s$  point to  $x$  we are *referencing*  $x$ .
- We say that  $x$  is *referenced by* the address  $s$ , and when we obtain  $x$  by following the pointer from  $s$  we are *de-referencing*  $s$ .

It is important to understand referencing and de-referencing, because they are used a great deal in computing. In high-level languages there may be support for implicit referencing and de-referencing, which obscures the fact that they are going on, but at lower levels there is no hiding from them. Also, because memory addresses are just bit-strings like other kinds of data, they can be stored in memory like other kinds of data as well. So we can have an address,  $a$ , that is a reference to  $b$ , which itself is a reference to a non-reference value,  $v$ . So we have to de-reference  $a$  to find the address,  $b$ , of the value  $v$ , then de-reference  $b$  to get  $v$ . This is known as *indirect addressing*.

---

<sup>39</sup> Strictly speaking, although the byte has always been and remains the smallest chunk a platform can address, most modern Level 2 platforms (but not CdM-8, which was deliberately designed to be simple) can access memory in more than one size of chunk: this could be 1, 2, 4 or even 8 consecutive bytes at a time.

## 3.14 Data structures

So far we have discussed individual data items that are defined in isolation: data items of this kind have a set of operations applicable to them, which process every bit of their representing bit-string. In a sense those data items are indivisible, and because of this they are often referred to as *atomic* or *simple* data items. True, they contain constituent parts (numbers contain digits, sets contain members and strings contain characters), but an individual digit is not a number, a member is not a set, and a character is not a string. There are of course one-digit numbers, one-member sets and one-character strings,<sup>40</sup> but they are represented and manipulated in the same way as larger numbers, sets and strings. Importantly we do not introduce operations on *parts* of these objects.

In the vast majority of cases the data we have to work with when solving practical problems is not atomic in nature. For example, a customer's delivery details may include a name, an address and a telephone number. Each of these could be turned into a character string and then the three could be concatenated to form a single string, but that would make it difficult to separate them out again for other purposes. What we need is a way of gathering together these atomic data items into an aggregate that we may treat as a single 'large' item, and from which we can extract one or more of the constituent items when we need to.

### Definition 11 : Data structure

A **data structure** is an *addressable* aggregate of two or more data items that provides a mechanism for *accessing* its component parts.

Data structures themselves can be gathered into higher level aggregates to form *compound* data structures. This is another example of hierarchical abstraction at work. Consider, for example, an entry in an electronic diary recording an appointment. It requires a date, a start time, a duration (length of appointment) a location, and a description. The date may be represented as a sequence of numbers (a data structure), the start time as another sequence of numbers (another data structure), the duration as a number (of minutes), the description as a character string, and the location as another character string. So we have a compound data structure made up of two simpler data structures and three atomic data items.

The function of a data structure is to combine components into a collection that can be manipulated as if it were indivisible, and to support the extraction, introduction, modification and reorganisation of individual components that form part of the collection.

### 3.14.1 Properties of data structures

There are many different ways to aggregate a collection of data items. However, all data structures have certain things in common. Each data structure has a dual purpose:

1. On the one hand, it is an *addressable container* for a collection of components that brings them together simultaneously.
  - Knowing that a particular data structure exists (stored somewhere in computer memory) is tantamount to knowing that all of its components exist at the same time.
  - The entire data structure can be referred to and manipulated as a single whole, and located by specifying its start address in memory.
  - An operation on the data structure as a whole may result in a change to one or more of its component parts, or to the relationship between them, but we still end up with a data structure of the same kind, identified in the same manner, upon which the same kind of operations can be performed.
2. On the other hand, a data structure provides an *access mechanism* that enables programs to locate and process individual components in a certain order and/or in a certain way.
  - We can perform operations on individual components that do not affect the data structure as a whole, and end up with a result that is not a data structure of the same kind.

<sup>40</sup> A one-character string contains one character and an "end-of-string" marker, so whilst a single character may be stored in one 8-bit byte a one-character string takes up two.

In order to achieve these two purposes, the definition of a data structure must explain:

1. what kinds of component bit-strings it contains;
2. how its component bit-strings are “glued together” to make a single, longer, bit-string;
3. how to locate the beginning and end of each component bit-string.

Data structures may be *static* or *dynamic* in nature:

- For a static data structure (a data structure of pre-defined size) the definition provides a *framework* into which components are slotted, a *map* that describes the location and size of each component within the framework, and the *data type* of the component item that will appear in each location.
- The definition of a *dynamic* data structure (a data structure whose size may change during the running of a program) still gives the data types of component items, but rather than providing the framework and the map in full it describes *how to construct* each of them as and when it is needed.

The simplest data structures - such as arrays and records - are regular in form, and are represented by bit-strings that are stored as continuous blocks of memory. A data structure of this kind that has a start address of  $x$  and is  $n$  bytes in length will occupy all memory locations from address  $x$  to address  $x + n - 1$  inclusive.<sup>41</sup>

However, there are cases where this is either difficult to achieve, or just an unnecessary constraint. In these cases the components are not all arranged in one long sequence in memory. In order to achieve this we employ *referenced* data structures (also known as *linked* data structures).

In a referenced data structure some of the data items are *references* (also known as *links*) to others. The referencing mechanism may vary from platform to platform. A reference may be a *pointer* to a location, or an *associative* reference, which provides a value associated with the location (typically referred to as a “key”).

- A pointer to a location can be de-referenced immediately, by following the pointer to the location and then reading the data item located there.
- An associative reference requires a search mechanism (such as a table that associates keys with locations), which has to be utilised to find the data item.

An advantage of pointers is their efficiency. A disadvantage of pointers is their lack of flexibility and the need to keep them consistent. Indeed if several pointers are pointing at the same data item, and the item is moved to a different location, all pointers to it stored anywhere in the platform must be changed to point at the new location before they may be used again. So it is imperative that any pointer to a data item is somehow registered with the platform for the lifetime of the data item.

When referencing is done by association, the data item being referenced can be relocated at any time provided that the table of associations is updated accordingly. All associative references to the data item continue to be valid, and it is not necessary to know how many of them there are, and where they are held at any given time. There are many other pros and cons associated with the various techniques for referencing items within referenced data structures.

### 3.14.2 Representing data structures

A data structure may be thought of as a single continuous bit-string, comprising a sequence of shorter bit-strings that are concatenated together, each of which represents a simpler data item. Ideally it should be possible to retrieve the whole data structure from memory in one go, and to manipulate it as one long bit-string. However, for a variety of reasons, this is seldom how things are in reality.

Platforms can typically only handle a bit-string all at once if it is of a specified length. So while a data item, whether aggregated or indivisible, may be represented by a single continuous bit string, for implementation

---

<sup>41</sup> Why not  $x$  to  $x + n$ ? Think about it ... the 1st byte is in location  $x$  (which is  $x + (1 - 1)$ ), the 2nd is in location  $x + 1$  (which is  $x + (2 - 1)$ ), the 3rd in location  $x + 2$  (which is  $x + (3 - 1)$ ), etc.

purposes it must be treated as being made up of a sequence of equal-sized chunks. Each of these chunks is a bit-string that constitutes part of the overall bit-string that represents the data item.. The length of bit-string that can be fetched, stored and manipulated by the platform engine all in one piece is called the platform's *word length*, and a bit-string that is of this length may be referred to as a *machine word*.

Some platforms can fetch and store data in 8-bit (1 byte) chunks, some in 16-bit (2 byte) chunks, some in 32-bit (4 byte) chunks, and some in 64-bit (8 byte) chunks. Some platforms that have large machine words can also work with smaller chunks of data. For example, current desktop computers tend to have a 64-bit word (which is what we mean when they refer to them as 64-bit machines), but many can also work with words of size 8, 16, and 32 bits.<sup>42</sup> Our own Cdm-8 engine has a very small, 8-bit, word, so no data item that is longer than 8 bits can be fetched, stored or processed all in one go. In fact no data item that is shorter than 8 bits can be fetched or stored either.

Bit strings tend to be partitioned into machine words within memory so that they can be fetched and stored more easily. This gives rise to two concerns: *granularity* and *alignment*. These two concerns come to the fore when the length of the bit-string representing a type of data item (whether it is a data structure or a single indivisible data item) is not a whole-number multiple of the word-size.

### 3.14.3 Granularity

The granularity of a machine is the shortest length of bit-string it can store, retrieve and manipulate. As mentioned above, some machines can handle several different “chunk-sizes”, but let's suppose for a moment that a machine has a fixed word length of 32, so that it can only fetch and store bit-strings of length 32.

What happens if we want to be able to fetch or store a data item represented by a bit-string of length *less than* 32 (such as a 7-bit string representing a single ASCII character)? We have to put it into a 32-bit word. Only the part of the word that represents the item will be used, and the rest is wasted space. So if we wish to build a data structure containing millions of individual items of this type, we will end up wasting a considerable amount of space. In this case we say that the *granularity* of the platform (i.e. its ability to accommodate small objects) is too coarse.

In the case of a 7-bit ASCII character stored in a 32-bit word nearly 80% of the word is wasted space. Even when the platform has a granularity as small as 8 bits we are wasting 1/8 of the space if we use an 8-bit string to store each 7-bit ASCII character.

There is a similar problem when the bit-string representing a single item is a little longer than a machine word. For example, if we want to store and manipulate data items that are represented by 33-bit strings on a machine that employs a 32-bit word. In this case we need two words to represent each data item, and nearly half of the space is wasted, but we also need to fetch, store, and manipulate two words every time we want to work with a data item of this type.

The solution when a machine's granularity is too coarse may take the form of *blocking*: combining collections of several small data items into a bigger block to improve utilisation of bit-string space in memory. For example, in a 32-bit machine we might choose to put a sequence of 4 ASCII characters into each 32-bit word. This would reduce the amount of wasted space from nearly 80% to just 12.5%. Blocking can be useful, but it can only be used effectively when the length of the machine word is 2 or more times the length of a data item, and it has its costs.

We still have to read/write a whole word when we wish to work with a bit-string that is of a shorter length, and when we wish to update one component part of the block we have to take care not to change its neighbours within the same block.<sup>43</sup> We also lose uniformity of access, by which we mean that the data item we wish to work with may be the first, or the second, or the third (or the *n*th) in the block, so the bit-string representing it may not start at the beginning of the word. This means we need to know *where* in the word we'll find it, and we need to have a platform that provides special machine operations that can be used to extract and manipulate the data item we want.

<sup>42</sup> Note that a single word may be several bytes long. The byte is the smallest chunk in memory that we can identify / address on its own, but this is not necessarily the minimum length of bit-string that can be read in at one time by a machine. For example, some machines can only cope with reading or writing a whole 4-byte word at one go. In order to make hardware memory interchangeable between platforms we adopt byte-by-byte addressing as a standard, so in such a machine only every fourth address (0x00, 0x04, 0x08, 0x0C, 0x10, etc) will be a legal place to start reading or writing a data item, and all data will be read and written in chunks of four bytes (32 bits).

<sup>43</sup> There are also some really nasty situations that can arise when two programs running concurrently both have access to the same data structure, but that is an issue for another textbook.

### 3.14.4 Alignment

Alignment is another issue. If a component is represented by an  $n$ -bit string, where  $n$  is less than the word length, and a data structure is made up of a sequence of those components, there are likely to be occasions when a word boundary falls part way through one of the components. In other words, one or more  $n$ -bit data items will sit in the data-structure bit-string in places that straddle a boundary between consecutive machine words. When this occurs the platform will not be able to fetch the whole of a data item in one go, even though it is less than one word in length. So two words will have to be fetched and manipulated just to get to the point where we have a bit-string that we can work with.

Things can get really complicated in some cases. For example, suppose we wish to store the null-terminated string of ASCII characters that represents the word “aligned” on a machine that has an 8-bit word length. This string contains 7 printing characters and an ASCII NUL, so it requires 8 ASCII codes, each of which occupies a 7-bit string. The ‘obvious’ thing to do is to concatenate them all into one long bit-string of length 56 ( $8 \times 7$ ), which means the character string can be stored in 7 words.

The first character will occupy the first 7 bits of the first word. The second character will occupy the last bit of the first word and the first 6 bits of the second word. The third character will occupy the last 2 bits of the second word and the first 5 bits of the third word,etc, until we get to the null character, which occupies the last 7 bits of the seventh word. In order to obtain a specific character we need to know three things: the number of bits in a character, the number of bits in a word, and the position of the character in the string. We then need to work out in which bit of which word the character code starts and ends, and fetch either one or two words depending on whether it starts or ends on a word boundary, or is split across two words. Then we need to manipulate the words we have fetched to extract the correct 7 bits. All this just to obtain a single character from an ASCII character string!

So, whilst we may save memory space by packing the components of a data structure together tightly, misalignment of this kind makes handling a data structure very inefficient. The way around this is to insist that *each* bit-string representing a component of a data structure should occupy a whole number of words (this requirement is one of the reasons why some platforms have more than one word-size). In the case of a null-terminated string on a machine with an 8-bit word the obvious solution is to store each ASCII character in its own 8-bit word and to take the hit associated with ‘wasting’ one bit in 8.<sup>44</sup>

At first sight it seems obvious that the best approach is to have a machine that stores and retrieves very small words. This means that we can have very fine granularity, and it reduces the number of problems with alignment. However, a machine with an 8-bit word takes 4 times as long to store, retrieve and process a 32-bit string as a machine with a 32-bit word, and 8 times as long to store, retrieve and process a 64-bit string as a machine with a 64-bit word, so there are efficiency savings to be had by adopting longer word lengths.<sup>45</sup>

We now begin our exploration of data structures with those that are reference-free as they serve as a basis for more complex ones.

### 3.14.5 Arrays

An array is a static (fixed-length) data structure that consists of a specified number of data items *of the same type*, each of which is represented by a bit-string of the same length. Arrays are therefore *homogeneous* data structures (each component of an array is of the same type as every other component). Each of these items is called an *element* of the array, and the type of an item is said to be the type of the array. The number of items it contains is referred to as the *length* (or the size) of the array. The bit-string representing an array of  $n$  elements, each of which is represented by a  $k$ -bit string, is formed by concatenating the  $n$  elements in a given order, to form a single  $(n \times k)$ -bit string. The whole thing is stored in memory as one long bit-string starting at a specified memory address.

The *framework* of an array is therefore an ordered sequence of  $n$  equal-size bit-strings. The *map* tells us where the each element is located within the framework. In the case of an array the first element has the

<sup>44</sup> The ASCII code is stored in the least significant (left-most) 7 bits of the word, with the most significant bit set to zero. However, most platforms make use of the fact that there are a further 128 8-bit patterns that have 1 as their most significant bit and use them to represent a further set of ‘special’ characters.

<sup>45</sup> This is particularly important when dealing with numbers, which are typically represented using 32-bit or 64-bit strings, and which need to be stored, retrieved and processed in 32-bit or 64-bit chunks.

same address as the array itself, and subsequent elements start a multiple of  $k$  bits after that. The *type* tells us how to interpret each element, and what operations may be performed upon them. For example, we might declare that we are using a size-15 array of 8-bit two's complement integers. The framework is a collection of 15 identical items, stored in order. The map tells us that each element occupies 8 bits, and they are stored in consecutive memory locations, starting at a specified address. The data type of the elements (they are two's complement numbers) tells us how the bit-strings should be interpreted, and what operations may sensibly be applied to them.

A data structure is also supposed to provide an access mechanism. For an array it is as follows. Each element of an array of length  $n$  may be uniquely identified by an integer number ranging from 0 to  $n - 1$ , called its *index*. An element can be accessed *directly*, at a *fixed cost* (in a fixed amount of processing time, using a fixed amount of temporary workspace,etc) by specifying a valid index (i.e. an index that is within the range  $0 \dots n - 1$  for an array of length  $n$ ). The element selected by the index is a data item in its own right: it can be read, replaced or altered at will, provided that this does not change its data type or its size in bits. The representations of all elements are concatenated in the ascending order of index to form the representation of the array as a whole.

Memory can be pre-allocated for storing an array, since the bit-string that represents an array has a known size (which is the size of one element times the number of elements). As with any data structure supported by a platform, an array must observe the platform's granularity and alignment. So each element has to occupy an identical fixed number of words. Array elements should consequently be of a type that allows an efficient representation in a fixed number of whole bytes.

### 3.14.6 Array indexing

The location of an array is assumed to be the address of its first element (the element with index 0). This is sometimes called the *start address* of the array. The location of an arbitrary element in an array where elements are of size  $k$  bytes can be computed easily *at run time*. If the element has the index  $i$  its location is calculated by adding  $i \times k$  to the start address of the array. The number  $i \times k$  is known as the *offset* of element  $i$  from the start of the array. For example, a 23 element array of 16-bit numbers occupies 46 bytes, the first element of which (element 0) is stored in the two bytes beginning at the array's start address,  $s$ . The offset of element 5 is 10 bytes, so it occupies bytes  $s + 10$  and  $s + 11$ , and the offset of element 22 is 44 bytes, so it occupies bytes  $s + 44$  and  $s + 45$ .

Whilst it may be necessary to know the position of the whole array when a program is written, we do not need to know which element will be required to perform a particular operation until the program is running. Because of this programs can compute indices at run time. We can even use one element of an array to point to another. It is important to understand that the changing content of arrays, and the opportunity to calculate array indices at run-time, can make them hazardous to a programmer. It is especially important that the programmer pays attention to the array size, because if (s)he does not it is possible that the program will generate an array index - and consequently calculate the address of an element - that lies *outside* the array. This is known as an *out of bounds* index.<sup>46</sup>

When an array is referenced by an index that is out of bounds, any value that is fetched will be unpredictable, and any change that is made may do unpredictable damage, because the bit-string in question is not part of the array. The unpredictable values and damage stem from the fact that each array, as well as most data, is embedded in a larger data structure. The computer memory is typically the largest data structure that exists on any given platform, and every data item tends to be embedded in it. The array access mechanism in its simplest form relies on the program not using out-of-bounds indices at run time. If that is not guaranteed, the access mechanism *may* accidentally provide access to parts of memory that do not belong to the array and that are potentially taken by other data items. This is known as *memory corruption*, and it is a programming error that is extremely hard to diagnose as both the data and the program itself (which is nothing more than yet another data structure held by the platform at run time) may be changed, destroying all traces of the cause as well.

---

<sup>46</sup> If we have an array of  $k$ -bit elements of length  $n$ , with start address  $s$ , all references to elements must be in the range  $s$  to  $s + (n - 1)k$ . Any indexing calculation that comes up with a different address points at a memory location that is *out of bounds*.

### 3.14.7 Arrays of character strings

The requirement that all elements of an array are of the same type is to do with the requirement for a simple, fixed-cost, way to calculate the offset of each element. This requirement can only be met if every element is represented by a bit-string of the same length, so an array of NULL-terminated strings is an impossibility, even though all the elements are, at first glance, of the same type. There are two conventional ways to implement arrays of NULL-terminated character strings:

1. Use an array of memory addresses, each of which is a pointer to a NULL-terminated character string. Memory addresses are of fixed size, and can be indexed as well as any other fixed-size data items.
2. Dictate that no string can exceed a known *maximum length*. Then the bit string for each element can be of a fixed size (equal to the maximum length of a character string in bytes). In this case the initial part of the bit-string is taken up by a NULL-terminated string (up to and including the NULL character) and the rest of the bit-string is ignored, so it is wasted.

The advantages of the *pointer-based* representation are that strings can be of any length with very little waste, and a string can be stored anywhere there is spare space in memory. The disadvantage is that accessing a string requires two look-ups: first find the address of the string in the array, then find the string itself. The advantage of the *maximum length* representation is the cost of access (fast, based on offset calculation), the disadvantage is the memory waste, which depends on the difference between the average length of character strings in the array and the guaranteed maximum length of a character string.

### 3.14.8 Defining arrays

To define an array type in this book we will use a table that sets out the number and type of its elements. We may also assign a name to the array type if we choose to do so. For instance, an array of 5 unsigned integers representing a single student's marks for an academic session may be defined as follows:

```
array: marks
```

size	type
5	unsigned

Here is how array element representations are packed into the representing bit-string of an array **a** of type **marks**:

a[0]	a[1]	a[2]	a[3]	a[4]
0n	1n	2n	3n	4n

The array's representing bit string is  $5 \times n$  bytes in length, where  $n$  is the number of bytes required to represent an unsigned integer. In CdM-8  $n = 1$ , because each unsigned integer is represented in a single byte, so the array takes up  $5 \times 1 = 8$  bytes, which is  $5 \times 1 \times 8 = 40$  bits. The red numbers below the box show the offset from the start of the array in bytes. In Level 4 platforms it is customary to denote an array element by specifying the array name followed by the index value/expression in square brackets: **a[3]**, **b[i+1]**, etc. We will use the same style in comments to Platform 3<sup>1/2</sup> programs. Given this definition, we can refer to the object **a** as being of type

```
size 5 array of unsigned integers
```

or as being of type **marks**, because we declared that the name **marks** refers to this definition. So the name **marks** can be used as a *type synonym* for the above.

### 3.14.9 Records

A record is a static (i.e. fixed-length) data structure that consists of two or more data items that may be of *different types*, each of which is represented by a fixed-length bit-string. Records are therefore *heterogeneous* data structures (the different components of a record may be of different types). The component data items are known as the *fields* of the record.

Each field in a record may represent a completely different kind of data from the others, and each field may be represented using a bit-string that is of a different length to the others. Even when two fields are of the same type it is usually the case that they are being used to represent different kinds of thing in the real world. The representations of the fields are concatenated in some order, similar to the way array elements are concatenated, to form the representation of the record as a whole.

The *framework* of a record is therefore an ordered sequence of  $n$  (possibly different-sized) bit-strings, each of which represents a different field. The *map* tells us the order in which fields are stored within the framework. The first field will have the same address as the record itself, and subsequent fields start at fixed positions after that. Unlike an array the start positions of the fields may not be equally spaced, so index arithmetic cannot be used to access them individually. The *type* of the record tells us the types of the component fields, and what operations may be performed upon them.

#### Example 4 : A date record

We might declare a record type called `date` that is made up of a day of the month (represented as an unsigned integer in the range 1..31), a month of the year (represented as one of the length-3 character strings “Jan”, “Feb”, “Mar”, etc), and a year number (represented as an unsigned integer in the range 1..4095). The framework contains 3 items, and we may choose any ordering for them that is convenient, as long as we stick to it for all records of the same type.

In this case we will choose `year` followed by `month` followed by `day`. The map tells us that the year field occupies the first 12 bits (a 12-bit binary number can represent any of the whole numbers between 0 and 4095), the month field occupies the next 21 bits (no need for a NULL, because all month names are of fixed length), and the third occupies 5 bits (a 5-bit binary number can represent all whole numbers between 0 and 31). These are concatenated to form a bit-string of length 38. The data type of the record tells us how each of the three component bit-strings should be interpreted, and what operations may sensibly be applied to them.

Of course a record of this type will fall foul of the constraints on word alignment, because platform word-lengths tend to be multiples of 8 bits. We could pad the representation to 40 bits, which is 5 bytes, but we would still have alignment problems when we wanted to extract the individual fields.

In practice we would pad each of the fields to a length that makes sense from the perspective of alignment on our chosen platform.

For example, on a platform with a word-length of 8 we would represent the year in 16 bits (2 bytes) instead of 12; the month in 24 bits (3 bytes), using a sequence of three 8-bit representations (one for each 7-bit ASCII character); and the day in 8 bits (1 byte). Thus we would end up with a 6 byte string for a whole record, with the first field being the 2 bytes starting at byte 0, the second being the 3 bytes starting at byte 2, and the third being the 2 bytes starting at byte 5.

Once again we need an access mechanism. For a record it is even easier to access an individual field than it is to access an array element: we just need to know the offset of each field from the beginning of the bit-string that represents the record.

A field offset differs from an array index offset in that it cannot be calculated based on the ordinal number of the field (such as first or second), since fields, unlike array elements, may be represented by bit strings of differing lengths. So the program needs to know where each field will be located with respect to the start of the record before run time. This information can be gathered into a single offset table, where each field is given a name, a type and an offset. The information is static, i.e. it can never change in the process of program execution.

### 3.14.10 Defining records

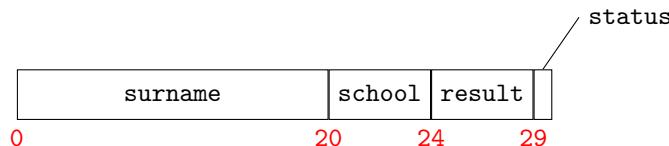
We will adopt a similar approach to define record types in this book to the approach we adopted for arrays. Once again we use a table, but this time it has multiple rows and three columns. Each row corresponds to a single field of the record. Each column declares a specific property of that field: the first declares its name; the second declares its data type; the third declares its offset (in bytes) relative to the start address of the record.

Here is an example:

```
record: student
```

field	type	offset (bytes)
surname	size 19 string0	0
school	size 4 array of char	20
result	marks	24
status	int	29

And here is the bit-string layout:



This record has four fields, of lengths 20 bytes, 4 bytes, 5 bytes and 1 byte, appearing at offsets 0, 20, 24 and 29 bytes into the bit string that represents it. Names are given to the fields for ease of reference (by human beings) in platforms 3 and above. The offset for a particular field is calculated by adding together the lengths of all the fields that precede it in the representing bit-string.

The field **surname** is placed first (offset 0) and contains a NULL-terminated character string of up to 19 characters in length, giving a field length of 20 bytes (ASCII characters packed into 8 bit chunks with the most significant bit always 0). Next there is a field **school** which contains a size 4 *array* of characters: the intention here is to store a 4 character acronym that takes the place of the school's full name. Next, there is an array of results of type **marks** (defined earlier), and finally there is the field **status**, which is an 8-bit (signed) integer. So the size of the whole record is 30 bytes or 240 bits.

A record type is fully defined by a table such as the one above. The order in which fields are listed in the table is essential as well as the content of each row, since the same fields in a different order would have different offsets, so the record access mechanism would not be able to provide correct access to the content. The names given to fields, if they are used in a program, are required to be unique in a given record type, so whilst two fields of a record may be of the same type, no two fields can have the same name. Otherwise it would be impossible to work out which field was required, and to determine the correct offset to use.

The above record as a whole may represent a student's exam results. Notice that the fields are data items in their own right and as such can be read or updated using appropriate operations on such items. Some of the fields are indivisible (e.g. **status**) , and some are data structures themselves (**result** is of an array type defined earlier).

Records are an example of a software aggregation mechanism. Real-life objects can be abstracted as collections of fields representing the object's important attributes and then platform programs can be introduced that process them, including create new records, update existing records, and merge & split records. Even though a platform may not provide operations for manipulating aggregates (low-level platforms typically do not), we can declare data structures and add software so as to "virtualise" a type of data item, making it "as if" it is an atomic value, which can be processed using "instructions", which are in fact programs, subprograms or instruction aggregates known as macros.

### 3.14.11 Null-terminated strings vs arrays of characters

You might be inclined to ask why the first field in the definition of a `student`, which we have called `surname`, is not defined as an *array* of characters. The reason is that an array requires not only simultaneous existence of all array elements (which would be the case here as well) but also a mechanism for accessing an element that does not depend on its index. Such a mechanism is, unfortunately, not available.

In an array, the element with a given index either exists or it does not, and whether or not it exists depends *only* on the size of the array.

- For an ordinary array the size is part of the type specification and checking whether an element exists is a matter of comparing its index with the array size, at fixed cost.
- If a variable-length character string were stored as an array, a character indexed  $k$  would either be a legitimate element or not, depending on the length of the string within the array.

Assuming that we still use a NULL-terminated representation, this means that any index that references either the NULL character or a character that is further on in the string is out of bounds. Only those indices that are less than the position of the NULL correspond to legitimate elements. Checking this requires scanning the string up to the given index  $k$ , which is zero cost for the initial character ( $k = 0$ ) and is proportional to the string size for the last, which is in violation of the array cost requirement.

So NULL-terminated strings are *not* arrays of characters and should not be treated as such. Their access is in fact completely sequential: in order to get to a  $k$ th character (counting from 0), the previous  $k$  characters must be read and examined.

## \*3.15 Variant records

Quite often a category of real-life object can take more than one form, and these different forms may be abstracted in different ways, depending on what sub-category we are dealing with. For example, a vehicle might be a bicycle, a car, a motorbike, a van, a lorry, or a tractor. Each of these shares certain common characteristics with the others, but there are also some characteristics that may belong to one type but not to another.

Imagine that the objects to be modelled can only be motorcycles or bicycles. For a motorcycle, we would naturally include the engine capacity as a field in the record, while for a bicycle this is neither necessary nor useful, but the frame size is required for a bicycle, but does not belong in a record describing a motorcycle. Yet a program could exist capable of processing either vehicle, for instance, to display its details on the website of a shop that sells both bicycles and motorbikes.

What is needed is a record type that has more than one variant of structure. Which variant is used will be indicated in the record itself, but the set of variants must be known in advance and must not change at run time. This can be achieved by including a field at the beginning of the record, which is common to *all* the variants, but which indicates which variant is in use. Such a field is called the *selector*, and usually contains an unsigned integer value. The selector field indicates which variant of the record structure should be used to look up the names, offsets and types of the fields in the record.

The selector field contains the *variant number* which can take one of only a very few values: just enough to number all of the different variants. Generally very few variants will be required; in the example above we need only two - one for bicycles and one for motorbikes.<sup>47</sup> In order to use variant records, a *variant table* is required which lists variant numbers against the record types associated with the different variants.

### 3.15.1 Declaring variant record types

The declaration of a variant record type requires two things: a set of ‘ordinary’ record declarations - one for each different variant- and a table that associates the different variants with variant numbers.

For example, suppose we identify two categories of people within the school system: *students* and *staff*.

We have already declared a record type **student**, and now we declare an (ordinary) record type **staff** of the following structure:

```
record: staff
```

field	type	offset (bytes)
surname	size 19 string0	0
staffNo	unsigned	20
payGrade	unsigned	21

Suppose we wished to create a variant record (named **person**) to enable a program that computes, for instance, teaching room occupation statistics, to process two variants of **person**, we could unite both record structures using the following variant table

```
vrecord: person
```

variant no	type
1	student
2	staff

When dealing with a data item of type **person** the program would first check the selector which, for any variant of this record type, occupies one byte at an offset of zero in the representing bit string. If the value of that byte interpreted as a binary number is 1, then the rest of the string is a data structure of type **student**, and if it is 2 the rest of the string is a data structure of type **staff**.<sup>48</sup>

<sup>47</sup> The precise nature of the selector is completely irrelevant; any type of data item that will take up a very small amount of space in the bit-string representing the record could be used — unsigned integers, however, are quite convenient.

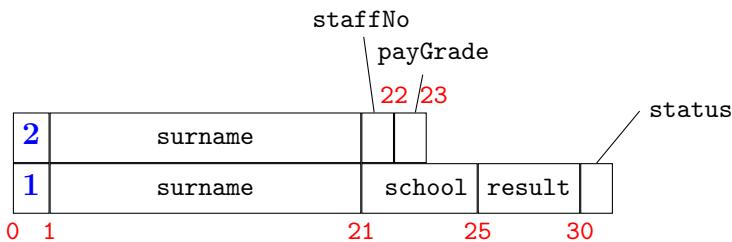
<sup>48</sup> Note that if a value other than 1 or 2 appears in this byte there is an error, because the variant table has only two entries, and they are numbered 1 and 2.

If we need to deal with staff pay grades, then we must write our program in such a way that it first checks whether the selector value (the variant number) is 2, and if not, proceed to another record. If the variant number is indeed 2, then the pay grade is contained at offset 21 from the beginning of the staff record, which is offset 22 from the beginning of the variant record, allowing for the one byte selector.

Notice that the sizes of the two variants are different: one is 31 bytes, the other 23. Since the information about which variant is in use is part of the record itself, a program processing data of this type should assume the maximum size. Indeed, the fact that a program accepts objects of type `person` means that it accepts both variants, so any allocation of bit strings for data should be sufficiently large to accommodate both. Also, if different variants were assigned different amounts of storage space it would not be possible to create an array of records of type `person`.

The downside is that an array of items of type `person`, of size  $n$ , would require  $31n$  bytes, rather than  $23n$ , even though it might end up containing only records of variant 2. So  $8n$  bytes would be “wasted”.

Here is the layout of the representing bit-string:



Notice that the bytes with offsets between 23 and 30 when the selector is equal to 2 represent wasted space and can contain any sequence of zeros and ones. The program processing an object of type `person` variant 2 must not and cannot access these offsets, since no name/type was declared for them in the `staff` record type.

One question: if a fundamental characteristic of a data structure is the simultaneous existence of all its components, why are variant records data structures? Surely only one variant exists at the same time for a given record? The answer is: the variants of a record are variants of the bit-string interpretation. It can be interpreted as variant 1 or as variant 2 depending on the selector. All parts of the data structure exist at the same time, no matter which variant is in use: the variant and offset tables are only used to determine how the bit-string should be interpreted.

### 3.16 General data structure types

The definitions we have given for arrays and records in earlier sections are what is known as *type constructors*. Each of them provides a method for constructing a data structure from components that are either atomic data items or other, smaller, data structures.

There is no limit to the level of nesting of data structures, beyond the practical limits that are imposed by the characteristics of the platform. So components can themselves be data structures that take as components further indivisible data items and data structures. Consequently, data representation can be arbitrarily complex and hierarchical: records containing arrays of records of strings, etc.

All that is required to maintain the ability to handle data structures is access to the relevant record and variant type tables and to array types and sizes. As long as these are known before run-time, and are not changed by the execution of the program, the program can always access any component of any data structure by following offsets, calculating array offsets from array indices and extracting aligned atomic data items that can be dealt with directly using the basic operations directly supported by the platform.

## 3.17 Addresses and pointers

### 3.17.1 Address consistency

Early platforms often had a small machine word. In such systems it was sometimes necessary for addresses to occupy more than a single machine word in order that the machine might support a memory of sufficient size to perform useful work. This is especially true of practical 8-bit platforms, which can do useful work but which often need more than  $2^8 = 256$  machine words to accommodate programs and data.

Since memory needs to be accessed in as few steps as possible (the number of steps affects platform efficiency in the most dramatic way), this makes such machines less efficient. In larger machines it is usually the case that addresses are unsigned integers occupying exactly one machine word each. This still limits the amount of memory a platform can use, or as we say *address*.

**Definition 12 : Address consistent** A platform is said to be **address consistent** if its address space (the number of different memory locations that can be addressed) is the same as the number of different bit strings of the same length as a single machine word.

An address-consistent  $n$ -bit machine can therefore support a memory of up to  $2^n$  words.

All modern platforms are address consistent, though it is most common for the smallest unit of addressable memory to be an 8-bit string (one *byte*). If a platform has an address space that is either larger or smaller than the number of different representations available in one machine word it is said to be *address inconsistent*.

Until recently, most commercial desktop platforms were 32-bit, meaning that the machine word was 32 bits in size, since the platform engine could process numbers of at most that size at once. Address-consistent 32-bit platforms can support a maximum of  $2^{32}$  different memory addresses, so for all practical purposes platforms (and hence the programs that ran on them) were limited to 4GiB of memory ( $2^{32}$  smallest machine words, i.e. bytes).<sup>49</sup>

With the introduction of 64-bit hardware platforms,  $2^{64}$  different addresses became available to each machine, so the current limit on computer memory stands at  $2^{64}$  bytes per machine, which is in excess of 10 million terabytes. This is approximately 4 billion times the previous limit, and it seems likely that it will not prove to be problematic for some time.

It should be noted that a machine does not have to have a full complement of memory to be address consistent. The issue is the size of the *address space*, not the number of bytes of memory present in the platform. The standard way to deal with a memory that does not fill the entire address space is to restrict addresses to the low-end of the number range. So when a 64-bit machine has 16 GiB of memory only the low-order 34 bits of the address are actually used. The high-order 30 bits should always be zero.<sup>50</sup>

Back to CdM-8 Platform 3½, which has, as will be revealed in the next chapter, a 256-byte address space. It is address consistent since it is an 8-bit machine, so addresses fit in one machine word in memory and there are no 8-bit strings that cannot be used as addresses. Furthermore, CdM-8 Platform 3½ provides 256 bytes of storage, so every address in the address space has a corresponding byte in memory.

### 3.17.2 Pointers

Each address - whether given or calculated - can be used to identify (and hence point at) a memory location. However, the address tells us nothing about what is stored in its associated memory location - not even what kind of data item we will find there.

**Definition 13 : Pointer**

A **pointer** is an address that *points to a data item*. It may be an indivisible item, or a data structure, or an item that is a component of a data structure, but a pointer will always contain the *start address* of the item in question.

By convention data items are never placed at the address 0, which in most cases is used to store instructions

<sup>49</sup> 1 gigabyte, 1GB is 1 billion ( $10^9$ ) bytes. 1 gibibyte, 1GiB, is the nearest power of two to 1 billion bytes, which is  $2^{30}$  bytes. This is  $1024 \times 1024 \times 1024$ , which is 1,073,741,824.

<sup>50</sup> The organisation and addressing of large memories is a topic requiring a whole different textbook.

(as opposed to data) or special data that belongs to the operating system, which we will discuss at the end of the book. The value 0 when used by a pointer does not mean the address 0 of the memory space. It is a special value interpreted as an absent item, a reference to nothing. Pointers can be and often are checked on being pointers to nothing, or so-called a “null-pointer”<sup>51</sup>. Finally, notice the difference between pointers and memory addresses. A pointer, if it is not a null-pointer, *must* point to a memory address occupied by a data item or its first byte (in the case of a multi-byte item). When a pointer is loaded with an address that is not so occupied, the pointer is invalid. In such a case it is often called a *hanging reference*. Hanging references may appear as a result of reclaiming memory occupied by a data item without simultaneously adjusting all pointers that point to it.

The significance of pointers will become clearer in the next section.

---

<sup>51</sup>Notice a low-case “null” as opposed to an upper-case NULL, which we use as an ASCII character and which is sized 7 or 8 bits in all cases.

## 3.18 References and linked data structures

Each pointer is a *reference* to a data item, and the item it points to is said to be *referenced* by that pointer. Armed with pointers we can do things hitherto unavailable to us. For instance, we can have an array of pointers to NULL-terminated character strings, without specifying a maximum length. NULL-terminated strings are of unpredictable size, but every pointer in an address-consistent platform is of the same size, namely the largest machine word, which, for our Platform 3 $\frac{1}{2}$  is exactly one byte. So we can legitimately have an array of pointers, each of which is the start address of a different NULL-terminated string stored elsewhere in memory (outside the array):

```
array: foo
```

size	type
5	->string0

Notice that the type name has an arrow `->` before it, which we use as a shorthand for “a pointer to a ...”.

The type `foo` defined above is an example of *linked* data structure. The character strings that the array elements point to are represented as bit-strings like everything else in a computing platform, and those bit-strings are stored somewhere in memory, properly granulated and aligned to byte boundaries. The address of the zeroth byte of a string is the value of its corresponding pointer, which is the corresponding array element in the array type `foo`.

It is important to understand that when we define a linked data structure it is not enough to just declare that a field (or an array element) is a pointer. It is also necessary to declare what kind of data the pointer points at (the data type of the item it references). Without this information a programmer won’t know how long a bit-string needs to be retrieved from the location the pointer points to, and it will be impossible to write a program to interpret the linked data. In any case, before writing a program to solve a problem, it is necessary to understand the nature and cost of the data structures that will be used so that an appropriate method can be devised for solving the problem. Proper declarations of data types - including pointers - also aid the automatic checking of certain aspects of programs by programming tools.

We will now briefly discuss some useful data structures that are almost as common as numbers, characters and sets to a large variety of applications.

### 3.18.1 Linked list

This is a data structure that offers minimum facilities for *sequential* access to data. Imagine that all the programmer needs to be able to do is maintain a sequence of objects of some type. To maintain a sequence means to

1. always locate the first member of the sequence at a fixed cost.
2. given a location of a member, find the location of the next one in sequence (or determine that the member is the last one) at a fixed cost.

You may be wondering why we need a new kind of data structure when we already have an access mechanism like this in an array. Indeed, given an index value  $k$  into an array of size  $n$ , if  $k = n - 1$ , we know that element  $k$  is the last one. Otherwise the value  $k + 1$  immediately gives the index of the next array element. All of that is true, and an array can do *more* than this *at the same cost*: it can instantly locate *any one* of the  $n$  elements that it accommodates. It is said that an array is a *direct access* data structure.

However, there are facilities that arrays do not support. If we wish not only to maintain the sequence of items but also to insert and delete members, we would find that this could not be done at all efficiently. In order to insert an element between those currently indexed  $k$  and  $k + 1$  we must first of all create a new array that can hold 1 more element than the current one. Since the array size is part of its type, we must know in advance that such an insertion will be required, and declare that type. Worse still, if several insertions are needed, or a deletion becomes necessary, the number of different array types that need to be declared in advance will become impractical.

We can devise a partial solution to this problem by using a larger array than we need and allowing those elements with an index above some number  $w$  to be “reserved space”. Each time a new element is inserted all existing elements with indices greater than the index of the insertion point will need to be moved up by one and the number  $w$  will need to be incremented. Each time an existing element is deleted all elements with indices greater than the index of the deletion point will need to be moved down one position and  $w$  will need to be decremented.

The cost of these operations is not fixed: the closer the insertion or deletion point is to the beginning of the array the more existing elements need to be moved to make or reclaim space. Besides, we will have to know in advance what the maximum length of the sequence will be, and whenever the sequence is not of this maximum length we will be wasting memory. We conclude that arrays are *not suitable* for maintaining and manipulating a sequence.

With the aid of pointers we can create a data structure — a linked list — that provides sequential access and permits insertion into and deletion from a sequence:

```
->record: list_of_X
```

field	type	offset (bytes)
next	list_of_X	0
payload	X	1

Where X is any type, an indivisible or a data structure, including a NULL-terminated string. The record is essentially a pair of fields, first is the address of another such pair, and the second is a “payload”, a member of the sequence that we wish to maintain and manipulate. A new aspect here is that the name of the record has the pointer symbol `->`. What this means is that `list_of_X` is a *pointer type*. If an object `a` is of type `list_of_X` then it is assumed that:

1. `a` is a pointer
2. `a` points to a record that consists of two fields
3. the first field (offset 0) is a pointer of the same kind as `a`
4. the second field (offset 1) is data of type X

It is easy to see that by following pointers we can locate the next member, the member after next, etc. What is not clear is how we know where to stop. As we said before, all pointers are assumed to recognise a special value 0, reference to nothing, a null-pointer. By using the null-pointer as the value of field `next` we mark the current pair as last.

Here is an example declaration of a list of signed integers:

```
->record: list_of_int
```

field	type	offset (bytes)
next	list_of_int	0
payload	int	1

An object `x` is of type `list_of_int` and contains the value<sup>52</sup> 13. The memory contains the following data (“...” stands for “don’t care” items)

---

<sup>52</sup> remember that pointers, being memory addresses, are interpreted as unsigned 8-bit numbers in the range 0-255; we use quantities rather than representations everywhere in this example merely for ease of reading; for a practical platform addresses will be shown in hex as they tend to be very long numbers, especially for commercially available machines.

address	content
...	...
10	15
11	7
12	...
13	10
14	5
15	20
16	-3
17	...
18	...
19	...
20	0
21	8
...	...

To see that  $x$  is in fact a sequence of integers, we follow the pointers.

- The first record is pointed to by  $x$  directly, location 13. The field **payload** is at the location with offset 1, which has address 14. We note the first member of the sequence, 5, then follow the pointer contained in the field **next** at offset 0, (location 13).
- The value there is 10, which points at address 10, where we expect to find a record of the same type. The second field of the record, **payload** at offset 1 (address 11) is the number 7. So already the sequence consists of two members (5,7).
- We continue to follow the **next** pointer to 15, where we find the number -3 as the **payload**, a pointer to location 20,
- At location 20 the **next** field is 0, indicating that this is the last member, **payload** 8.

The sequence is now complete: (5, 7, -3, 8).

Notice how the members of the sequence are scattered in memory. Unlike array elements they do not have to occupy a contiguous segment, nor even take addresses in any particular order: the first member in our example has a higher address than the second and lower than the third. In fact wherever a segment of memory may be free to use, a list record can be accommodated and linked with the rest of the list.

One must also understand that specific address *values* of the pointers are not important: a member can be moved to a different place in memory as long as the pointer to it in the previous member is adjusted accordingly. Only the payload defines the list content; the pointers are there to maintain the sequence and nothing more.

Let us now illustrate how members can be inserted and deleted. Specifically we will insert 9 between members 7 and -3:

address	before	after
...	...	...
10	15	18
11	7	7
12	...	...
13	10	10
14	5	5
15	20	20
16	-3	-3
17	...	...
18	...	15
19	...	9
20	0	0
21	8	8
...	...	...

The member with the payload 7 is represented by a record that begins at location 10.

Previously its **next** pointer pointed to location 15, where the record with the payload -3 was placed.

We want it to point to a newly inserted member with the payload 9. We place that member in free memory, for example, at location 18. Its **next** pointer should now point to the member following this in the sequence, which is the member with the payload -3, location 15, so we copy 15 into location 18.

Now the newly inserted member must be next to what it is being inserted after, which is the member in location 10 which we have already looked at.

So we must adjust that member's **next** field: change it from 15 to 18. Thus by changing only two pointers we have inserted a new member into the existing list sequence: (5, 7, 9, -3, 8).

Deletion of a list member is even easier. All we need to do is switch the pointer of the previous member.

To delete the *first* member the program may simply update the value of the pointer  $x$  of type `List_of _int`. Since it is pointing at location 13, the program could just replace that value by the content of the **next** field of the record at that location, i.e. 10.

Following the pointers we will discover that  $x$  now represents the sequence (7, 9, -3, 8). If we wish to delete a member other than first, say the one with the payload -3, then that can be done by changing the **next** field of the previous member to make it point to the successor of -3, which has the payload 8. That can be achieved by setting the content of location 18 to 20.

Notice that deleting the last member is done in exactly the same way: its successor is `nil`, so `nil` is copied to the **next** field of the predecessor.

There can be more than one pointer in a member-record of a linked data structure. For example, a list member can have a pointer to the predecessor in the sequence as well as the successor. This enables the program to walk the list in both directions efficiently:

```
->record: double_list_of_X
```

field	type	offset (bytes)
next	double_list_of_X	0
prev	double_list_of_X	1
payload	X	2

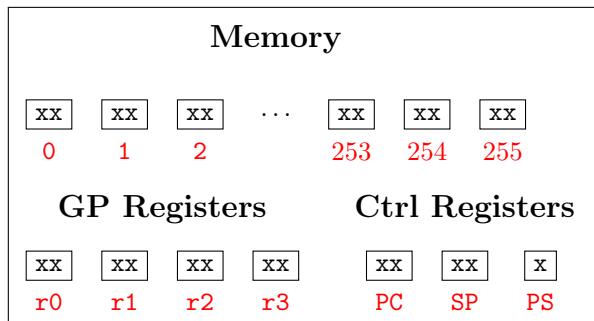
# Architecture and Operation of Platform 3<sup>1/2</sup>

As previously noted, a Level 3 (and a Level 3<sup>1/2</sup>) platform is largely a human-friendly wrapper for an underlying Level 2 machine. A platform's primary purpose is to offer its users (who in this case are programmers) a set of resources that they may use to solve problems, and CdM-8 Platform 3<sup>1/2</sup> is no exception. We refer to the set of resources, how they are organised, and how they interact with one another, as the *architecture* of the platform.

The resources offered by a Level 3 platform are mostly the same as those offered by its underlying Level 2 machine, except that they are offered in a form that makes them usable by a human programmer. The additional functionality that is offered is almost all concerned with making the programmer's job more straightforward (and less unpleasant). To this end CdM-8 Platform 3<sup>1/2</sup> (as well as CdM-8 Platform 3) provides programmers with human-friendly access to the resources provided by CdM-8 Platform 2, which is an address consistent 8-bit Level 2 machine. Alongside this it provides a set of tools for creating and testing programs that can be executed by the CdM-8 Platform 2 computing engine.

## 4.1 Resources

CdM-8 Platform 3<sup>1/2</sup> provides its programmers with access to a complete set of resources, some of which would ordinarily be implemented in hardware and the rest of which would always be implemented in software. We start by examining those that - in a 'normal' machine - would be hardware-based, and that are essentially Level 3 / 3<sup>1/2</sup> manifestations of hardware resources provided by the underlying Level 2 machine. These may be illustrated diagrammatically:



### 4.1.1 Memory

The memory provided by CdM-8 is a size-256 array of memory cells, with each cell holding an 8-bit string (one byte). Note that a 256-element array requires indices that run from 0 to 255: a range of unsigned integers that can be represented in exactly 8 bits. So each of the  $2^8$  different addresses we have at our disposal refers to a different memory cell, with none left without an address, making the platform address consistent.

An important aspect of the platform is that it is a von Neumann<sup>1</sup> machine, so its memory is used to store both data and the program that processes it. So, as well as the data types we have already met, an 8-bit

<sup>1</sup> American mathematician John von Neumann is considered to be the founding father of computer architecture

memory cell can hold a bit-string that represents a *machine instruction* or a part of an instruction, i.e. information about a single action to be performed by CdM-8 Platform 2. Contiguous segments of memory can be used to deploy parts of the program, while other segments can be occupied by data, and yet other memory segments may be left unused by a particular program.

### 4.1.2 Registers

Registers are like memory cells, but are usually internal to a Level 2 platform's central processing unit (CPU), so they do not have addresses like memory cells, and they are capable of supporting operations that cannot be performed upon memory. Registers are typically characterised by two things: their size (in bits) and what they can be used for. Some machines provide separate sets of special-purpose registers that may be used for performing specialist operations on particular kinds of data. CdM-8 provides four 8-bit "general purpose" registers and four special-purpose (control) registers, of which only three are of any significance to CdM-8 Platform 3<sup>1/2</sup>.

A machine's registers serve as a "scratch pad" for performing data manipulations and for keeping track of the state of execution of programs. The use of registers means that a platform has to deal with only a very small number of possible cells (rather than the whole of memory) when it needs to perform calculations or other operations:

- Machine instructions need not refer to locations in 'main' memory, so they do not need to include memory addresses, and can therefore be shorter
- Operations can be made to work faster, because registers are internal to the CPU

There are, of course, dedicated facilities in the platform that deal specifically with copying data backwards and forwards between the memory and the general purpose registers.

**General Purpose (GP) registers.** Each GP register is 8 bits in length. These registers may be used as temporary storage areas for data items that have been copied in from memory cells, and for those that are the (intermediate) results of computations. They also play a direct part in computation, because platform instructions can only perform operations on data items held in registers. In CdM-8 the GP registers are numbered 0, 1, 2 and 3, and we use the names r0, r1, r2 and r3 to refer to them in assembly language programs. Those are reserved names, which cannot be used in any other sense; this helps to ensure that registers are not confused with symbolic names of other entities that the programmer is free to choose.

**Control Registers.** Unlike GP registers these are not immediately available for arithmetic operations. Rather they are used by the platform for the purposes of addressing and control. Specifically:

- The Program Counter
- The Stack Pointer
- The Processor Status register

We now examine the structure and function of each of these in more detail

### 4.1.3 The Program Counter (PC)

The Program Counter is an 8-bit register that holds the *address* of the *next* machine instruction that will be executed.

The PC is therefore used as a *pointer* to the start address of the next instruction. The PC is crucial to the correct sequencing of a program and is involved in several classes of machine instructions. When an uninterrupted sequence of instructions is being executed it is a simple matter to update the PC so that it points at the next memory location. Each instruction occupies a known number of bytes, and when a byte is fetched the PC is immediately incremented to point to the next byte *before* anything else happens, in particular before the instruction starts to be executed. The PC cannot be accessed directly by a programmer, but its contents can be updated indirectly by *control* instructions.

#### 4.1.4 The Stack Pointer (SP)

The Stack Pointer is an 8-bit register that holds a memory address.

We will discuss the idea of a *stack* and stack exchange operations in section 6.3.

For now, we note that the address in the SP is used to point to an area of memory known as the *stack*. The stack is used for temporary storage of working data when we run out of registers, and also for managing subroutines, i.e. segments of code that are used repeatedly by temporarily passing the control to them from various points in the program (more on this in section 6.3).

#### 4.1.5 The Processor Status register (PS)

The Processor Status register comprises eight bits, four of which (the low order) are used exclusively for managing *conditional computations*. The other four will be dealt with in one of the last chapters, see section 13.5.1, when we discuss the concept of interrupt, which is to do with how a Level 2 platform interacts with its environment. Since we must first study platform organisation in and of itself, we will, for the time being, silently assume that the PS register only has the low-order four bits.

Quite often a program will be required to execute one sequence of instructions under one set of conditions and a different sequence of instructions under a different set of conditions. These conditions depend on the value(s) of one or more data items. For example, a program that computes the magnitude of a signed integer  $x$ , usually written  $|x|$ , needs to copy  $x$  into a GP register, and to examine the register value to determine whether or not the integer is negative. If  $x$  is negative the program must execute a sequence of instructions that cause the sign of the number to change; if not, the program should do nothing.

The PS register is updated by all operations that *test* the contents of a GP register or compare the contents of two such registers; also most operations that involve arithmetic affect the Processor Status. Thus the PS register serves as a temporary store for metadata (data about data); specifically information about the effects of a recently executed instruction. For example, when two numbers are added together, all four bits of the PS register are updated to reflect what happened (see below). The operations that update the PS register are identified later in this book.

The 4-bit PS register is traditionally viewed as a set of four 1-bit *flags*, each of which indicates something about the result of the most recent operation performed on a register by the computing engine. These flags are denoted C, V, Z and N, and they are interpreted as follows:

**C** is the *Carry* flag. This bit will be updated on completion of any arithmetic operation that produces a carry-out from bit 7 of the result. The carry-out value (whether 0 or 1) is transferred to the C bit of the PS register.

**V** is the *oVerflow* flag. This bit will be updated on completion of any arithmetic operation that can give rise to a *two's complement* overflow event. The flag is *set* if an overflow event has been detected and *cleared* otherwise.

**Z** is the *Zero* flag. This bit will be updated on completion of any arithmetic or logic operation. The flag is *set* when the result *is* an all-zeros bit-string and *cleared* when it is not.

**N** is the *Negative* flag. This bit will be updated on completion of any arithmetic or logic operation. It is *set* if the operation resulted in a bit-string that is, or would be interpreted as a negative number based on the sign bit. Specifically, the value of bit 7 is copied to the N flag, so N is *set* when the result is a bit-string that could represent a negative number and *cleared* otherwise.<sup>2</sup>

The PS register is examined by control instructions to change the PC (thus changing which instruction will be executed next) if a certain combination of flag values is present at this point. It is important to understand that the PS register is updated based on the category of operation performed (arithmetic/logic operation), not because the update is wanted by the program. If the result of an operation is to be used to control program execution the flags must be checked *immediately*, by the next machine instruction, unless the intervening instructions are such that they do not affect the flags.

<sup>2</sup> This description is a bit tortuous, because it is not a requirement for the bit-string in the register to be *used* as a negative number, or for it to be the negative result of some calculation: all the N flag tells us is whether the most significant bit of the bit-string was 1 or 0.

#### 4.1.6 Macro-assembly language

This is the language used by programmers to control CdM-8 Platform 3<sup>1/2</sup>. The CdM-8 Platform 3<sup>1/2</sup> macro-assembly language contains

- all of the operations and other facilities provided by the CdM-8 Platform 3 assembly language. In fact it includes that language as a sublanguage.
- some additional named Level 3<sup>1/2</sup> operations, each of which corresponds to one or more of CdM-8 Platform 3 assembly language instructions

The CdM-8 Platform 3 assembly language supports

- all instructions that are part of the CdM-8 Platform 2 programming language, except that they are expressed in human-readable form
- comments that may be used to explain to a human being what the program does and how it works
- a set of *pseudo-instructions* that are used by the *assembler* to determine how various parts of the program should be interpreted and where they should be located when it is transferred to the Level 2 machine

The CdM-8 Platform 2 programming language (also known as CdM-8 Platform 2 machine code) is the ‘native’ language of a CdM-8 processor. Every instruction in this language is an 8-bit pattern which is chosen with hardware implementation in mind.

The macro-assembly language also provides operations and control constructs that correspond to one or more machine code instructions (these are called *macros*, and are what raises it from Level 3 to Level 3<sup>1/2</sup>).

#### 4.1.7 Software tools

CdM-8 Platform 3<sup>1/2</sup> provides programs that translate macro-assembly language programs into CdM-8 machine code, and enable us to run those programs and examine the results of doing so.

A *macro-assembler* (*cocas*) and a *linker* (*coco1*) are provided that together translate a program from macro-assembly code and convert it into a single bit-string that will fill up the whole memory of a CdM-8 Platform 2 machine. A bit-string of this kind is known as a *CdM-8 Platform 2 memory image*. For the sake of brevity we will refer to the process of producing a memory image as *compilation* and the combination of assembler and linker as the *compiler*, even though two tools separate tools are involved in getting from source code to memory image.

We will look at this process in a little more detail later. For the time being we will treat the compiler as a “black box” that requires certain information to do its job, and will concentrate on identifying what information it needs.

## 4.2 How the platform operates

### 4.2.1 Initialisation

1. Before anything useful can be done, the platform has to be brought to a known *state*. This is required to avoid *nondeterminism*, i.e. a situation in which the same program run again can produce different results. While the program should not rely on the content of the registers and memory that it itself has not put there, it is convenient to have some fixed values placed there initially in case it does (for example, when the programmer has made a mistake). In the case of CdM-8 Platform 2, the platform is *reset* before every use, i.e. all registers are given the 8-bit zero as a content, and also every memory cell receives the same value.
2. A program is loaded into memory, along with its initial data in the form of memory image.

Note: Platform 3½ is a “cross” platform.<sup>3</sup> It does not run its own system software,<sup>4</sup> and the basic machine has no facilities for input from or output to peripheral devices, so it is incapable of supporting the editing or compilation of programs. It cannot even load a program into memory.<sup>5</sup>

Instead the program comes as a 256 byte *memory image* prepared by tools running on a commercial platform: Linux or Mac OS X, or Microsoft Windows. The memory image is stored in an image file (with the file extension .img). The platform emulator (or the circuit simulator, when we move on to look at how electronic circuits may be used to implement the platform) has the ability to load such files into the platform memory (the emulator resets the platform before a memory image is loaded).

An image file has the initial values of all 256 memory cells, whether they are to be used for instructions, or for data, or their initial content is left unspecified by the program (in which case the value will be set to 0 by the tools).

3. The initial value of the PC — after the program has been loaded — is set to 0 according to step 1 above, so the first machine code instruction the platform will execute is the one that starts at address 0.

### 4.2.2 Execution

When a program is run the first thing that happens is that an instruction is copied from the memory cell pointed to by the PC:

1. The platform copies the instruction into the machine’s (private) Instruction Register (this is called *fetching* the instruction), identifies what it is (*decodes* it) and *executes* it. The content of the PC is adjusted to point to the next instruction. This schedule of program execution is known as the *fetch-execute cycle*.
2. That next instruction can either be the one following the current one in memory, or an instruction at a different address, if the current instruction is a control one. We will discuss control instructions later, in section 5.8.
3. The fetch-execute cycle is repeated until an instruction is encountered that tells the engine to stop fetching new instructions, at which point the platform halts awaiting a reset.

---

<sup>3</sup> Not an angry one!

<sup>4</sup> CdM-8 is so tiny that no proper programming tools could possibly fit in its memory.

<sup>5</sup> Later we will explore an extension of the basic machine that is able to communicate with input and output devices, but even that is still unsuitable as a platform for programming tools.

### 4.2.3 Level 2 machine instructions

Each level 2 instruction is ultimately a bit-string that has been granulated into 8-bit chunks and aligned on byte boundaries, then stored in memory. However, at this stage we will only look at them through the prism of a rather advanced macro-assembly language, as this is what Platform 3<sup>1/2</sup> provides.

You will not need to examine the bit-string representations of machine code instructions. It is quite sufficient to understand that each Platform 3<sup>1/2</sup> instruction

- occupies one or more (consecutive) memory cells;
- may read or update the contents of GP registers and of memory;
- may update the PS register;
- may update the PC by an amount other than the normal increment, though most do not;
- may update the SP, though most do not.

# Introduction to Programming CdM-8 Platform

## 3<sup>1/2</sup>

Not even mainstream platforms but simply the majority of them are programmable. Programming is the art of creating sequences of machine instructions for achieving a certain transformation of data. The data is either initial data contained in the program (when the point of the program is to compute a result functionally dependent on it, for example to find the first prime number greater than a given one) or more often data values received from the platform's environment, e.g. a keyboard, mouse, network, etc.

It makes sense to first study the kind of programming that is used for computing a final result after which the program *halts*. Such programs use a platform on its own, and can be written, understood and executed in isolation. We shall approach the art of programming at the highest level of abstraction available to us in this book (and which we can eventually trace down to the lowest level of computer organisation, i.e. hardware elements). We must not go higher, lest we lose the ability to navigate the hierarchy of platforms as the transition between levels becomes impossible without sophisticated compilation technology. We must not go lower, as programming in a very low-level language is dominated by more or less standard instruction patterns, which it is our intention to study, but which can easily obfuscate the meaning and intention of a program if they need to be used explicitly and extensively.

Accordingly, in the current chapter we learn about programming for CdM-8 Platform 3<sup>1/2</sup>.

### 5.1 Writing (macro-)assembly language programs

A Level 3 assembly language program is a list of instructions written in human-readable form, each of which can be directly compiled into Level 2 instructions. The list also contains directives that guide the compiler in the process of translation, see pseudo-instructions below.

#### 5.1.1 Instructions

Each Level 3 or Level 3<sup>1/2</sup> instruction is made up of a named operation and a number of operands. The *name* takes the form of a *mnemonic*. A mnemonic is an English word or a contracted form of a word (e.g. `cmp` stands for “compare”) identifying the operation, which it is easier for a human to remember than the bit string used for the same by a Level 2 machine.

Whilst a Level 3 language has a one-to-one correspondence with a Level 2 language, the presentation of CdM-8 Platform 3<sup>1/2</sup> additionally uses instruction aggregates which the compiler (macro-assembler) inserts, adapts and expands every time such an aggregate is mentioned in the program. This makes it a little harder to see the Level 2 machine behind an assembler program, but we believe that the advantages outweigh the disadvantages due to the avoidance of standard repetitive patterns, as was explained earlier. A slightly higher level of abstraction also makes programs much more readable, since the macros, due to their mnemonics and succinctness, help to convey the algorithmic *intention* of the programmer.

#### 5.1.2 Pseudo-instructions

As is the case with any assembler-based platform, CdM-8 Platform 3<sup>1/2</sup> offers not only executable instructions but also pseudo-instructions. These are not intended for execution by the platform, but are merely directives

that tell the compiler how to compile the program: which memory locations will be set aside for data, what they will contain initially, where the various sections of machine code should be placed in memory, etc.

In this text we will typically refer to pseudo-instructions as instructions, except where it is important to differentiate between them and executable instructions.

### 5.1.3 Programs

In an assembly language program each instruction is placed on a separate line of text in a *source file*. There are a few rules that govern how a correct assembler program is composed from individual lines, and many of these involve *pseudo-instructions*. In particular,

- The program source must start with a *section command*. This is a pseudo-instruction whose purpose will be explained below
- It must finish with the word `end` on a line on its own. This is another pseudo-instruction that tells the assembler it must stop translating.
- It is also necessary to tell the platform to stop executing the program when all work has been done. This is achieved by the instruction `halt`

Here is a simple example program:

```
asect 0
fred: ldi r0, x
      ld r0, r1
      ldi r2, y
      ld r2, r3
      add r1, r3
      ldi r0, bill
      st r0, r3
      halt
x:   dc 10
y:   dc 25
bill: ds 1
end
```

### 5.1.4 halt and end

You may have wondered why a program needs both a `halt` and an `end`. The reason is that one can be translated into Level 2 machine code, and the other is a pseudo-instruction. The instruction `halt` corresponds to a machine code instruction that tells the Level 2 engine to stop executing the program, whereas the pseudo-instruction `end` tells the Platform 3½ compiler that it has reached the end of the source code and there is no more program to translate.

So why not just assume that `halt` marks the end of the program? Two reasons:

1. A program can contain both instructions and data, and the data may be placed between the `halt` instruction (which stops program execution) and the `end` directive (which marks the end of the program).
2. There may be more than one circumstance under which the platform should halt, so a program may contain more than one `halt` instruction.

In the example program the instructions `asect`, `dc`, `ds` and `end` are all pseudo-instructions. We have already dealt with the last of these. We will examine the use of the other three in Section 5.5

## 5.2 Literals

A literal is a text string that is included in the source code of a program that represents a specific value. For example, the decimal string 25 represents the quantity twenty-five. We use the term *literal* because the string is meant to be read literally. Most non-numeric text strings that occur in a program are references to registers, to memory locations or to instructions. Before we start discussing programs in more detail, let us introduce the rules for writing strings that represent literal data in assembly language programs.

### 5.2.1 Number literals

A number literal can be represented in an assembly language source file in one of the following formats:

**Decimal** A number in decimal format must represent a quantity within the range (-128..255) inclusively and be written using conventional decimal positional notation. So the quantity twenty-five is written 25. If the quantity represented in decimal is in the subrange (-128..127) the decimal number will be converted by the assembler into a two's complement 8-bit number; if it is in the subrange (0..255), it will be converted into an unsigned 8-bit binary number. Note that quantities in the subrange (0..127) have identical representations in 8-bit two's complement and 8-bit unsigned binary format, so there can be no ambiguity about which representation will be used.

**Binary** A number in binary format must always start with the two-character prefix 0b followed by exactly eight *binary digits*, so that the full bit-string representation is given. Thus the quantity twenty-five is written 0b00011001 in binary.

**Hexadecimal** A hexadecimal (hex) number is written as 0x followed by exactly two hexadecimal digits (digits in the range 0..F). You may use either upper-case (A..F) or lower-case (a..f) for the hex digits that are represented as letters. The quantity twenty-five is written 0x19 in hex.

Note that it is only in decimal numbers that we are allowed to use a minus sign. The other two are really just used as alternative ways of representing bit-strings. This doesn't mean that we cannot use binary or hex to represent a bit-string that will be interpreted as a negative number: it just means that we must reflect the underlying (8-bit two's complement) binary representation in the (binary or hex) digit-string we use, rather than prefixing the literal with a minus sign.

Our example program contains five number literals, each expressed in decimal format.

### 5.2.2 Character literals

We can also include ASCII data in a program. Of course this could be done by specifying the bit-string for a character, either in binary or in hex, just as it can for numbers. But this would mean looking up each character in the ASCII table, so a much more convenient representation is available. A single ASCII character is represented as a literal by enclosing it between quotation marks, for example "A" or ")" or "?". There is a special case of the character representing a quotation mark, ", ASCII code 0x22. Since the same character is used as a delimiter, the assembly language requires that, for the avoidance of ambiguity, the quotation mark as a character literal be preceded by the backslash: "\>".

It is worth noting that when a decimal figure is written within quotation marks (e.g. "6") the bit-pattern that is used is the one that represents the relevant ASCII character, **not the decimal digit**. So the character literal "6" is replaced by the bit-string 0b00110110 (=0x36) rather than the bit-string 0b00000110 (=0x06). Because each character is represented by a single 8-bit byte it may be stored in a single memory cell, and loaded into a single GP register.

### 5.2.3 String literals

A string literal is a sequence of ASCII characters enclosed in quotation marks, such as "hello". This will be converted into a bit-string which will be stored as a sequence of consecutive bytes, each of which represents a single character. This allows us to store a character string in memory without having to specify a list

of individual character literals. String literals are more than one byte in length, so they cannot be loaded into GP registers all in one go. However, we can specify that a particular sequence of ASCII character representations will appear as a sequence of bytes starting at a given memory address. The same special case of the quotation mark is present here: all quotation marks inside a character string must be preceded by a backslash, which is used solely to identify the quotation mark, and which is not itself included in the string.

## 5.3 Labelling program code

As we have mentioned before, a program written for a platform at Level 3 or above is translated into a bit-string, granulated into machine words, containing Level 2 instructions. An assembly language program due to the conceptual proximity of the language to that of the Level 2 machine does not hide the placement of code and data from the programmer, and it is possible to make direct reference to the address at which (the first byte of) each instruction in the program will be stored.

### 5.3.1 Introducing labels

We may *label* any source code line with a name of some kind, and later use that label to refer to the starting address of the instruction on that line. A *label* is a string of alphanumeric characters that symbolically represents the start address of a bit-string corresponding to a line of code in an assembly program. Within the program it serves as a *name* for the memory location at which the translated version of the labelled line will appear. In this sense a label is always a *symbolic address*.

There are three labels in our example program: `x`, `y`, and `bill`. Each of them is just a name for a memory cell, and so could be replaced by the address of that cell. However, to do this we would need to know exactly how many bytes were taken up by each instruction in the program, and if we chose to insert or delete lines, or re-locate instructions, we would have to re-calculate each address and make sure it is used correctly everywhere it appears in the program. So what we do is name the locations, and leave the compiler to work out the memory addresses for which our labels are names. Labels are placed before any instruction on a line or even on an empty line (save for possible comments).

How do symbolic names become associated with addresses? The compiler reads the program line by line and places the equivalent Level 2 instructions in an *image* of memory (i.e. a bit string the size of memory that will later be copied to real memory for program execution) starting at a certain position in it<sup>1</sup>. As the assembler places more and more instructions in the memory image, it also takes note of any labels. Every time it encounters a label it checks what address in the memory image it is about to fill in. That is the address that the label will be associated with, and it is that address which will be substituted for the label (you can think of it as substitution of a number literal for a name) when it is *used* in any instruction or pseudo-instruction. In this sense the assembly language programmer should appreciate that each line of her program has a *current* memory address, i.e. the address in the memory image the compiler will be placing the contents defined on that line at. Notice, that the assembler first places the whole program in the memory image and only then starts to substitute numerical addresses for any labels *occurring* as operands in instructions. This requires as many as two readings of the program.

It also means that the label can be mentioned as an operand *first* and placed after it; the assembler will still be in a position to substitute a numerical address even though it has yet to encounter the labelling occurrence of the label in the second reading, because it *already* encountered it in the first.

From the practical point of view, labels should be sufficiently mnemonic and unambiguous. In the following program the programmer has done a much better job of choosing her labels:

```
asect 0
ldi    r0, boys
ld    r0, r1
ldi    r2, girls
```

---

<sup>1</sup>In fact, the section pseudo-instructions, `asect` and `rsect`, are there to specify which address in the memory image the process must start with

```

ld      r2, r3
add    r1, r3
ldi   r0, children
st     r0, r3
halt
boys: dc    10
girls: dc    25
children: ds    1
end

```

By looking at the program we can tell that the data in the location named by the label `boys` is the number of boys, and the data in the location named by the label `girls` is the number of girls, the label `children` names the location where the number of children will be stored.

However, it is still unclear what the program does, and what it is for, even if you understand the individual instructions. In fact this is a program to add together the number of girls and boys in a primary school class, and store the result back in memory, but how do we communicate this to a casual reader? Or even to the programmer who wrote it and put it to one side for a few months?

## 5.4 Making programs understandable

Programs define *how* a given platform will be used to solve a problem. The platform does not need to be given any information whatsoever about what the problem is, about what any inputs mean, or about what any outputs will be used for. All it needs is enough information to collect a set of input data items, and to manipulate them to obtain a set of output data items. Platforms at Levels 2 and below do not even need to know what type of data they will be manipulating since all kinds of data at low level are treated as bit strings.

Assembly language programming is genuinely low level: you, the programmer, control each small piece of computing machinery either directly, by including instructions in your program, or indirectly as a result of those instructions acting upon data. Almost nothing is automatic, and relatively simple processing may take many instructions. It is easy to become overwhelmed by small details and lose sight of the bigger picture. So assembly language programs are extremely hard to understand, and even harder to diagnose and correct, unless they are accompanied by explanations of what is being done and why.

Fortunately, human-readable languages (Level 3 and above) contain facilities for introducing symbols (such as labels), and for including explanations of what the platform is being commanded to do and why, in the form of *comments*. Higher-level languages (Level 4 and above) provide programmers with a user-extendable set of types and expressions that represent ‘bigger steps’. This makes programs easier to read, and comments less necessary, but they are still required.

### 5.4.1 Comments

The algorithmic steps in an assembly language program are so tiny that it is nearly impossible to ‘see’ where they lead. To alleviate the problem we include “high-level” comments that explain all but the most obvious lines in any program. A programmer’s comments should explain the purpose and/or the meaning of the line *in terms of the problem being solved*, rather than the meaning and effect of individual instructions in terms of platform resources (this is captured by the instruction itself, and seldom requires a comment).

The CdM-8 Platform 3<sup>1/2</sup> macro-assembly language allows a comment to be placed on each line, provided it is preceded by a hash character (#). When the assembler encounters a hash character on a line (unless it is placed between the quotation marks and is therefore part of a character/string literal) it simply ignores all characters from that point up to the end of the line. So a comment may only be placed at the *end* of a line. For example:

```
    asect 0
start: ldi      r0, boys
        ld       r0, r1
        ldi      r2, girls
        ld       r2, r3
        add     r1, r3
        ldi      r0, children
        st      r0, r3
        halt
#  Code done, data follows
boys:   dc      10    # The number of boys in the class
girls:   dc      25    # The number of girls in the class
children: ds      1     # Set aside 1 byte of memory to store the total
end
```

## 5.4.2 Layout

Another powerful tool for improving program readability is structuring. Quite often inexperienced programmers produce a stream of code which contains code for meaningful actions each taking several lines. For example, data may be fetched from memory, which involves loading an address in a register and then fetching data at that address; two lines, two machine instructions together represent a single meaningful action: fetching a value from a memory cell.

Those types of actions should be made more readable by separating them from one another with, for example, a blank line. Also useful is the use of delimiters presented as comments, for instance, #####... , \*\*\*\*\*..., etc., either as whole line delimiters, or even as boxes of hashes, vertical lines of hashes and asterisks, etc. How exactly the structure of a piece of code is revealed and emphasised is a matter of taste and judgement. What is important though, is for the programmer to be consistent: decide what level/granularity of structure to reveal, how to format it (taking care to always precede any structuring elements with a hash so that the assembler may ignore it), and what textual comments, if any, to supply in addition to the delimiting elements.

Another serious structuring device, which is often compulsory in higher-level languages (Level 4 and above) is *indentation*. Some of the code structures may *contain* others. For example, certain registers may hold important data but be also required as temporary storage for some other computation. In this case the registers have to be copied to memory, used, and then the former content restored by loading them with the previously copied data. Clearly, there are preliminary actions that happen before the computation (copying the content of the registers to memory) and after it (restoring the register content). The computation is thus *inside* a code environment which has a beginning and an end. To help the reader see it, the code for the beginning and end actions is aligned horizontally, and the inner computation is indented, i.e. shifter to the right by a certain number of spaces and aligned to that position as well:

```
ldi r0,storage_area          # save
st r0,r1                      #           r1

move r2,r1          # exchange r2
move r3,r2          #           and r3
move r1,r3          #           using r1 as a scratch pad

ldi r0,storage_area          # restore
ld r0,r1              #           r1 from storage area
```

Even when no code environments are present indentation and horizontal alignment are useful for improving the visibility of labels and comments. Here is an example:

```
#####
#                                     #
#      PROGRAM: ClassTotal          #
#      A program to add together the number    #
#          of boys and girls in a primary school   #
#                                     #
#####
asect 0
start: ldi     r0, boys
       ld      r0, r1      # r1 = the number of boys
       ldi     r2, girls
       ld      r2, r3      # r3 = the number of girls
       add    r1, r3      # r3 = no of boys + no of girls
       ldi     r0, children
       st      r0, r3      # total stored to cell 'children'
       halt
# Code done, data follows
boys:   dc      10    # the number of boys in the class
girls:  dc      25    # the number of girls in the class
children: ds      1     # memory cell for the total
end
```

### **5.4.3 Use of ‘white space’**

It is possible to insert blank lines anywhere in the source file without changing the program; judicious use of this often helps to improve readability by adding *vertical white space*.

Also, whilst horizontal alignment conveys no information to the assembler, it is advisable to use it to improve readability, so in any place where a single space or TAB character is allowed it is permissible to use multiple spaces or TABs, or a combination of the two (*horizontal white space*). Also, a line that contains only white space or a comment is treated by the assembler as if it were a blank line.

“Whole line” comments and blank lines should be used to convey “headings” and “titles” in a longer program, to identify where different parts start and end, and to explain what each part of the program is for and how it may participate in the overall solution. By using a combination of meaningful label names, narrative comments, and good layout a programmer creates source code that is well presented and well documented, which can be read and understood by the author and other programmers alike.

**Do not underestimate how hard it is to understand assembly language programs: even those you wrote yourself. Good layout and commenting is essential.**

## 5.5 Placing code and data in memory

All programs operate on and produce data items. The data values may be included with the program, or may be obtained via peripheral devices. Whilst the full CdM-8 system can access peripheral devices, at the moment we are working with a basic system that can not. So the data we need the program to process must form part of the memory image that we are going to load into the machine, and we must also allocate memory space for the results that we intend to produce. In other words we must choose the memory locations that we are going to use for initial and result data, define the initial values that will be stored in those locations (i.e. define what values they will contain at the moment that the program starts) and make sure that none of the locations where we intend to store results are being used for any other purpose.

This last point is very important, since the accidental over-writing of memory (e.g., when result data is stored to a location that holds a machine instruction) can cause errors that are extremely hard to detect and eliminate: almost every bit-string of length 8 is a legitimate machine code instruction of one kind or another, so when an instruction is overwritten with an arbitrary byte of data, we end up with a new program that will perform a completely different task. Whenever the PC points at this memory location the platform will attempt to execute whatever ‘instruction’ is stored there, in particular this may turn out to be identified as a memory-write instruction, or a control transfer instruction, resulting in massive unintended corruption of memory and potentially obliteration of all traces of the original program. If the program is run to termination (to a halt instruction), it may even fail to terminate (if that instruction becomes overwritten) or if it does, the programmer may find that all traces of the error that led to this have already been destroyed by subsequent unintended overwriting.

### 5.5.1 Placement of machine instructions and data in memory

As we mentioned before, the goal of the compiler is to create a memory image that contains the initial content of each memory cell. We started looking into this process already in section 5.3.1 to better understand how labels become associated with addresses. During the second reading, the compiler fills consecutive memory cells of the image with appropriate values as it reads and compiles the program. It starts at a certain address, specified by a pseudo-instruction, and continues until either the whole program has been compiled or a new starting address is specified, producing a contiguous *section* of machine code. The resulting memory image is a set of non-overlapping sections, between which the content of memory is left unspecified by the currently compiled program. Other, separately compiled parts of the program may occupy those gaps or they may be used as data storage. For the time being we will assume that the whole program is compiled at once and that every memory location that it uses is specified in it.

As the assembler reads the assembly program line by line and produces machine code according to it, it also maintains the “current address” i.e. the address at which the machine code (instructions or data) is currently being stored in the memory image. The compiler’s current address is somewhat similar to the Level 2 machine’s Program Counter in that the assembler always stores compiled code at the current address, just as the Level 2 machine always fetches instructions at the address contained in its PC. Also just as the PC is advanced after use, so is the current address (the latter to prevent placing further code in already used cells).

Three pseudo-instructions are involved in this process.<sup>2</sup>

**asect** is used to specify the **a**bsolute (i.e. numerical) start address of a **s**ection. Technically it sets a (new) current address.

**dc** is used to **d**efine a **c**onstant, or an initial data value, that will be placed at the current address in the memory image. More than one constant may be defined, forming a sequence of constants, which are placed, byte-by-byte, with an appropriate adjustment of the assembler’s current address.

**ds** The third is used to **d**eclare **s**pace, i.e. to ‘reserve’ an area of memory (typically to be used for storing results or temporary data), so that the compiler does not fill it up with instructions or other data when it is creating a memory image. Technically, it causes the compiler to advance the current address by the amount of memory being reserved.

Of course all assembly language instructions (as opposed to pseudo-instructions) are compiled into a sequence of one or more bytes, which are placed, byte by byte, in the same way as data by the pseudo-instruction **dc**.

<sup>2</sup>There is actually a fourth, but we will not meet it until much later.

**Absolute Section.** This pseudo-instruction has the following format:

```
asect n
```

Here  $n$  may be specified using a decimal number in the range 0..255, or a hexadecimal or binary number. It defines the start address of the section of code that follows. For example,

```
asect 0x1C
```

tells the compiler that the instruction on the next line will be placed at address 0x1C (which is 28 in decimal), and subsequent instructions will be put in subsequent memory cells. A program may contain any number of `asects`, though it is important that there are sufficient memory cells between the starts of consecutive absolute sections to fit in the code that is expected to go there. Between `asects`, the placement of bytes containing the machine code proceeds contiguously<sup>3</sup> in the ascending order of addresses.

So if a section of a program that translates to eleven bytes of code follows the instruction `asect 0x12` the code will be placed in the memory image at (hex) addresses 12,13,14,15,16,17,18,19,1A,1B and 1C, and the next `asect` (if there is one) must have a start address of 0x1C or higher, or an address lower than 12 so as to avoid an overlap. We used an `asect` instruction in our earlier example program to tell the compiler that the start address of the program is 0.

**Define Constant.** A constant is an initial data value for use in the program. It is specified at the time the program is written rather than at run-time. The ‘define constant’ (`dc`) pseudo-instruction tells the assembler to allocate space for data and fill that space with binary representations of the literal values that follow:

```
dc item1, item2,...
```

The `dc` instruction tells the compiler that what follows is a comma-separated list of values, each of which is to be stored in memory, starting at the current address. There can be any number of constants in the list.

Each item can be a number (in one of the formats described earlier), an ASCII character, written as "c" between double quotes or an address constant (a label), which we will discuss later. The items are placed in consecutive locations starting from the current placement address: the one specified in the `asect`, if the preceding instruction is an `asect` or the first unallocated address after the previous instruction.

Note that `dc` defines the *initial content* of some memory locations, so the word “constant” should not be misunderstood to mean that those locations will *always* have the declared values: the program can, and in many cases will, change the content of the cells mentioned in a `dc`.

Here is an example. This program:

```
asect 0x10
dc -3, 0b00000101, "a"
end
```

defines the following initial memory content:

address	data
0x10	0xfd
0x11	0x05
0x12	0x61

An item can be a character string consisting of more than one character, in which case the characters are placed consecutively in memory as if they were separate items, for example, this code snippet

```
asect 0x10
dc "abc"
end
```

<sup>3</sup> A contiguous sequence of memory cells is a sequence in which consecutive cells have consecutive addresses.

defines the following memory content:

address	data
0x10	0x61
0x11	0x62
0x12	0x63

**Declare Space.** Quite often, especially when we need space for data that we are going to compute, the initial content of that space in memory is not important. One could still use `dc` for allocating such areas of memory, but it would look rather confusing in a program, albeit not invalid, if arbitrary content were declared for the sake of the space that it would then occupy.

A rather more direct way of reserving space for results is via the `ds` pseudo-instruction:

```
ds    n
```

Here  $n$  is a nonnegative number (in whatever form) that defines how many memory cells (bytes) will be set aside for data. In reality, the compiler will always fill these bytes with the initial value 0, but a programmer should never rely on this. If 0 is what you need a sequence of  $n$  bytes to be filled in with, it is advisable to make this intention clear by writing a `dc` instruction with a comma-separated list of  $n$  zeros.

Example:

```
asect 0x10
ds    5
...
```

The assembler will reserve all locations in the range `0x10–0x14` for use as data and will not place any instructions or constants in those cells. The current address on the line following the above fragment will be `0x19`.

### 5.5.2 Labelling data locations

It is typically useful to label constant definitions and space declarations so that they can be referred to later using names rather than numeric addresses:

```
label: dc item1, item2,...
```

```
label: ds n
```

The start address of the first constant in the list will be associated everywhere in the program with the symbolic *label* if it is present. Taking the example given before

```
asect 0x10
foo: dc -3, 0b00000101, "a"
bar: ds 2
end
```

The label `foo` is associated with the address `0x10`, and the label `bar` is associated with the address `0x13`.

Symbolic addressing is very useful. It means we can refer to a precise location in memory without having to know its numerical address. It also means that changes to the declaration of `asects`, or to the amount of memory space allocated before the line that has been labelled, can be ignored by the programmer, because the compiler will work out the correct address to attach to the symbol. Finally symbolic addresses convey the intention of what they label: for example: the name `start` may mark the start of the program and the name `size` may label a memory cell that holds the size of an array.

### 5.5.3 Referring to labels

Since a label is actually just a name for a memory address, and since memory addresses are just 8-bit binary numbers, we can use a label to refer to an address anywhere we can use a number literal<sup>4</sup>. So a dc item list can contain labels just as readily as number literals, character literals and string literals. For example,

```
    asect 0x10
    dc    foo, bar
foo:   dc    13, "ba"
bar:   dc    17, "de"
      end
```

has the following memory footprint:

address	data
0x10	0x12
0x11	0x15
0x12	0x0D
0x13	0x62
0x14	0x61
0x15	0x11
0x16	0x64
0x17	0x65

### 5.5.4 Relative addresses and offsets

Sometimes it is convenient to refer to an item in a program by stating the distance in bytes between it and the address associated with a label. In other words we give the address of one thing *relative* to another.

In such a case the address given by the label is referred to as the *base* address, the distance in bytes is referred to as the *offset*, and the address of the item of interest is given *relative to the label*, using *label + n*, or *label - n* where *n* is a number signifying the offset.

This comes especially handy when the label marks the start address of a record in memory, and the offsets are those of individual fields within the record. For instance, in the above example, one may see **foo** and **bar** as data structures of record type **SN**, defined as follows:

```
record: SN
```

field	type	offset (bytes)
N	int	0
s	size 2 array of char	1

Then a dc that contains two pointers to field **s** of **foo** and **bar** can be placed thus:

```
    asect 0x10
    dc    foo+1, bar+1
foo:   dc    13,"ba"
bar:   dc    17,"de"
      end
```

---

<sup>4</sup>except, of course, the **asect** pseudo-instruction, since a label labels an already *allocated* memory cell while **asect** must declare a fresh address

This would result in the following content:

address	data
0x10	0x13
0x11	0x16
0x12	0x13
0x13	0x62
0x14	0x61
0x15	0x17
0x16	0x64
0x17	0x65

Generally it is legal to use any label with a positive or negative offset defined by a number (in whatever format): `foo+3`, `bar-2`, etc. Normally this only makes sense for very small offsets, which are usually defined by a single decimal digit. Larger offsets are frequently unsafe as it is easy to miscalculate the distance between two addresses by looking at an assembly program.

In almost all cases it is advisable to define additional labels by placing them at the `dc` instructions of interest; if one wishes to refer to the middle of a data structure, it is best to split it up and make the item of interest the opening item of a separate `dc`.

The `dc` instruction is potent enough to deploy all kinds of data in memory, including linked data structures, in a manner that makes the programmer's intention quite clear. For example the linked list on page 84 can be programmed as follows:

address in hex	content in decimal (and hex)	# The code fragment below declares a linked list of numbers # Each item is a record of the form (next, payload), where # 'next' is a pointer to the next item # and 'payload' is the number that's in the list
...	...	
0A	15 (0F)	
0B	7	
0C	...	
0D	10 (0A)	asect 0x0D item1:
0E	5	dc item2, 5
0F	20 (14)	asect 0x0A
10	-3	item2: dc item3, 7
11	...	asect 0x0F
12	...	item3: dc item4, -3
13	...	asect 0x14
14	0 (00)	item4: dc 0x00, 8
15	8	end
...	...	

Here it is quite clear that the first member `item1` points to the second one, `item2` in its `next` field, that the second member `item2` similarly refers to the third one `item3`, etc. The clarity is achieved by using symbolic addresses (labels) `item1`, `item2`, etc. instead of numerical addresses, and by giving the field names in comment lines. In fact all the `asects` could be dropped (perhaps with the exception of the first one) without changing the meaning of the list, its payloads or their position in the sequence, even though the list is scattered in memory.

## 5.6 Loading and storing data

CdM-8 has a *load and store* architecture. This means that the only operations the processor can perform with memory are *loads* (copy from memory to register) and *stores* (copy from register to memory). All other operations are internal to the CPU, and therefore involve manipulation of registers.

In order to process data on the platform we require the ability to copy byte values between registers and memory. The Load-and-Store group of operations are available to help us do that. There are three of them, and none of them affects any of the CVZN flags in any way.

### 5.6.1 Load Immediate

This is key for memory-register exchange. Indeed, there is a chicken-and-egg situation here, characteristic of any load/store architecture, of which CdM-8 on which our Platform 3<sup>1/2</sup> is based, is a typical example.

In order to fetch data from memory one needs the address of the data to be available. However, the address is also data, and in order to get that from memory to a register one needs to load the address of the location where that address is held first, etc. To break the vicious circle most modern microprocessors include an operation that loads a register *with a data item contained directly inside the instruction*. Such a data item does not require an address, and is usually called an *immediate operand*.<sup>5</sup>

CdM-8 Platform 2 provides a Load Immediate instruction, which Platform 3 and Platform 3<sup>1/2</sup> adopt:

```
ldi      rn, x
```

This is a reflection of a true hardware instruction, and it is worth looking at the form it takes.

The instruction is made up of two parts: *named operation* and a comma-separated list of *operands*. The name is in the form of a *mnemonic*.

- Here the mnemonic is **ldi** (which stands for **LoaD Immediate**)
- The operand list is

**rn, x**

where **rn** is one of the 4 GP registers (register *n*) and **x** is a byte-sized immediate constant (a number, a character or an address - in the form of label or label and offset. Essentially the second operand of **ldi** has the same options as an *item* on a **dc** list except one: it is an *immediate* operand and therefore has a limited size of 1 byte, so it is not possible to load a string literal in the register.)

Taken together, the operation and operand list may be translated into a single Level 2 instruction. Other Level 3 instructions are specified in the same way, though some require no operands. CdM-8 Platform 3<sup>1/2</sup> also provides some additional operations (some of which require operands) that are translated into two or more Level 2 instructions, plus a set of additional pseudo-instructions.

Let us look at an example.

```
asect  0x00
ldi    r0, 3
ldi    r1, "c"
ldi    r2, a
ldi    r3, a+1
halt

asect  0x20
a:     dc    3, 7
end
```

---

<sup>5</sup> the immediate data value is included in the memory image of the instruction being executed.

Remember that an assembly language program must include the pseudo-instruction `end` on the last line of the source code, and that the execution of the program will only stop when the platform reaches a `halt` machine instruction.

The above is, in fact, a completely valid and a fully functional assembly language program, which can be assembled, linked and run on Platform 3<sup>4/2</sup>. Except .... it isn't commented.

Upon termination the content of the registers will be as follows:

register	content
r0	0x03
r1	0x63
r2	0x20
r3	0x21

The PS register and the SP are left unchanged, and the PC will contain ... what value? Run it and see.

### 5.6.2 Load

Now that we are able to get immediate data into registers, we are equipped to study the Load operation. It has a very simple form and meaning:

`ld rn,rm`

where `rn` and `rm` are two (not necessarily different) GP registers. What this does is use the *content of the first* register as a *pointer* to a memory location, from which it will copy a byte of data into the *second* register. When we say that one value points to another, we mean that the former is the address of the latter. We have seen pointing before when we studied pointers in an earlier chapter.

To summarise, the first register points to a memory cell; the content of that cell is copied from memory into the second register. Examples:

```

        asect    0x00
        ldi      r2, a
        ld       r2, r0

        ldi      r3, a+1
        ld       r3, r1
        halt

        asect    0x20
a:      dc      3,7
        end
    
```

Upon termination the content of the GP registers will be as follows:

register	content
r0	0x03
r1	0x07
r2	0x20
r3	0x21

Once again, the PS register and the SP are left unchanged, but what value will be held in the PC?

If all we were interested in doing was to get the contents of the two bytes at locations `a` and `a+1` into two registers, we could have used just the two:

```

asect 0x00
ldi    r0, a
ld    r0, r0
ldi    r1, a+1
ld    r1, r1
halt

asect 0x20
a:    dc    3,7
end

```

In this case we first loaded an address into `r0`, then copied the data from that address into `r0` (replacing the address). We then loaded a second address into `r1`, then copied the data from *that* address into `r1`. The results stored in `r0`, `r1` remain the same as in the previous example, but no other registers are changed, and the two addresses are no longer in registers.

This is one illustration of the fact that there is a certain *internal order* to the execution of a single instruction: registers are read first and only then overwritten with new values. In the above example `ld r0,r0` first reads `r0` to obtain the memory address, then reads the memory cell at that address, and only after that overwrites the content of `r0` with the new value — all perfectly valid and manageable by the platform.

### 5.6.3 Store

The last operation of this group is the *store* operation whose effect is the ‘opposite’ to *load*. Once again we specify two GP registers, with the first containing an address, but this time the contents of the second register are copied into memory at the location pointed to by the first register,<sup>6</sup> as follows:

<code>st rn,rm</code>
-----------------------

Here the address is provided in `rn` and the data in `rm`. The effect of the instruction is that the memory cell pointed to by `rn` changes its value to match the contents of `rm`. For example:

```

asect 0x00
ldi    r0, a
ld    r0, r0

ldi    r1, a+1
st    r1, r0
halt

asect 0x20
a:    dc    3,7
end

```

The memory contents before execution of the above fragment:

address	data
0x20	0x03
0x21	0x07

and after that:

address	data
0x20	0x03
0x21	0x03

---

<sup>6</sup> The same register may be used for both operands, just as it may for `ld`

## 5.7 The Arithmetic-Logic Unit (ALU)

Platforms are a bit like cars: they have engines and steering, only of a somewhat different kind. The engine of a Level 2 platform is called the Arithmetic-Logic Unit, which we do not see in Platform 3<sup>1/2</sup>, since we do not control it directly in the program<sup>7</sup>.

The ALU has the ability to perform arithmetic and logic operations, some of which operate on a single GP register, and others of which operate on a pair of GP registers at a time. The registers specified in the instruction are called the *operands* of the operation. For example, in the instruction `inc r0` the operation is `inc` and the operand is `r0`, and in the instruction `add r0, r1` the operation is `add` and the operands are `r0` and `r1`.

Additionally, ALU operations affect the CVZN flags that make up the PS register mentioned earlier. For each operation the values of *some* flags will depend on the result of performing that operation on its operand(s), and the values of others will be cleared.

**Summary** Each ALU instruction is made up of an operation and one or two operands. ALU instruction operands are always GP registers.

- ALU operations fall into three categories
  1. Arithmetic operations, which treat the contents of registers as numbers
  2. Logic operations, which treat the contents of registers as sets
  3. Data movement operations, which treat the contents of registers as bit-strings
- Any GP register can be used as an operand.
- When two operands are required (e.g., for addition), they do not have to be different registers.
- Some operations update a register that was given as operand
  - Where there is a single operand that register may be updated.
  - Where there are two operands, the *second* register may be updated, but never the first one.
- Every ALU instruction writes a bit-string to the PS register

Registers *may* be updated, because there are ALU operations whose only objective is to update the CVZN flags. The majority of the ALU instructions in fact do update a register as well.

Next we will consider each of the three groups of ALU operations in turn. For each operation we will give a short summary (synopsis) of its effect on the registers and indicate how it affects the flags. Then we will give some examples.

### 5.7.1 Arithmetic operations

Figure 5.1 summarises the arithmetic operations.<sup>8</sup> The trickiest part is probably the flags.

The difference between `add` and `addc` is that the former is ordinary 8-bit binary addition and the latter adds the value of the carry flag to the result (as a carry-in to bit 0) before the addition commences. This operation provides support for bit-sliced addition. Another subtlety is the difference between `sub`, which is normal subtraction, and `cmp`, which is also subtraction but one that does not update the second operand.

`cmp` is a *comparison* operation, and is used solely to set the flags. The program may, at a certain point, need to compare two numbers loaded into registers to find out whether they are equal, one is greater than the other,etc, without changing the contents of either register. This is achieved by `cmp`. A control operation

<sup>7</sup> When we look into the Level 2 platform that supports CdM-8 Platform 3<sup>1/2</sup> - as we will in the last chapters of this book - it is clearly visible and its workings are totally comprehensible.

<sup>8</sup> In this table the symbol `:=` indicates *assignment*, so the statement `x := x + y` means “calculate `x+y` and load the result in `x`”

Instruction	Synopsis	Formula	Flags Affected
add <i>rn,rm</i>	add <i>rn</i> to <i>rm</i>	$rm := rn + rm$	C,V by operation; Z,N by result
addc <i>rn,rm</i>	add with carry <i>rn</i> to <i>rm</i>	$rm := rn + rm + C$	"
sub <i>rn,rm</i>	subtract <i>rm</i> from <i>rn</i> , copy to <i>rm</i>	$rm := rn - rm$	"
cmp <i>rn,rm</i>	as above, except <i>rm</i> left unchanged	$rn - rm$	"
neg <i>rn</i>	2's comp change of sign	$rn := -rn$	"
inc <i>rn</i>	add 1 to <i>rn</i>	$rn := rn+1$	"
inc <i>rn</i>	subtract 1 from <i>rn</i>	$rn := rn-1$	"
clr <i>rn</i>	clear <i>rn</i>	$rn := 0$	C=1,V=0,Z=1,N=0
tst <i>rn</i>	test <i>rn</i>	$rn := rn$	C=0,V=0; Z,N by result

Figure 5.1: Arithmetic operations

may then be used to choose which of two execution paths the program will follow, depending on the values of the flags. Control instructions are discussed in section 5.8 .

It is important to remember that the flags are changed by ALU operations, whether or not they are going to be used for control. The operation `cmp` is available for those situations when *no other* useful work is to be done apart from flag setting.

A single-operand version of `cmp` is available as well: the `tst` (test) operation. This operation simply examines its operand register to find out whether it contains zero, a negative or a positive number, and sets the flags accordingly.

Aside: It might be a little confusing to see the flags Z and N changed by `cmp` to indicate “the operands are equal/not equal or less/not less”, respectively when it is used to compare two registers. However, you will probably be pleased to learn that the flags themselves are rarely referred to by name; there is a large and redundant set of “conditions” such as *greater-than* and *plus*, which are made available to the Platform 3½ programmer, and which we will encounter later on.

Example program:

```
# Exercising the ALU
    asect 0x00
    ldi    r0, data
    ld     r0, r0          # r0=10

    ldi    r1, data+1
    ld     r1, r1          # r1=-3

    ldi    r2, data+2
    ld     r2, r2          # r2=-128

    add   r0, r1          # r1=10-3=7;      C=1, V=0, Z=0, N=0
    sub   r0, r1          # r1=10-7=3;      C=1, V=0, Z=0, N=0
    addc  r0, r1          # r1=14;          C=0, V=0, Z=0, N=0
    clr   r3              # r3=0;           C=1, V=0, Z=1, N=0
    dec   r3              # r3=-1;          C=0, V=0, Z=0, N=1
    add   r2, r3          # r3=127;         C=1, V=1, Z=0, N=0
    cmp   r0, r2          # no change;       C=0, V=1, Z=0, N=1
    tst   r2              # no change;       C=0, V=0, Z=0, N=1
    neg   r2              # no change;       C=0, V=1, Z=0, N=1
    inc   r2              # r2=-127;        C=0, V=0, Z=0, N=1
    halt

    asect 0x20
data: dc 10,-3,-128
end
```

Instruction	Synopsis	Formula
and <i>rn,rm</i>	Load <i>rm</i> with the bit-wise <b>and</b> of <i>rn, rm</i>	$rm := rn \wedge rm$
or <i>rn,rm</i>	Load <i>rm</i> with the bit-wise <b>or</b> of <i>rn, rm</i>	$rm := rn \vee rm$
xor <i>rn,rm</i>	Load <i>rm</i> with the bit-wise <b>xor</b> of <i>rn, rm</i>	$rm := rn \oplus rm$
not <i>rn</i>	Load <i>rn</i> with the bit-wise <b>not</b> of <i>rn</i>	$rn := rn'$

Figure 5.2: Logic operations

### 5.7.2 Logic operations

The logic operations are summarised in figure 5.2. They are straightforward bitwise applications of Boolean logic connectives to the corresponding bits of the operands. The flags V and C are cleared by all logic operations, since no arithmetic of any kind is performed, and the other two flags, N and Z are, in all four cases, determined based on the result interpreted as a signed number. Of course the operations themselves do not make much sense if the operands are interpreted as numbers, but the flags N and Z are still meaningful and useful even for the result interpreted as a bit string: N indicates the content of bit 7 and Z whether or not all bits of the results are zero.

The operations **and**, **or** and **xor** each take two registers as operands. In each case the logical connective is applied to corresponding bits in the two registers. For example, for **and r0,r2**, bit 3 of the result is the conjunction (logical “and”) of bit 3 of **r0** and bit 3 of **r2**. The operation **not** applies logical negation to each bit of its single operand, replacing 1s with 0s and 0s with 1s.

Interestingly, the purpose of logic operations is not at all to support automated reasoning or deduction! Far from it, logic operations are merely a way of achieving *selective manipulation of bits in a bit-string*.

We may use a logic operation to change any particular bit(s) in an 8-bit string to 1, to change selected bits to 0, to flip selected bits (make them 0 if they are 1 and make them 1 if they are 0) or to flip all bits in a string. These actions are achieved by the operations **or**, **and**, **xor** and **not**, respectively.

To set bits *x, y* and *z* of register **rm** to 1, we create a bit-string which has bits *x, y* and *z* as 1’s, and the rest as 0’s, load it into register **rn** and then **or** it into **rm**. A bit-string (or number) specially created for the purposes of manipulating bits of another bit-string is commonly referred to as a *mask*. Here is an example:

```
# setting bits b0 and b3 to 1
    run      code
    asect    0x20

data:   dc      0x35          # some number to try

code:   ldi      r0, data
        ld      r0, r0          # r0 = 0x35 = 0b00110101

# load the mask into r1
        ldi      r1, 0b00001001 # bits b0 and b3 of r1 are set, the rest are cleared

# set selected bits of r0, as specified by the mask
        or      r1, r0          # sets b0 and b3 in r0 to 1
                                # after which r0 contains 0b00111101 = 0x3d
        halt
end
```

If we wished to *clear* specific bits in an 8-bit string, we would use the operation **and** with which we would require a mask that marked the bits to be cleared with 0’s and the bits to be left unchanged with 1’s. It is opposite to the kind of mask that we used with the **or** above.

Here is the same example, but now the bits are cleared rather than set:

```

# clearing bits b0 and b3

    ldi      r0, data
    ld       r0, r0          # r0 = 0x35 = 0b00110101

# load the mask
    ldi      r1, 0b11110110 # bits b0 and b3 of r1 are cleared, the rest are set

# clear selected bits of r0, as specified by the mask
    and      r1, r0          # clears b0 and b3 in r0
                                # after which r0 contains 0b00110100 = 0x34
    halt
data:   dc      0x35          # some number to try
end

```

Flipping selected bits requires the same kind of mask as was used for setting them. It marks the bits to be flipped with 1's and the rest with 0's:

```

# flipping bits b0 and b3
    ldi      r0, data
    ld       r0, r0          # r0 = 0x35 = 0b00110101

# load the mask
    ldi      r1, 0b00001001 # bits b0 and b3 of r1 are set, the rest are cleared

# flip selected bits of r0, as specified by the mask
    xor      r1, r0          # flips b0 and b3 of r0
                                # after which r0 contains 0b00111100 = 0x3c
    halt
data:   dc      0x35          # some number to try
end

```

Finally, flipping *all* bits requires no mask. This is achieved by the `not` operation thus:

```

# flipping all bits
    ldi      r0, data
    ld       r0, r0          # r0 = 0x35 = 0b00110101

# flip all bits of r0
    not      r0          # after which r0 = 0b11001010 = 0xca

    halt
data:   dc      0x35          # some number to try
end

```

**Testing bit strings.** Just like numbers may need to be tested to determine their sign or equality to zero, bit strings often require a test that certain bit positions in a string are all 1's, all 0's or a mixture. The technique for testing these properties is similar to the patterns shown above, with some small differences:

**all 0's** This is achieved with a 1-mask, just like setting bits. Load a 1-mask in a register, say `r0`. For example, if we are testing bits 0,1, and 5, load mask 0b00100011 in `r0`. Now we can test, for example, `r1` by issuing `and r1,r0`. If the bits in positions 0,1, and 5 in `r1` are 0's then the `and` operation will yield a result of 0b00000000. Indeed for every 1 in the mask we have 0 in the corresponding bit of `r1` and the conjunction yields 0. All other bits in the mask are already 0's, ensuring a 0 result of the conjunction. Of course if at least one of the corresponding bits of `r1` were set, the resulting bit would be 1. All that remains is to examine the flag Z; if Z is set, the test succeeds, if not it fails. Notice that the mask is destroyed as a side-effect of the test, but, as we would expect, the value being tested is not changed.

**all 1's** Interestingly, this test cannot be done in one instruction, it requires two. First load a 0-mask, where the bits being tested are marked with 0's and the rest of the bits are 1's. Continuing with our example, we load the mask 0b11011100 for bits 0,1, and 5 in `r0`. Now the trick. Issue `or r1,r0`. If bits 0,1, and 5 are *all 1's* in `r1`, then (and only then) the result will be 0b11111111. Recall that this bit string is the representation in 8-bit 2's complement of the quantity  $-1$ . Now issue `inc r0` and examine the flag Z; if Z is set the test succeeds, if not it fails. Notice that the mask is destroyed as a side-effect of the test, but, as we would expect, the value being tested is not changed.

**a mixture** It is easy to see that if both of the above tests fail, the only third option is that the bits in the positions of interest are a mixture of 1's and 0's.

**Bit-strings as sets.** Bit-strings representing sets and operations on them were discussed in section [3.10](#) Recalling the dense set representation we can see that

- the operation `and` computes the intersection of two size-8 sets
- the operation `or` computes their union
- the operation `xor` computes their symmetric difference
- and the operation `not` computes the complement of a single set

Using small sets and set operations (available as logic operations of the platform) is quite common. In fact the platform itself uses a 4-member *set* of flags CVZN, which corresponds to various conditions arrived at after an ALU operation. The platform performs what could be seen as a set operation by examining the flags as it makes decisions that affect the next executable instruction. How exactly this happens is explained in the next section.

Instruction	Synopsis	Formula	Flags Affected
shra rn	arithmetic shift right rn	$rn := rn/2$	$C=(rn \text{ bit } 0)$ before op $V=0; N,Z$ by result
shla rn	arithmetic shift left rn	$rn := rn+rn$	same as add
shr rn	sliced shift right rn	$rn := rn/2$ $rn \text{ bit } 7:=(C \text{ before op})$	$C=(rn \text{ bit } 0)$ before op $V=0; Z,N$ by result
shl rn	sliced shift left rn	$rn := rn+rn+(C \text{ before op})$	same as addc
rol rn	rotate left rn	$rn := rn+rn+(rn \text{ bit } 7)$	$V=0;$ the rest same as addc
move rn,rm	move rn to rm	$rm := rn$	$V=C=0;$ Z,N by result

Figure 5.3: Data movement operations

### 5.7.3 Data movement operations

Those operations are displayed in figure 5.3. The first five operations are various *shifts* and the last one copies data from one register to another. The term “movement” reflects the facts that the effect of a shift operation is as if the bit-string had moved inside the register, with end bits falling out or slotting in. In fact what moves is the *pattern* of bits in the bit-string, not the bit-string itself, since it is not physical and is unable to change its location inside the machine.

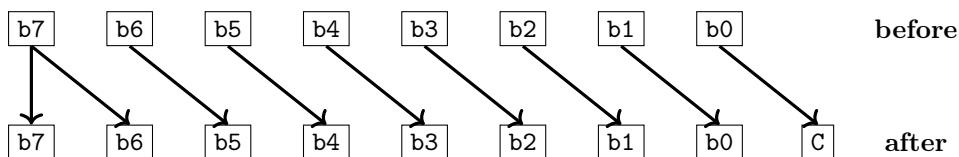
**Shift operations** cause the bit-string in a register to ‘shift’ one position to the left or right. Thus, a shift right causes the contents of bit-5 to ‘move down’ to occupy bit-4, the contents of bit-4 to ‘move down’ to occupy bit-3,etc. These operations differ in the direction of the shift, in what happens to the bits at either end of the bit-string held in the register (b0 and b7), and in the effect they have on the flags in the PS register.

**A reminder about bit numbering.** By convention, the bits in an  $n$ -bit string are numbered as follows:

- Individual bits are numbered from 0 to  $n - 1$ . In CdM-8 they are numbered from 0 to 7.
- Bits are numbered *from right to left*, so
  - Bit 0 is the *right-most* bit in the string
  - Bit  $n - 1$  is the *left-most* bit in the string
- Bit 0 (or b0) is the *least* significant bit.
- Bit  $n - 1$  (b7 in the case of CdM-8) is the *most* significant bit.
- In a signed number, the left-most bit ( $n - 1$ ) is used as the sign bit.

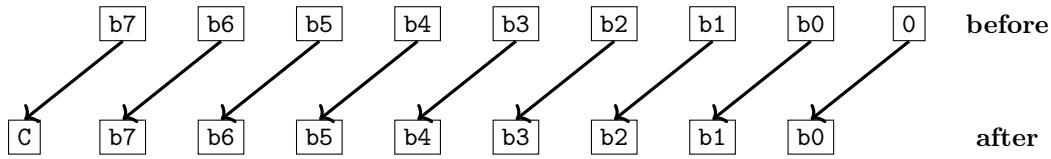
It is important to remember these conventions when looking at shift operations.

**Arithmetic shift right.** The operation **shra** treats its operand as a signed number:



The effect of the arithmetic shift right is identical to dividing by 2 with the remainder shifted into C. For example, the number 0b10101010 represents the quantity  $-86$  in 8-bit 2’s compliment. After the arithmetic shift right the value will become 0b11010101, which is the expected original quantity halved:  $-43$ . The carry flag C will be 0. If, otherwise, we started off with the value 0b00000101 , or the quantity 5, then after the shift it would become 0b00000010 or 2 with the C flag set to 1.

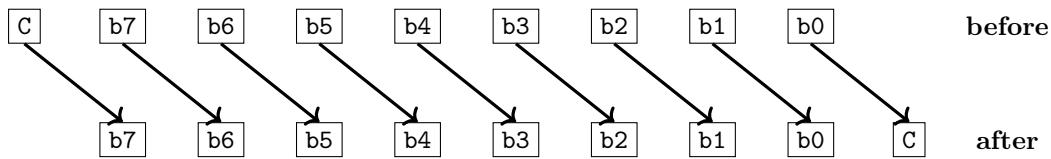
**Arithmetic shift left.** This one is the ‘opposite’ to the `shra` operation. It effectively multiplies the operand by a factor of 2.



If the operand is negative and the result positive or vice versa, the flag `V` will be 1 (indicating that the result is too large for an 8-bit two’s complement representation), otherwise 0. Examples:

value before	value after	flag values
0b00000010	0b00000100	C=0, V=0, Z=0, N=0
0b00000110	0b00001100	C=0, V=0, Z=0, N=0
0b11111110	0b11111100	C=1, V=0, Z=0, N=1
0b10000000	0b00000000	C=1, V=1, Z=1, N=0

**Sliced shift right.** The operation `shr`, treats the operand as an 8-bit slice of a larger number which needs to be shifted to the right:



It is included for a similar reason to `addc`. Recall from section 3.5.5 how bit-sliced addition may be done using linked carry bits: the carry-out of one slice becomes the carry-in of another. An `addc` operation on one pair of registers can be *chained* with an `add` operation on the other pair to perform bit-sliced addition on two 16-bit numbers.

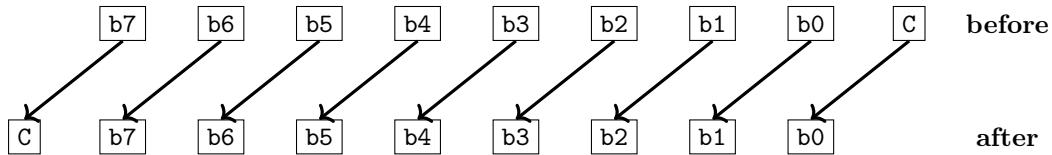
Here the carry is used in the same in-out fashion, but to implement bit-sliced division by 2 by chaining together a `shra` operation with a `shr` operation. So we implement division by 2 for a 16-bit two’s complement number by performing a `shra` on the register that contains the high-order byte of the number, followed immediately by a `shr` on the register that contains its low-order byte.<sup>9</sup> Observe:

```
# Trying 16-bit shifts
# the 16-bit number 0000 0011 1010 1010 is used to represent the decimal value 938
# We have placed it in memory in big-endian form
# First we load the two bytes into a pair of registers
    aset 0x00
    ldi      r0, data      # r0 -> data  (the first byte of the number)
    ld       r0, r1        # r1 now contains 0b00000011
    inc      r0            # r0 -> data+1 (the second byte of the number)
    ld       r0, r2        # r2 now contains 0b10101010
# Next we shift each of the two registers one place to the right:
    shra    r1
    shr     r2
# The result will be r1=0b00000001, r2=0b11010101 representing the number 469=938 /2
# Flags will be C=0, V=0, N=1, Z=0. The same code also works for negative numbers
    halt
data:   dc      0b00000011, 0b10101010
end
```

<sup>9</sup> This is where the endianness issue discussed in section 3.5.6 becomes relevant. We have chosen a big-endian representation for our example. Our platform can cope just as well with a little-endian representation but the program needs to be updated to do so.

Note that only the C flag contains useful information about the 16-bit result at this point (C contains the remainder after division by 2). All other flags contain information that is relevant *only to the low-order byte*. This is because the PS register is overwritten by each ALU operation, and the second shift operation was performed on the low-order byte.

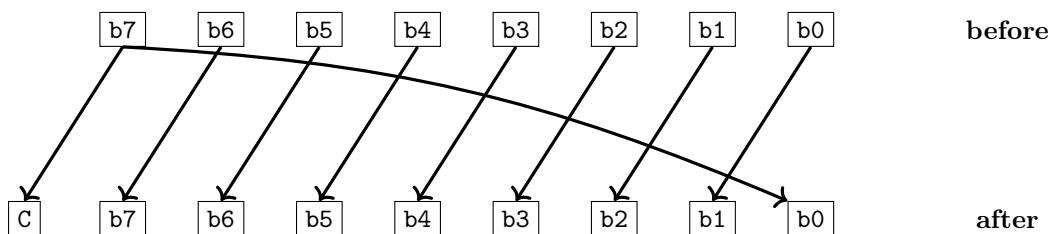
**Sliced shift left.** The operation `shl`, treats the operand as an 8-bit slice of a larger number which needs to be shifted to the left:



Here the idea is the same as for `shr`, with the difference that the direction of the shift is the opposite. The operation affects all flags and gives exactly the same result as would be obtained using `addc rn rn`.

It can be chained with a `shla` operation to perform multiplication by 2 on 16-bit two's complement numbers in a similar manner to the way `shra` can be chained with `shr` to perform division by two, except that this time we `shla` the low-order byte first, then `shl` the high-order byte, and the V flag tells us whether or not there's been an overflow.

**Rotate left.** This is a shift with a twist. It looks very much like a left shift, but here the sign bit does not only end up being shifted out into C, it is also copied into the result's b0:



The purpose of rotation is to examine a bit pattern bit-by-bit without losing anything. Eight such rotations bring the pattern back to its original value, whilst all the time setting and clearing C, which can be used for decision making.

**Move.** This is the biggest misnomer of all in conventional computer architecture. Hardware engineers are accustomed to using the word “move” instead of “copy”.

When you move data from place to place around a platform, you cannot really *remove* it from any place, since that place is digital memory. Such memory always has data in it, which in a real system would typically be unpredictable until the first programmer-specified value is copied into it.

Any bit-string “moved” from one place to another is actually still in the original place *as well as the intended destination*, and will remain there until another bit-string is “moved” in. So if you have computed your data in one register but you wish to *also* have it in another, you “move” it there.

```
move rn, rm
```

The first operand is called the *source* register and the second is called the *destination* register.

Platform 3<sup>1/2</sup> is based on the CdM-8 processor, which treats the `move` operation as an *ALU* operation, hence when the data is copied from the source register to the destination register, the flags in the PS register are changed according to the value copied into the destination. This is efficient (i.e. saves memory for instructions) when data movement coincides with a point in a program where a decision must be made about what sequence of instructions to execute next.<sup>10</sup>

---

<sup>10</sup> If you are eagle-eyed and on your toes you may be asking “what happens if the source and destination are the same

## 5.8 Controlling the order of execution of instructions

So far we have been able to write programs that were a strict *sequence* of instructions, from first to last. Real programs are rarely like this. In most cases the sequence of actions that has to be taken depends on the value of one or more data items. To be able to deal with this a programmer needs *control constructs*.

A control construct is not an individual instruction, but a group of logically related instructions that, taken together, can be used to control which sequence(s) of instructions are executed, and how many times.

We will now explore the control constructs provided by the Platform 3½ macro-assembly language, but before we do, one common feature of them should be noted. Control constructs interact with the rest of the instructions solely by examining the CVZN flags that make up the PS register.

The CVZN flags are the property of the ALU and can only be changed by an ALU operation. The rest of the operations, *including* the control operations (as well as those that are incidental to control and are also covered in this section) **never** change any flags.

### 5.8.1 Conditional execution of a sequence (a simple if)

We start with an example of a situation where a *conditional* construct is required.

Perhaps one of the simplest is the computation of the absolute value (also known as the *modulus*) of a signed number. The algorithm is as follows: test the number; if it is negative, change its sign; if it is non-negative, do nothing.

Here we have three ingredients of a control construct: the computation of some decision data in order to set the flags (in this example we test the number); checking the flags to determine whether a condition has been met (is the number negative?); and a sequence of instructions to execute if the condition is satisfied (change the sign of the number). Assuming the initial data is in r0, we use the following code:

```
if
    tst r0
is mi
    neg r0
fi
```

Note: the indents are there just for ease of reading by a human being. The assembler ignores all white space at the beginning of a line, so indents have no effect on the Level 2 machine code it generates.

- The instruction **if** *opens* the conditional construct
- The instruction **is** marks the *decision point*
- And the instruction **fi** *closes* the construct.

That takes care of the *structure* of a simple conditional construct, but what about the *content*?

- The part between the **if** and the decision point is called the *analysis*. It is here that the CVZN flags are set up. It is not important when the flags are changed: what is important is what values they have just before the decision-point operation.
- The operand of the decision point is called the *if-condition*. It defines *how* the flags should be interpreted to determine whether or not the consequent should be executed.
- The sequence of instructions between the decision point and the close of the construct (indicated by **fi**) is called the *consequent*. This sequence of instructions will be executed in the event that the if-condition is satisfied.

In this example, the if-condition is **mi** which stands for “minus”. The platform interprets that as “N = 1” and checks whether that is the case. If it is not, then the consequent is ignored.

---

register?”. No problem. The contents of the register are copied over itself (so the net effect on the register is no change) and the flags are set, so we get the same outcome as we’d get from a **tst** operation.

condition	test	interpretation
eq/z	Z	equal, equal to zero / Zero is set
ne/nz	$\neg Z$	not equal, not zero, Zero is clear
hs/cs	C	unsigned higher or same / Carry is set
lo/cc	$\neg C$	unsigned lower / Carry is clear
mi	N	negative (minus)
pl	$\neg N$	positive or zero (plus)
vs	V	oVerflow is set
vc	$\neg V$	oVerflow is clear
hi	$C \wedge \neg Z$	unsigned higher
ls	$\neg C \vee Z$	unsigned lower or same
ge	$(N \wedge V) \vee (\neg N \wedge \neg V)$	greater than or equal, greater than or equal to zero
lt	$(N \wedge \neg V) \vee (\neg N \wedge V)$	less than, less than zero
gt	$(\neg Z \wedge N \wedge V) \vee (\neg Z \wedge \neg N \wedge \neg V)$	greater than, greater than zero
le	$(Z \vee N \wedge \neg V) \vee (\neg N \wedge V)$	less than or equal, less than or equal to zero

Figure 5.4: Control conditions.

### 5.8.2 Control conditions

Each possible decision-point condition is listed in fig 5.4, complete with a Boolean expression that shows how the flags are combined to determine whether or not that condition holds (the *test*), and how the result of that evaluation may be interpreted.

All conditions are distinct except that, for program readability, there are alternative names for the conditions that check just the Z or just the C flag. The first pair of alternatives stems from the fact that Z is set

- when *comparing* two equal numbers, in which case the flag has the meaning “equal”, or
- when testing a single 0-valued operand, or moving it (i.e. copying) to another register, in which case the meaning of the flag is “zero”.

The second pair of alternatives is provided because the value of the C flag also tells us different things under different circumstances:

- When comparing unsigned numbers the C flag is 1 if the first number is higher-or-the-same-as the second, and 0 if the first is lower than the second. For these circumstances we use the names hs (higher or same) and lo (lower).
- Other ALU operations (importantly shifts) change C and it may be necessary to do different things depending directly on its value. In these circumstances we use the names cs (C set) and cc (C cleared).

However, the conditions eq and z, hs and cs are completely synonymous, as are ne and nz, lo and cc: it is the choice of *test* that is important, not the name we use for it. On the other hand, programs are easier to follow if the name used corresponds to the purpose of the test.

It might be surprising that some of the conditions involve Boolean expressions that combine several flag values, for instance, gt uses *three* flags - one of which is oVerflow - in order to determine whether one number is greater than another. The reasons for that were explained earlier, in section 3.6.8.

### 5.8.3 Conditional execution with an alternative (an if with an else)

Our first example dealt with a situation where the execution of a sequence of instructions is *optional* - the program requires that the platform either does *something*, or does *nothing*, depending on the value of a control condition. It is also possible to devise conditional constructs that can be used to *choose* which one out of two (or more) different sequences of instructions will be executed, based upon tests of one or more conditions.

The simplest of these is the situation where the program is required to do one thing when an if-condition is met, and a different thing when it is not. Such situations may be dealt with in CdM-8 Platform 3<sup>1/2</sup> macro-assembly language by adding an `else` section to the `[if...is...fi]` construct.

An `if` with an `else`:

- The instruction `if` *opens* the conditional construct
- This is followed by the *analysis*: a segment of code that sets up the flags in the PS register
- The instruction `is` marks the *decision point*, and contains a *condition*
- This is followed by the *consequent*: a segment of code that is executed when the condition is *true*
- The instruction `else` marks the end of the consequent
- This is followed by the *alternative*: a segment of code that is executed when the condition is *false*
- The instruction `fi` *closes* the construct.

For example, here is a piece of code that adds 10 to `r0` if the register contains zero and multiplies it by 2 otherwise:

```
if
  tst r0
is z
  ldi r1, 10
  add r1, r0
else
  shla r0
fi
```

Here the condition `z` in the decision point stands for “zero”, which is evaluated by checking whether or not Z is set (is `Z=1?`). The analysis, consequent and alternative can contain any macro-assembly language code, including control constructs, some of which may be conditionals in their own right.

Continuing with the current example, let us further demand that if `r0` contains a nonzero number it is only multiplied by 2 if that number is in the range -64 .. +63 (thus avoiding overflow). Otherwise it will be left unchanged.

Here are two possible ways of writing this:

```
if                               if
  tst r0                         tst r0
is z                            is z
  ldi r1, 10                      ldi r1, 10
  add r1, r0                      add r1, r0
else                           else
  ldi r1, 63                      if
  if                                ldi r1, 63
    cmp r0, r1                     cmp r0, r1
  is le                           is le
    ldi r1, -64                   if
    if                                ldi r1, -64
      cmp r0, r1                  cmp r0, r1
    is ge                           is ge
      shla r0                      shla r0
    fi                             fi
  fi                             fi
fi
```

Note that it doesn't matter whether the `[ldi r1, ...]` instructions appear before, or within, the `if`: it is the `cmp` that changes the values of the flags.

### 5.8.4 Complex conditions

Finally, it may be necessary to test more than one condition in a conditional construct. In such a case two or more conditions may be connected by logical connectives, which are usually logical-and or logical-or.

Here are some examples of algorithms that compute  $y$  given  $x$ :

1. If  $x$  is positive **and** even, assign  $x/2$  to  $y$ ;
2. if  $x > 3$  **and**  $x \leq 7$ , assign  $x - 3$  to  $y$ , else set  $y$  to 0;
3. if  $x = 1$  **or**  $x$  is even, assign  $x - 1$  into  $y$
4. if  $x < 0$  **or**  $x$  is even **and**  $x \geq 10$ , assign  $x$  to  $y$ , else assign  $-x$  to  $y$

If you have a knowledge of elementary logic you will immediately see that any condition at all can be captured by using **ands** and **ors** to combine the listed control conditions. We do not need **not** because each condition that may be used in the decision point of a conditional construct comes complete with its negation, e.g. **eq/ne**, **gt/le**, etc.

One question that remains is whether or not we need brackets when we have a compound condition where parts of it are shared between **ands** and **ors**, as is the case in example 4 above.

- The macro-assembler follows the prevailing convention in logic, namely that **ands** have a higher precedence than **ors**.

For example,  $p \text{ or } q \text{ and } r$  has the same truth value as  $p \text{ or } (q \text{ and } r)$ .

- It is also known from logic that it is always possible to convert an expression (bracketed or not) made up of propositions joined with **ands**, **ors** and negation, to groups of **ands** connected by **ors**<sup>11</sup>. Hence we never really *need* any brackets.

### 5.8.5 Complex conditions in macro-assembly language programs

Back to Platform 3<sup>1/2</sup> operations, the macroassembler offers another form of **is** which is suitable for coding those groups of **ands** connected by **ors**. We can specify a logical connective (either an **and** or an **or**) as the second operand to **is**, and construct a complex condition by stringing together a sequence of these, giving an overall if-condition. For instance, example 4 can be written as follows:

```
# x is assumed to have been loaded in r0
# Afterwards r0 will still contain x and r1 will contain y

if                      # IF x < 0 .. (tested below)
  tst    r0              # test r0 against 0, setting flags for 'is'
is lt, or                # OR x is even .. (tested next)
  move   r0, r1           # copy r0 into r1
  shra   r1              # Divide by 2. If r1 was even, C becomes 0
is cc, and               # AND x >= 10 .. (tested next)
  ldi    r1, 10           # load 10 into r1
  cmp    r0, r1           # compare r0 with r1
is ge
then                   # THEN
  move   r0,r1            # r1 := r0
else                   # ELSE
  move   r0,r1            # r1 := r0
  neg    r1              # r1 := -r1
fi                      # ENDIF
```

<sup>11</sup> This is called *Disjunctive Normal Form*

Notice that if you use the and/or connective in an **is**, the instruction **then** *must* be included. It marks the point at which all parts of the complex if-condition have been listed. It is only at this point that the platform can determine whether or not it should execute the consequent, which follows immediately after the **then**.

When programming (especially for Level 3½ or below) it is rare that we will need to make a decision based upon a combination of two conditions, and we will almost never need one that is based upon three or more. It is more common for an **if** to include further conditional constructs in its consequent, or its alternative. In such cases different conditions control different segments of code.

Most programmers will only ever use **ifs** with a simple condition. However, it should be reassuring to a discerning code writer that a complete solution is available should it be required at any point.

### 5.8.6 Lazy evaluation of conditions

When an **if** construct contains a complex condition like the one above we must assume that the process of evaluating its parts proceeds in the order given, and no part of the consequent - or the alternative - is executed until this has been done.

- Each logical operator is applied in turn, with a ‘running’ result being carried forward to the next.
- If the end result is *true* (here represented by 1) the consequent is executed.
- If the end result is *false* (0), the alternative (if any) is executed.

However, this form of evaluation has an interesting - and useful - side-effect. The rules for *logical or* and *logical and* tell us that *true or p* is always *true*, and that *false and p* is always *false*.

- So when the *condition* in an

*is condition, or*

turns out to be *true* the platform can take a shortcut directly to the consequent, without executing any more analysis segments.

- Similarly, when the *condition* in an

*is condition, and*

turns out to be *false* the program can take a shortcut directly to the alternative, without executing any more analysis segments.

This is useful for the cases when it may only be safe to perform a later test if an earlier test has already been passed: for example if testing for  $a \neq 0$  and  $b/a > 3$ , we do not want to perform the second test until *after* we have established that  $a \neq 0$ ; otherwise we might attempt to divide  $b$  by 0 (an illegal operation, because its result is undefined).

Such behaviour of logical connectives when used in programs, namely the assumption that *p* has been found to be *true* before we attempt to evaluate *q* in the expression *p and q*, and the assumption that *p* has been found to be *false* before we attempt to evaluate *q* in the expression *p or q* is called *lazy* evaluation of Boolean expressions. Both connectives in a Platform 3½ **if** construct are thus lazy.

### 5.8.7 Repetition

It is hard to find an algorithm that does not require repetition of the same action several times (possibly with some variations). Yet all code we have seen so far executes top down. Clearly, just replicating instruction sequences to achieve repetition is not a solution: this takes large and potentially unlimited (when the number of repetitions depends on data) amount of memory. What we need instead is a construct that allows us to execute a sequence of instructions, then execute the same sequence again, without having to insert it into the program another time, and to keep doing so until some condition has been met.

There are two mechanisms for achieving this: *iteration* and *recursion*. Iteration is achieved using *program loops* and recursion is achieved using *recursive subroutine calls*. CdM-8 supports both of these techniques, but for the time being we will concentrate on iteration. We will discuss subroutines and recursion in section [6.2](#).

### 5.8.8 Program loops

A program loop is a control construct that executes a segment of code repeatedly while (or until) a certain condition is met. Accordingly, Platform 3<sup>1/2</sup> has two loop constructs: **while** and **until**.

- A **while** loop executes a segment of code repeatedly *while* a certain condition *stays* true. The condition is checked at before the *start* of segment execution.
- An **until** loop executes a segment of code repeatedly *until* a certain condition *becomes* true. The condition is checked after the *end* of segment execution.

All loops have certain things in common:

- The code segment is usually called the *body* of the loop.
- Each time the body is executed, we call this an *iteration* of the loop.
- In many algorithms it is important to know how many iterations have been performed so far. This is done by setting up an *iteration counter*.
- It must be possible for the result of testing the condition to change from one iteration to the next, or else the loop will either never execute or never terminate.

The Platform 3<sup>1/2</sup> instruction set does not supply a construct with a built-in iteration counter to specifically support counted loops. If a count is required, a register or a memory cell must be allocated as an iteration counter, and incremented on each iteration. Note, however, that the counter *must be initialised* before the loop is entered, since iterations only adjust it up or down.

### 5.8.9 The **while** loop construct

The condition of a **while** loop is often called the *continuation condition* since iterations continue (including the very first one) only if the condition evaluates to true. Having executed the body the program returns to the start of the loop to test the condition again, etc. until the continuation condition evaluates to false, at which point program execution continues from the first instruction *after* the loop.

**Note:** Because the continuation condition is always tested *before* the program execution reaches the loop body it is possible for the body of a **while** loop *never* to be executed.

Here is an algorithm showing how a **while** loop can be used to multiply together two unsigned numbers held in **r0** and **r1** by repeatedly adding **r0** to **r2**:

```
load the first number into r0
load the second number into r1
initialise r2 to 0
while r1 is not equal to 0
    add r0 to r2
    decrement r1
end-of-loop
```

Here is the same in CdM-8 Platform 3<sup>1/2</sup> macro-assembly language:

```
# r2=r0*r1 (assuming r1 is non-negative)
    clr  r2
while
    tst   r1
stays gt
    add   r0, r2
    dec   r1
wend
```

Here we use the instruction `wend` to mark the end of the loop body, and the decision point is marked by the instruction `stays`, which introduces the *continuation condition* (in this case it is `ne`). We may use exactly the same conditions for loop control as we did in conditional constructs. These are set out in figure 5.4.

A `while` loop is structured as follows:

- The instruction `while` opens the loop construct
- This is followed by the *analysis*: a segment of code that sets up the flags in the PS register
- The instruction `stays` marks the *decision point*, and contains a *condition*
- This is followed by the *body* of the loop: a segment of code that is executed when the condition is *true*
- The instruction `wend` marks the end of the body, and closes the loop construct

### 5.8.10 The until loop construct

In an `until` loop, the loop body is executed first, and then the *termination condition* is tested. If it is false the program executes the loop body again. This continues until the *termination condition* evaluates to true, at which point program execution continues from the first instruction *after the end* of the loop.

**Note:** Because the termination condition is only tested after the loop body has been executed, it is guaranteed that the body of an `until` loop will be executed at least once.

A good example of an until loop is a program that finds a zero in an array of numbers. This program will work its way through the array until it finds the first zero, and will then stop.

```
# find a zero
    ldi    r0, array-1      # Initialise r0 to point to the cell before the first element of the array.

do
    inc    r0      # point r0 to the next element
    ld     r0, r1 # read the element into r1
    tst    r1      # examine it
until z          # if r1 is 0 then exit, otherwise continue

halt
array:   dc 3,5,2,7,3,0,57,8,0,6
end
```

Notice that the until loop construct is opened by the instruction `do` which merely marks the start of the body. The word `until` marks both the decision point and the end of the loop body.

An `until` loop is structured as follows:

- The instruction `do` opens the loop construct
- This is followed by the *body* of the loop
- The *analysis* is part of the loop body. It is usually the very last part, so that the flags are set correctly immediately before the termination condition is evaluated
- The instruction `until` with its *termination condition* is both the decision point and the end-of-loop marker.

### 5.8.11 Nesting of control constructs

It is important to bear in mind that the body of a loop can contain any code we wish, including conditional constructs and other loops. And these loops need not be of the same type as the ‘outer’ loop, or as each other.

This practice is known as *nesting* of control constructs, and is frequently required in order to solve non-trivial problems.

The ability to nest one construct inside another is common to all control constructs, and indeed to pretty much all programming languages. Control constructs are, in fact, *scoping* instructions that are used to introduce program segments, which can, and often do, contain further scoping instructions of any kind. They have well-defined syntax that tells programmers (and the macro-assembler) what words to use in order to obtain each of them, including how each starts and ends, and what parts need to come in between.

Control constructs, and the nesting of those constructs, are very powerful tools in the programmer’s toolbox, but with great power comes great responsibility, and the potential to really mess things up! When we were dealing with simple sequences of instructions the only thing that mattered was that we use the right instructions in the right order. Now the situation is more complex, because we are embedding whole segments of code inside one another, and if we get it wrong the program may execute many instructions under the wrong circumstances, or in the wrong order. A common trick in building iterative programs is to think about them in terms of individual iterations and use technology to stop the program at decision points to test the programmer’s assumptions.

### 5.8.12 Early termination of an iteration

It is not uncommon to find that some iterations, or even the whole of a loop, may require early termination. This typically happens when the loop continuation or termination condition expresses a hard limit of how far the iterations may go and the precise point is determined inside the iteration code, often when something unusual happens that makes further processing either impossible or undesirable.

For example, in the multiplication program presented in Section 5.8.9 the loop should terminate early if overflow occurs after an addition. If the loop were to continue, the next addition could clear the V flag, which would make it look like the overflow never happened, and there would be no way to spot the fact that there had been a problem. Here is how this could be remedied:

```
# r2=r0*r1 (assuming non-negative r1) with overflow detection
    clr    r2
while
    tst    r1
stays gt
    if
        add    r0, r2
    is vs
        break
    fi
    dec    r1
wend
# At this point V=1 if multiplication resulted in overflow
# Otherwise V=0, in which case the result in r2 is valid.
```

**break** The instruction **break** causes program execution to jump to the first instruction after the end of the loop, as if the correct condition had been met for loop exit. Interestingly, a **break** can occur inside an **if** construct of arbitrary complexity, so it is possible to set up a complex termination condition for either type of loop this way<sup>12</sup>.

<sup>12</sup>The CdM-8 Platform 3<sup>1/2</sup> language is extremely basic. Without the **break** instruction it would be impossible for loops to support an arbitrary condition, since the decision points **stays** and **until** do not permit the kind of composite conditions that we saw in the two-operand decision point **is**. In higher-level languages, there is no restriction on conditions. The instruction **break** or its analogue is only used for incidental loop termination, just as we use it most of the time in CdM-8 Platform 3<sup>1/2</sup>.

**continue** When it is necessary to terminate the execution of the current *iteration* of a loop, but to continue executing the loop from the beginning of the *next* iteration if the condition permits it, we use the **continue** operation. For example, if we wanted to work through an array looking for the first zero that is not immediately preceded by a 3, we could modify the until loop given in Section 5.8.10 thus:

```
# find a zero that does NOT come after 3 in an array
    clr r1          # assume the previous element is 0 at the start
    ldi r0, array-1 # previous address before the start of the array
do
    move r1, r2      # r2 will hold the previous element
    inc r0          # point r0 to the next element
    ld r0, r1        # get that number in r1
    if
        ldi r3, 3
        cmp r2, r3    # previous element = 3?
        is eq          # if yes
        continue       # not interested in current element, continue to next iteration
    fi
    tst r1          # examine current element
until z          # if current element = 0 then exit, otherwise go on
halt
array:
    dc 3,5,2,7,3,0,57,8,0,6
end
```

**break n** By default, a **break** terminates the innermost loop in which it appears, so if it is inside a loop that is, itself, nested within another loop, it will terminate only the innermost loop. However, a single **break** may be used to terminate a whole nest of loops (up to 4 deep) at once, by supplying a numeric operand, so **break 2** will terminate both the innermost loop in which it resides, and the loop within which that is nested (but nothing further out). An example can be found in section 6.1.

**continue n** The **continue** instruction can also have an optional operand and terminate the iteration of an outer loop, e.g. **continue 2** will pass the control to the beginning of the next iteration of the loop in which the current one is nested (if the loop's condition permits it), with the current loop being completely abandoned. The alternative would be to **break** out of the inner loop and then use **continue** to continue execution from the beginning of the next iteration of the outer loop, but this would mean having two tests: one at each level, and it would require some programming acrobatics to ensure that the outer test would only cause a **continue** when the inner test had caused a **break**.

## 5.9 Problem solving and complexity

Platform programming is all about problem solving. Before we can use a platform to solve a problem we need to establish:

1. What problem needs to be solved (the starting state of the platform and the desired finishing state)
2. What kinds of data a program will have to handle
3. What data structures will best meet our needs
4. What sequence of steps will solve the problem, including
  - (i) What choices the program will need to make
  - (ii) What code will need to be executed more than once, and what tests will be needed to control the number of iterations

Programming is not a straightforward matter at any platform level, but it can be particularly complex at Level 3½ and below, because the individual steps are so small that we have to use many of them to achieve even quite simple goals. When programming a Level 3½ or Level 3 platform we need to understand:

1. How to represent different kinds of data in memory
2. The set of platform operations available and what each of them does
3. How to write instructions to use these operations in a program
4. The order in which instructions must be issued / executed in order to perform certain common tasks
5. How operations on data can be used to set up conditions
6. How to use conditions to control execution
7. The instructions that form control constructs and their precise behaviour
8. When and how to choose and use control constructs in order to solve problems
9. How to nest them correctly

The macro-assembler will do its best to ensure that your instructions make sense (in particular it will check that you've used legal mnemonics and that you've provided appropriate operands), and it will check that your programs are properly scoped, i.e. that control constructs appear in sequence, or are nested, but do not overlap. For example, the compiler will detect that an `if` has a `while` instruction in the body, but that `wend` did not occur before the closing `fi`.

When a programmer breaks the rules the macro-assembler generates error messages about things like unknown names, missing operands, illegal scoping, and missing parts of control constructs, and the assembly process aborts. Whilst these may be of some help, an assembler cannot tell you *what* the correct pattern of code should be when it detected those violations.

The best it can do is to tell you how far it got before it had to give up trying to assemble your code because your error rendered the program unintelligible, and - possibly - let you know what sort of a mess it was in when it *did* give up. Worse still, no tool can tell you when you have made an error in *program logic* (your choice or ordering of operations, or of conditions) because a tool cannot know what you *meant* to write, or what you *should* have written in order to get the result you want.

So you should employ a certain discipline to ensure that your programs are perfectly scoped *before* subjecting them to analysis by the assembler. This is another reason for adopting a set of layout conventions when formatting your source code, such as how you will use indents and blank lines, and it is a motivation for the use of high quality comments. This combination will help you to avoid errors and reduce the likelihood that your program will be misinterpreted, either by tools or by its author and other developers. This becomes particularly important after time has passed and your initial intentions have been forgotten.

# Subroutines, Stack and Heap

With the addition of facilities for input and output, the resources and operations you have met so far are sufficient to enable us to solve any problem that has an algorithmic solution, provided that the solution is not too large. In principle, at least. However, that does not mean the solution will be efficient, or elegant, or easy to follow, or that we will obtain the results we want in a timely fashion. In fact, in *practical* terms, there are quite simple problems that it would be pointless to tackle using the elements of the platform you have met, and complex problems that would be more difficult to solve with this platform than it would be to design a more capable platform and program that instead!

One approach would be to construct a program that implements a more capable platform “on top” of CdM-8. Of course that can be done ... in fact it is fundamental to the Tanenbaum hierarchy. However, whilst that may make it easier for a human being to create programmed solutions to more complex problems, it doesn’t get us over the restrictions in terms of how much useful work CdM-8 can do in a single instruction, or how quickly each of them can be executed. These are limitations placed upon us by the Level 2 platform, and by the hardware used to implement it.

We now start to look at resources and instructions that are included to make CdM-8 programs better structured, and to improve their efficiency. Some of these are provided by the Level 2 machine, whilst others are built on top of it to create CdM-8 Platform 3½. The first of these is the stack, which is supported directly by CdM-8 Platform 2. It is managed using a special Stack Pointer register, and manipulated using a set of special Stack-oriented instructions.

## 6.1 Running out of registers... what to do?

```
save rn
```

```
restore rn
```

As mentioned earlier, loops can be nested. However, nested loops can be problematic. When they are used for purposes of traversing data and/or building some kind of result incrementally, the load on the registers quickly increases. There are, after all, only 4 general purpose registers available, yet there is much to be gained from keeping all iteration counters, pointers into data structures, array indices and similar in registers.

When we are short of registers we need to keep such data in memory instead, which means copying it backwards and forwards every time it is used, which in fact by itself increases the register demand, as registers are needed for memory operations too!

The vicious circle (need registers to free up registers) can only be broken by instructions that save and restore register content to memory without needing an address to be loaded in another (potentially spoken for) GP register, see the two instructions above. This can be achieved if the platform has some special place in memory that can be assumed rather than specified by giving an explicit address, and at which register content can be stored thus freeing up a register temporarily. Indeed, such a place exists and it is called the *stack*, see section 6.3 below. We will see the mechanics of stack operation later on. For now, it is only important to understand that the **save** and **restore** instructions use that special part of memory to do their job; that the part has to grow if more data needs to be stored in it at the same time, and that the platform

memory is finite: if left unchecked the stack will sooner or later encroach on the program and may destroy it.

The reader may wonder how this could happen given that there are only 4 registers to save, requiring 4 bytes of memory. That is not so; **save** and **restore** introduce a scope (like conditionals and loops) and the register content is saved and restored at the beginning and end of the scope, respectively. The scope itself can contain a further pair of **save/restore** referring to the same register, etc. In principle the stack can grow quite far and present a danger to the code that is contained in the same memory. Imagine a loop in which on some condition a register's content is saved and on another condition it is restored. The intention is that the interval *in time* between the former and the latter can be used to employ that register in another capacity, work out the result, perhaps store it in memory, then reload the register with its former value and continue to use it as before. This looks reasonable and valid. However, the conditions involved are evaluated at run time, when the program is executing. If the programmer's expectations prove wrong, this can create the imbalance between the number of times the **save** executes and the number of times the **restore** is carried out. In particular the program may issue many **saves** without a single **restore**, which would cause the stack to grow out of control.

If the programmer wishes to operate the stack in this manner, she still may, which is explained further in section 6.3. However, Platform 3<sup>1/2</sup> offers the above **save** and **restore** instructions as a safe alternative. The assembler checks at compile time that the scope introduced by **save** and **restore** is statically balanced. What this means is that the assembler ensures that once a **save** has been encountered, the program will under all conditions reach its matching **restore** instruction.

Below is an example of proper scoping. For the first time we include an assembler listing rather than a source file for illustration. The left hand side of the listing is given to the Platform 2bit-strings that the tool generates in response to the source program. The listing is produced by the **cocas v2.1** macroassembler, which is a compiler that compiles Platform 3<sup>1/2</sup> down to CdM-8 (our choice of Platform 2). Everything there is in hex, and hex memory addresses are prefixed to each productive line of the listing, separated from the memory content with a colon. To the right of the Platform 2 content one can see the line number running down from 1 to 53, which happens to be the last line of the program.

The example we have chosen is a search for Pythagorean numbers, i.e. integers that satisfy the equation  $c^2 = a^2 + b^2$ , using a table of squares and a triple loop. The three registers used to organise the loop are those that hold the current value of  $c$ ,  $a$  and  $b$ , which leaves only one register for intermediate results, **r3**. In the inner loop, on lines 10-11, registers **r2** and **r0** are freed up. It means that even though they still have current values, the programmer need not ensure that those values are preserved; she may use the registers for something else, since **restore** will reload them with what was there before **save**. In this case, **r2** is used as a work register for intermediate results, and **r0** is only available in a smaller scope, where it is used for addressing the table of squares. From line 20 down, **r0** is used as the holder of the variable  $c$  again; in particular on lines 25-27  $c$  is squared and the value of  $a^2 + b^2$ , which was computed while **r0** was used in a temporary capacity (rather than as the holder of  $c$ ) is subtracted from the square. Effectively the program uses 6 registers instead of 4, but the missing two are realised by scoping and sharing **r0** and **r2**.

CdM-8 Assembler v2.1 <<<test.asm>>> 15/06/2017 16:25:38

```
1 # find the greatest int c<15 for which there exists a,b
2 # such that a^2+b^2=c^2. (Such a,b,c are called Pythagorean.)
3     asect 0
00: d0 0e    4 ldi      r0,14          # r0=current c
5     do
02: 01    6 move    r0,r1          # r1=current b
7     do
03: 06    8 move    r1,r2          # r2=current a
9     do
04: c2    10           save r2    # free up r2
05: c0    11           save r0    # free up r0
06: d0 30   12           ldi r0,squares
08: 14    13           add r1,r0
09: b3    14           ld r0,r3          # r3=squares[r1]=b^2
0a: d0 30   15           ldi r0,squares
0c: 18    16           add r2,r0
0d: b2    17           ld r0,r2          # r2=squares[r2]=a^2
0e: 1b    18           add r2,r3          # r3=a^2+b^2
0f: c4    19           restore   # r0=c again
20           if
10: e3 15   21           is cs      # if unsigned OFL, not Pyth
12: 2f    22           shl r3      # ... ensure r3!=0
13: ee 1a   23           else
15: d2 30   24           ldi r2,squares # otherwise calc the diff
17: 12    25           add r0,r2
18: ba    26           ld r2,r2          # r2=c^2
19: 3b    27           sub r2,r3          # r3=c^2-a^2-b^2
28           fi
1a: c6    29           restore   # r2=a, r1=b, r0=c as before
30           if
1b: e1 1f   31           is eq      # (the flags were set by sub)
1d: ee 28   32           break 3      # exit 3 loops at once
33           fi
1f: 8a    34           dec r2      # otherwise
20: e1 04   35           until eq      # next iteration
22: 89    36           dec r1      # next iteration
23: e1 03   37           until eq
25: 88    38           dec r0      # next iteration
26: e1 02   39           until eq
28: d3 3f   40           ldi      r3,result
2a: ac    41           st       r3,r0  # store c
2b: 8f    42           inc      r3
2c: ad    43           st       r3,r1  # store b
2d: 8f    44           inc      r3
2e: ae    45           st       r3,r2  # store a
2f: d4    46           halt
47     squares:
48 # 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
30: 00 01 04 09   49 dc 0, 1, 4, 9, 16, 25, 36, 49, 64, 81,100,121,144,169,196
34: 10 19 24 31
38: 40 51 64 79
3c: 90 a9 c4
3f: 00 00 00   50 result: ds      3      # holds c, b, and a of the result
51           end
```

The usage rules are as follows:

1. Any control structure which occurs between a `save` and its matching `restore` must occur in that scope including both the start and the end parts, e.g. `if` and `fi`, `while` and `wend`, etc.
2. Unscoped control instructions `break`, `continue`, `is`, `stays`, etc. may only occur between `save` and its matching `restore` together with (inside) *the whole construct* that they control.
3. If the programmer wishes to restore the saved value to a different register, this can be done at zero cost by specifying that register as the argument of `restore`, e.g. `restore r3`.

On line 38 it is established that the number the program has been looking for is finally found, so all three loops need to terminate and the control must pass to the first line after the loops, line 46. Notice that placing `break 3` between a `save` and its matching `restore` in the inner loop would be in violation of the above scoping rules, since it would, at least potentially, leave a `save` unbalanced.

## 6.2 Subroutines

Loops are one form of repeated computation; a loop body is repeated a certain number of times depending on certain conditions, but it is still used in a program once. The control is passed to the loop by a preceding instruction and it passes from the loop to the next instruction that follows it. Even though a computation may be repeated, it is still not possible to repeat it at different points in the program; if that were necessary, one would have to include a copy of the loop there. Copying code for purposes of repeated computation has severe implications on all levels, from platform performance to software maintainability. Code copies take address space, which is critical resource for small, embedded systems (such as the microcontrollers in car engines and washing machines). The copies are identical only initially, when the program is first written. Whenever an engineer needs to modify the code, they have to remember to modify all copies consistently. This in itself is a source of errors that are difficult to detect and rectify.

Most platforms therefore provide a means of *referring* to common code, rather than replicating it. Specifically there is an instruction, which Platform 3<sup>1/2</sup> shares with its progenitor, the Cdm-8 processor, that is used for this purpose:

`jsr name`

where the *name* is the symbolic or numeric address of a common segment of code. Such segments are called *subroutines*. The abbreviation `jsr` goes back to the old IBM days where mainframes were loaded with programs, which were known as *routines*, and to the Cambridge computer scientist David Wheeler, who in 1952 invented the “jump to a sub-routine” instruction for contemporary computing platforms<sup>1</sup>. This kind of instruction persists to this day (not a mean feat in an area of science and technology that is barely 70 years old!). In a more general context, we usually refer to the program that jumps to a subroutine as a *caller*, the jump itself is often termed a *subroutine call*, and consequently the subroutine is often referred to as the *callee*.

Here is how it works. As was stated in section 4.2, instructions are placed in memory at certain addresses, and we have already seen that those are represented as labels in assembler. The labels label memory locations of any kind: data, space for data, or instructions. The *name* in `jsr` is the label placed at the beginning of a subroutine. The `jsr` instruction changes the control flow of the program. Instead of passing control to the next instruction (i.e. the one occupying the next memory cell), it passes it to the point labelled as *name*. The subroutine will be executed, instruction after instruction until the platform encounters the instruction

`rts`

called “return from subroutine” which restores the control back to where it was, i.e. passes the control to the instruction that immediately follows the `jsr` that caused the jump to the subroutine in the first place. An example:

<sup>1</sup> for a while it was known as the Wheeler jump, but computer science is quite ungrateful to its great men. There are remarkably few named concepts and objects there, compared, say, to natural sciences and medicine, where textbooks are replete with discoverers’ names. The Wheeler jump is now simply a jump to subroutine

```

# compute e=a*b+c*d

ldi    r0,a
ld     r0,r0  # r0=a
ldi    r1,b
ld     r1,r1  # r1=b

jsr    mult   # r0=a, r1=b, a*b expected in r2
move   r2,r3  # save a*b in r3 for now

ldi    r0,c
ld     r0,r0  # r0=c
ldi    r1,d
ld     r1,r1  # r1=d
jsr    mult   # r0=c, r1=d, c*d expected in r2

add   r2,r3  # r3=a*b+c*d
ldi   r0,e
st    r0,r3  # e=a*b+c*d, job done
halt

a:    dc    17
b:    dc    -2
c:    dc    7
d:    dc    5

e:    ds    1

# subroutine mult: computes r2=r0*r1
mult:
    clr r2
    while
        dec r1
    stays ne
        add r0,r2
    wend
    rts
end

```

Here the instruction labelled `mult`, namely `clr r2` will be executed right after each `jsr` and the `rts` will return the control to the instruction `move r2,r3` after the first `jsr` and to the instruction `add r2,r3` after the second one.

Two observations follow from this example. First, the calling code needs to be aware of the callee's use of registers. In the above example, the subroutine uses `r2` to return the result, and it could (but does not in this example) use the remaining register `r3` as a work register to facilitate its computation. Had it made use of `r3`, the caller would need to know that in order to use the subroutine, since otherwise important data could be lost. The arrangement of initial data for the subroutine (as well as the expectation of the result taking a certain place in memory or registers) is usually referred to as the *calling conventions* of a program. The rules that dictate which registers can be used as the subroutine's work space are named *register discipline*. The calling convention and the register discipline could be shared by a large set of subroutines or even be laid in the foundations of a programming language implementation, so it is fairly important to establish appropriate rules that are not too limiting and at the same time reasonably efficient.

The second observation is that whenever a subroutine is called, the address to which the control should be passed upon its completion (usually called *return address*) has to be saved somewhere. That, in fact, cannot be a single place, since the subroutine is a program in its own right and can consequently call further subroutines. When this happens the single place (if there were one) would be overwritten with a new return address, making it impossible to return from the first subroutine back to the main program.

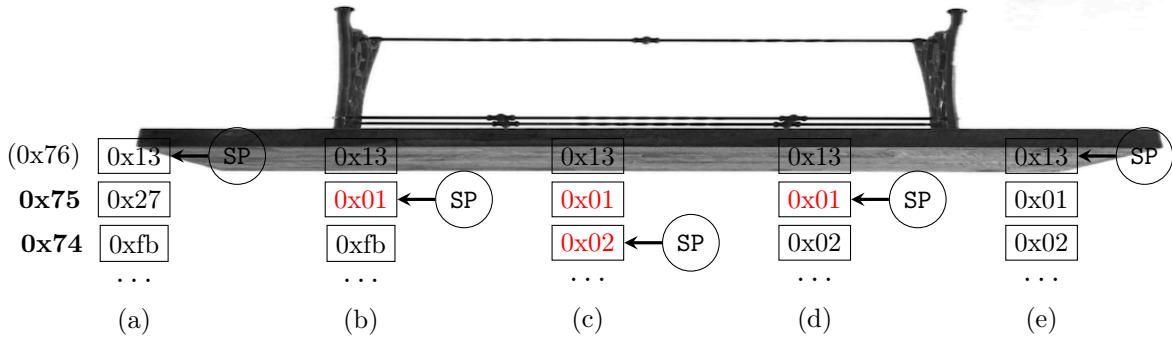


Figure 6.1: Stack behaviour: (a) initial state: the stack is empty; (b) after 0x01 has been pushed; (c) after 0x02 has been pushed; (d) after a pop; (e) after another pop, stack is empty again.

### 6.3 The Stack

Interestingly, the register discipline problem and the return problem have the same solution, which is based on a data structure called *stack*. Imagine a stack of books on a desk. If you put “Romeo and Juliet” on the desk first and then put “Hamlet” on top of it, the first book you will be able to take off the stack is “Hamlet”, which happens to be the last one you put on it. Only then will you be able to take “Romeo and Juliette” off the stack and it will become empty. A stack is consequently a queue of items following the LIFO principle: Last In — First Out.

A computer stack works in the same way. In Platform 3<sup>1/2</sup>, our running example, the top of the stack is a memory cell that is pointed to by a special register: the Stack Pointer (SP). There are two operations one can perform on that stack (just like those on the stack of books). The *push* operation places an item on top of the stack. The *pop* operation removes the item that has been placed on top of the stack before. Repeated pushes and pops will work according to the LIFO principle. If the stack is empty a pop is not allowed. If there are too many pushes without a pop, the stack may use up its available memory: this is called *stack overflow* (nothing to do with adding numbers, everything to do with overusing a resource).

Despite the fact that push and pop deal exclusively with the top of the stack, stack memory is just that, memory, and therefore can be accessed directly, by address. This is useful for using stack for subroutines’ internal data and for parameter passing as well, see section 6.4.2.

It so happens that many things grow in the wrong direction in the topsy-turvy world of computing. While normal trees grow up, a tree as a type of graph (studied in an undergraduate course of discrete mathematics) or a data structure (studied in a course of algorithms and data structures) is usually depicted growing down. The same, unfortunately, is true of stacks. Here is an illustration (see figure 6.1). The stack area begins at the address 0x75 and extends down, towards smaller addresses. One can think of the address 0x75 as the desk upside down, on which books will be piled (obviously with the planet Earth above us keeping them on with its gravity). The stack pointer SP initially points to the cell right above the stack area (0x76). If we push the number 0x01 onto the stack, the SP will go down first and then the number will be stored at the pointed address (stage (b)). Now if we push another number onto the stack, say 0x02, the SP will go down again and the new number will be stored at the pointed address. Now let us try to pop an item currently on top of the stack off the stack. This is done as follows: the cell that the SP points to is read, then the SP is incremented to point to the next cell up (stage (d)). Finally we pop 0x01 off the stack in the same way and it becomes empty (stage (e)).

Unsurprisingly there are two Platform 3<sup>1/2</sup> instructions for doing what has just been described:

`push rn`

and

`pop rn`

The former one pushes the content of *rn* onto the stack in the manner described above, and the latter one pops off the top cell of the stack and loads it into *rn* again, in keeping with the procedure described above. **Neither instruction affects the flags.**

Here is the same example but now as an assembler source:

```
# play with the stack
    ldi    r1,0x01
    ldi    r2,0x02
    push   r1
    push   r2
    pop    r0      # now r0=0x02
    pop    r0      # now r0=0x01
    halt
    end
```

The attentive reader would recall that something of this nature has already been discussed: the `save/restore` pair. Indeed the latter is supported by the former and is largely equivalent to it. The difference is that `push/pop` are Level 3 instructions and come with no assembler assurances regarding the scoping discipline, while `save/restore` are watched by the compiler.

But what about the Stack Pointer? Who sets its value so that it points to the correct address just above the stack area? Where is the stack area in Platform 3½? All of those are important questions to which answers are readily available. Recall that in section 4.2 it was stated that the platform after reset nullifies all its registers *including* the SP. Since the Stack Pointer must initially point to the address *just above* the beginning of the stack area, we must subtract 1 from 0 to locate that area. The answer is obviously 0xff, and that is the first cell that belongs to the stack. Since the stack grows down, it will take more and more memory as pushes are executed (and any pop will reduce the stack footprint until it eventually becomes empty). Naturally the program is placed in low addresses and grows up as we add new instructions to it. It is important to ensure that the imbalance between the pushes and the pops (which defines the area actually taken by the stack) is not too large or else the stack area will grow into the code area and will corrupt the code with stack data. However, this will only happen when *all* 256 bytes of platform memory are used up, either by the program (code and data) in the lower addresses, or by the stack in the higher addresses, and when, consequently, they meet somewhere in the middle. If the stack grew up (as a stack of books does on a writing desk) rather than down, it would be necessary to have a reliable estimate of its memory requirement since that amount of space would have to be lost to other uses, including temporary data required for the program. A stack growing backwards and the program placed from the beginning of the address space leave a *contiguous* shrinking and expanding segment of memory in the middle for temporary data. Any memory unused at any given time can be utilised for storing object as large as the total available memory without worrying about fragmentation. That is the main reason why in a modern architecture the stack is typically found hanging upside down from the top of the address space like a sleeping bat!

### 6.3.1 Subroutines and the stack.

So what is this all to do with the topic of the section, the subroutine? As mentioned before, a subroutine call has to store the return address somewhere, since the subroutine must, upon completion of its work, return the control to the instruction following the `jsr` that it received it from. Since the subroutine is a program in its own right, it can, and often does, call further subroutines. If *A* calls *B* and *B* calls *C*, the control should pass from *A* to *B* to *C* and then it should be returned from *C* to *B* and finally back to *A*. It is easy to see that the storage for the return addresses has to observe the LIFO discipline of a stack. The control should be returned *first* to the subroutine that lost it *last*. This is immediately achieved by pushing the return address (i.e. the current value of the Program Counter when a `jsr` is being executed) onto the stack, and then jumping to the subroutine starting address. A return from the subroutine does the exact opposite thing: *first* it pops the return address off the stack and then passes the control to the instruction at that address. It is as if `rts` executes the instruction `pop PC` (which does not exist since the PC is not a GP register and cannot be named in CdM-8 instructions).

Finally, to the register discipline. Software development follows a very important design principle first proposed by Dijkstra, so called “separation of concerns”. Software and systems tend to be very large and complex artefacts requiring considerable mental power to comprehend. The only way to beat that complexity is by relentless structuring and abstracting various parts, just as Tanenbaum does in his structured computer organisation discussed in section 2.1. In particular, one must strive to make a subroutine as self-contained and decoupled from possible “users” (i.e. callers) as possible. Yet it needs an interface for parameter passing and resources for doing its work.

The registers used to pass initial data to a subroutine and pass the result back to the caller do not present a problem as far as separation of concerns. The caller and the callee would need to know about the calling conventions, and as was mentioned earlier those could be well defined, formally stated, and adhered to collectively by a whole set of subroutines in a project, developer team, etc. The problem is really only the use of private resources by a subroutine. Memory locations used for keeping the subroutine's work data could easily be hidden from other subroutines by keeping the labels private (one would presume that a correctly working piece of code will not be storing data at random addresses but will use labelled locations that belong to that piece). Registers are another matter: there are only a small number of them and they have to be shared by all code. It would be convenient for the caller to assume that anything it does not change will remain unchanged after the return from the subroutine.

If a subroutine wishes to use registers other than those in which its initial data is presented to it according to the calling conventions, or if it needs to change that data in the process of working out the result, the affected registers must be saved (i.e. their values must be copied to memory) first and then restored (copied back). That way it will look to the caller as if the callee has not changed any of them. The subroutine can use `push/pop` or `save/restore` for this purpose.

Naturally the subroutine can just allocate a few memory cells for the same purpose and label them; however, this may not really be a solution as saving registers to specific addresses require free registers! Indeed the `st` command has two register-operands, one the address and the other the data. The second register (the data) is the one we are trying to save, but we must load the address at which the value is saved in another. But that other register needs to be saved as well, since all of them are either part of the calling convention or private to the caller.

To resolve this chicken-and-egg situation, the only solution (and it happens to be a good one) is to use the stack. The subroutine must push all the registers it intends to write into onto the stack before using them; it must also pop those values off the stack (in reverse order) back to the registers that they belong to before issuing an `rts`. Here is an example:

CdM-8 Assembler v2.1 <<<test.asm>>> 15/06/2017 16:31:04

```

1           asect 0
2 # subroutines that observe register discipline
3 # compute: if even (a+b) then res=2b else res=c-b
00: d0 18    4      ldi     r0,a
02: d1 19    5      ldi     r1,b
04: d2 1b    6      ldi     r2,w1
06: d6 1d    7      jsr     sum    # r0->a, r1->b, r2->res, res will get a+b
08: 08      8      move   r2,r0  # now r0 -> (a+b)
               9
09: d3 1a    10     ldi    r3,c
0b: bf       11     ld     r3,r3  # r3=c
0c: d2 1c    12     ldi    r2,res # r2->res for the computation ahead.
               13     if
0e: d6 26    14     jsr   even  # r0-> (a+b); C clear if that is even
10: e2 15    15     is cc
12: 95       16     shla  r1    # even(a+b), r1=2b
13: ee 16    17     else
15: 3d       18     sub   r3,r1  # odd(a+b), r1=c-b
               19     fi
16: a9       20     st    r2,r1  # r2-> res
17: d4       21     halt
               22
18: 17       23     a:    dc     23
19: 04       24     b:    dc     4
1a: f8       25     c:    dc     -8
1b: 00       26     w1:   ds     1
1c: 00       27     res:  ds     1
               28
               29
               30 # subroutine sum: r0->x, r1->y; result r2->res
1d: c1       31     sum:  save  r1
1e: c3       32     save   r3    # now can write into r1 and r3
1f: b3       33     ld    r0,r3  # r3=x
20: b5       34     ld    r1,r1  # r1=y
21: 17       35     add   r1,r3  # r3=x+y
22: ab       36     st    r2,r3  # res=x+y, job done
23: c7       37     restore
24: c5       38     restore
25: d7       39     rts
               40
               41 # subroutine even: r0->x; result: C=0 if even, C=1 if odd
26: c1       42     even: save  r1    # need a free reg, save r1
27: b1       43     ld    r0,r1  # r1=x
28: 91       44     shr   r1    # C is up if r1 is odd
29: c5       45     restore
2a: d7       46     rts
               47
               48     end

```

The program has the main part that calls two subroutines: **sum**, which computes the sum of two numbers given as memory addresses in **r0** and **r1** and stores the result in memory at the address given in **r2**; and **even**, which takes a number pointed to by **r0**, determines its parity, and raises the C flag if the number is odd, otherwise clears it. The purpose of the program is to compute the following expression given in pseudocode: if  $a + b$  is even then assign  $2b$  to **res** otherwise assign  $c - b$  to it.

Observe that the address of **b** is loaded in **r1** on line 5 in order for the subroutine **sum** to compute  $a + b$  (line 7). However, **b** is needed later either on line 16 or 18, where the result is formed following the subroutine

call on line 14, which determines which of those two instructions will be executed. Consequently, the writer of this example assumed that the value placed in `r1` on line 5 will persist in the register until line 15 despite the fact that two subroutines are called in that segment of code and both use `r1` for their purposes. The reason why the programmer was able to make this assumption is precisely the register discipline adhered to by all subroutines in the program: both `sum` and `even` save `r1` before writing into it, and restore it before passing the control back to the caller (lines 31/38 and 42/45).

### 6.3.2 Dirty subroutines.

As a final word, let us remark on an alternative register discipline. For a machine that has very few registers, such as ours, and for an application that uses several small-size subroutines, it may be more efficient in terms of memory space utilisation to assume the opposite extreme: that all registers may change after the return from a subroutine. One could call this discipline “dirty subroutine” to differentiate it from the “clean subroutine” discipline of the above example. This, naturally, makes the subroutines smaller since they no longer need to take care of the registers that they write into. What it also does is force the main program not to keep any important data in registers. Such data will be saved in memory (or upon the stack; according to the LIFO principle any stack usage by the subroutine for whatever purpose will be invisible to the main program) and loaded in registers when needed, certainly re-loaded after a subroutine call. Interestingly, if the algorithm is such that the results of the computation often have to be saved in memory anyway (due to the insufficient number of registers available), then the dirty subroutine discipline wins. The programmer should decide which discipline fits best the application she is working on. However, there is a further complication when using dirty subroutines. It is often required that one programmer prepares a set of subroutines and another, or others, uses them for a particular application. If that is the case, the subroutine designer must assume one discipline or the other, and it is impossible to make a choice if the overall application design is unknown. That is why the more universal clean discipline usually wins despite the greater cost: it is unconditionally safe to use.

## \*6.4 Recursion, stack memory and coroutines

This section focuses on the no man’s land between the low-level architectural world of platforms and the high-level abstractions entertained by programming languages and paradigms. We do not feel fully comfortable there because an exhaustive treatment of the issues raised in this section would lead us too far away from our main pursuit: exploration of platform hierarchies and their attendant cost analysis. However, we cannot fully excuse ourselves from studying them because they lead to particular *platformic* solutions that are interesting and elegant. As a compromise, a brief incursion into this contested territory has been decided upon, but only as an advanced section to be skipped in the first reading.

The questions raised here are as follows:

- Can a subroutine call itself?
- Can subroutines work cooperatively?
- Can subroutines be passed down as parameters to other subroutines?
- Can subroutines share memory space for their internal data?

All those questions have positive answers, which are discussed, explained and exemplified below.

### 6.4.1 Recursive subroutines

Any serious algorithm always depends on repetition. Usually repeated parts of a computation are supported in a program by a loop of some kind. Our thinking is well aligned with this form of repetition as it is to do with breaking the problem down into steps and applying a sequence of steps again and again until some condition is satisfied. This is the kind of repetition that we use outside the digital domain as well, so it is natural to us. Think, for example, of polishing a shoe by repeatedly brushing it until it shines, or building a wall from bricks until it is tall enough.

Repetition of this kind is a direct, prescriptive recipe: do this and then do it again until  $x$  becomes true (or while  $x$  remains true). There is another form of repetition which is induced by observation rather than prescription. This form is typically applicable to *self-similar* objects and the observation is one of self-similarity of the object as well as the problem being solved on the object.

This class of objects is also broader than the world of platforms. Its members are objects whose part is *of the same kind* as the whole. For example, a sequence is self-similar in that if we remove the first member of a sequence, the remaining part is a sequence in its own right. A mathematical set is self-similar for the same reason: if we remove part of a set, what remains is still a set. Closer to home, most linked data structures display strong self-similarity: a (linked) list of integers is a pointer to a record that contains another pointer to a record of the same kind. In other words, the tail of a list is a list in its own right.

A self-similar data structure supports a general “divide-and-conquer” solving technique, which can be described as follows. Imagine you are to determine a certain characteristic of a self-similar object  $x$ , e.g. the number of elements, the maximum element, etc. Call this characteristic  $P(x)$ . Then apply the following test:

1. Can you give the solution for  $P(x_0)$ , where  $x_0$  is the smallest (typically indivisible) object of the same kind as  $x$  without needing repetition?
2. For a larger object, split the object into a small part  $t$  and “the rest”  $r$ . Thanks to self-similarity  $r$  is of the same kind as the  $x$ . Assume that the problem has been solved for  $r$ , so  $P(r)$  is known.
3. Now can you express the solution for  $P(x)$  in terms of  $t$  and  $P(r)$  in closed form without needing any kind of repetition?

If your answer to questions 1 and 3 is positive, you are in luck. There is a simple and elegant solution method for your problem that does not involve loops but which can be achieved by using subroutines alone — without explicit repetition. Before we proceed to the method (which by the way is called *recursion*), let us apply the above 4-point test to some obvious problems so as to understand it better.

- Maximum of a sequence of numbers

1. The smallest sequence of numbers for which the problem makes sense is a sequence of 1 number. This number is obviously the maximum of the sequence as well.
2. Given a sequence  $x = x_0, x_1, x_2, \dots, x_n$ ,  $t = x_0$  and  $r = x_1, x_2, \dots, x_n$
3. 
$$P(x) = \begin{cases} t, & \text{if } t > P(r) \\ P(r), & \text{otherwise} \end{cases}$$

- Length of a linked list

1. The smallest list is the empty list. Its length is 0.
2. A list is a pointer  $x$  to a record, which has some payload field  $y$  and a pointer to the rest of the list  $r$ .
3. 
$$\text{length}(x) = \text{length}(r) + 1$$

- Does a given set of numbers contain 0?

1. If  $x = \{z\}$ , a one-element set,  $P(\{z\}) = (z = 0)$
2. A larger set  $x$  can be split thus :  $x = \{t\} \cup r$ , where  $t \notin r$ .
3. 
$$P(x) = (t = 0) \vee P(r)$$

etc. Notice that in the last example, the split of the  $x$  into  $t$  and  $r$  is not even unique: you can chose whatever element you like, and the result is still valid — that is how generic the method is.

Now the trick. Point 3 in each of these examples represents a *recursive definition* of a solution method. On the one hand, this is an equation that states that our problem is self-similar in that its solution can be expressed in terms of one for a smaller object. On the other, and this may not be immediately obvious, it actually provides the solution itself (or at least *a* solution).

Indeed if we imagine a subroutine  $P(x)$  for solving our problem what it could do is to:

- check whether the object  $x$  is a smallest possible object of its kind, and if it is, compute the solution as per point 1 above and return
- otherwise, split the object into  $t$  and  $r$ ,
- call *itself* on  $r$
- upon return compute the solution as point 3 above and return

We should not be too surprised that a subroutine “calls itself”. What it does is in fact call *a copy* of itself, which shares the code with the original. What the copy must not do is overwrite any data that forms the state of the original (registers, memory, etc) other than by way of returning the result. That way the effect of calling the copy is indistinguishable from that of calling the original, since the machine instructions that make up a subroutine do not get modified in any way by running a subroutine and since it does not matter whether the starting address of a subroutine is different from that of the caller or not. So if we understand how a call from  $A$  to  $B$  works, it requires little imagination to see that a call from  $A$  to  $A$  (a so-called recursive call) works the same way provided that  $A$  is perfectly clean (i.e. does not modifying local memory either). For a subroutine that only uses registers to do its work, perfectly clean just means clean. If a subroutine is too complex and needs memory to do its computation, section 6.4.2 below describes stack memory which can be used safely for this purpose for ordinary and recursive calls alike.

Recursive computations easily merit a whole separate textbook as they are the basis of the area of computer science known as *functional programming*. There is a large platform-related part of that area, too, which we could venture into, but again, not in an introductory text. Instead we will finish this section with an example of a recursive subroutine, this time taking a look at one that actively *modifies* the self-similar object passed down to it as a parameter (figure 6.2).

First the problem. Imagine a linked list of integers which contains zeros. There can be subsequences of 0’s (also called runs of 0’s) in the list sequence, e.g. the list [1,9,0,0,5,6] has such a run (two 0’s one after another), and there can be more than one such run. Problem: write a recursive subroutine that replaces every run of zeros in the list by a single zero.

A non-recursive solution is in fact quite cumbersome. One can envisage a loop that advances a pointer along the list every time checking whether the pointer points at an element with a zero payload. If it does, an inner loop is activated, which runs a second pointer from that point in the list forward searching for a non-zero. Upon loop termination, the list segment is replaced by a single element after which the outer loop continues.

Our recursive subroutine **elz** is much more elegant. It follows the three-point method presented above. First, on lines 4–9 it tests if the list is the smallest list in which runs of 0’s can be eliminated. That is clearly the empty list which does not contain 0’s anyway.

Since the first test does not use registers none are saved/restored. Otherwise we push three registers onto the stack, lines 11–13. Our next step is to divide the list into head and tail, the head is pointed to by the parameter in **r3** and the tail is pointed to by a pointer at offset +1, lines 15–17.

Finally the third step of the method. We must call the subroutine on the tail of the list and construct the overall solution from its returned result and the head. However, we first check whether the tail is empty, lines 19–27, and if it is, the list is one-element and so it cannot contain a run of 0’s. After restoring the previously saved registers (lines 22–24) we return the list unchanged (lines 25–26).

At this point we know that the list is a list of two elements or more. So we call **elz** on the tail (line 29). When the subroutine returns the control, we assume that the tail has been processed, i.e. that it has no runs of 0’s in it. Now we examine if the first element of the tail and the head of list  $x$  are both zeros (lines 32–37). If so, the whole list has a single new run: a length-2 run of 0’s made up by the head of the list  $x$  and the first list element of the tail. To eliminate that is very easy: we simply return the tail and ignore the head (lines 38–40). Otherwise the whole list (head and tail) contain no runs of 0’s and we re-connect the head to the tail (lines 43–47). We only need **r3** to accomplish that, so the restoring of the rest of the registers and returning to the caller is done last (lines 49–51).

Notice that the subroutine **elz** eliminates all runs of 0’s in the list. It is easy to see that the subroutine calls itself the number of times corresponding to the length of the list, since it calls itself on the tail of  $x$ , the tail of the tail of  $x$ , etc. until it finally calls itself on a single-element list (or the empty list if  $x$  is one). After that, as many **rts**’s are executed. Since each call results in the pushing of three registers onto the stack, the stack needs to have  $3 \times n$  words of memory or more in reserve for the recursion to be processed safely.

```

1 elz:    # eliminate runs of 0's in list x.
2     # parameter: r3=list x, result returned in r0
3     # clean protocol
4     if
5         tst r3
6         is z           # empty list
7             clr r0      # make output list empty
8             rts          # if we are to eliminate runs of 0s in an empty l.
9         fi           # the result is an empty l.
10
11     push r1
12     push r2
13     push r3
14
15     ld r3,r2      # r2=head of list x
16     inc r3
17     ld r3,r3      # r3=tail
18
19     if           # tail is null, no runs, return r0=r3
20         tst r3
21         is z
22             pop r3
23             pop r2
24             pop r1
25             move r3,r0
26             rts
27         fi
28
29     jsr elz      # eliminate 0s in the tail, get new tail of x in r0
30     ld r0,r1      # r1=payload of first elem of tail
31
32     if
33         tst r2
34         is z, and
35             tst r1
36             is z
37             then
38                 # return new tail (elim 1 zero)
39                 # it is already in r0
40                 pop r3
41             else
42                 # nothing to eliminate
43                 pop r3      # r3 = list x
44                 inc r3      # r3->forward ref
45                 st r3,r0      # forward ref=new tail
46                 dec r3      # restore r3
47                 move r3,r0      # result in r0
48             fi
49             pop r2
50             pop r1
51             rts

```

Figure 6.2: Subroutine `elz`

This is, as far as computing platforms are concerned, the main disadvantage of recursion. It tends to be stack-hungry, pushing many times the memory footprint of the data structure it processes on the stack and requiring the stack to be large enough to accomodate that much data.

That is the reason why recursion for all its elegance and expressive power is confined to desktop and server applications where there are very significant amounts of memory available at low cost. Small and especially embedded systems typically invite a purely iterative (loop based) style of programming. Still, in some application areas, e.g. Artificial Intelligence, Computer Algebra, etc., algorithms are most naturally expressed in recursive terms and as platforms steadily become more and more powerful, the use of recursion becomes more widespread as well.

#### 6.4.2 Subroutine memory: static vs stack.

Subroutines are based on the presupposition that their code is to be used repeatedly. It often happens that the algorithm embodied by a subroutine computes the result based solely on the data passed down to it (those data objects are normally referred to as subroutine *parameters*. If that is the case, any “work” memory required for computing the result can be reused by other computations performed in the program. Indeed, no value in the `ds` areas of the subroutine can ever be used again by another call to the subroutine, since if it were, then the result of the call would depend on previous calls, which we have assumed not to be the case. Yet any `ds` in the program holds the space for the duration of program execution, not just a single subroutine call, increasing the size of the program’s memory footprint by the amount specified in the `ds`.

Consider the following mental experiment. Suppose a program has two subroutines, *A* and *B*, each requiring memory space for its computation. Suppose one is to find the greatest prime number not exceeding 30 using the Sieve of Eratosthenes algorithm, and the other one the parity of a 30 byte long Fibonacci number. Neither subroutine reads any values left by any of its previous invocations. Clearly when one is running the other one is not, and the data space can be reused safely. Moreover, there could be more than two such subroutines in the program, and it would be advantageous to utilise *all* their temporary storage every time *any of them is run*. This looks like a problem for which one memory area whose address is available to all subroutines could be a solution. That is unfortunately not the case due to the fact that subroutines have the ability to call other subroutines. Consequently the caller may require to hold part of the available temporary storage and so the callee would have to utilise only what is left, but what part is it?

It comes as no surprise that the solution lies — again! — with the stack. The stack is such a useful part of platform machinery that any modern computer architecture simply cannot but have one. All a subroutine needs to do is lower the stack pointer by the amount of memory required; use that memory for whatever purpose (and call other subroutines if necessary, it is safe to do so), and before returning the control to its caller, raise the stack pointer to where it was at the beginning.

It takes little thought to realise that this way several subroutines can safely share work memory with one another and automatically release it when no longer needed. These segments of memory allocated on the stack are consequently called a subroutine’s *activation record* to differentiate them from the subroutine’s *static memory*, which is the memory that a `ds` command permanently allocates to the subroutine. The term “record” is justified by the fact that generally stack memory is allocated for a group of objects local (or “private”) to the subroutine. There are typically several of them, they have known types and are treated as fields to a single record that contains all local objects (often called “local variables” in L4 programming languages). A subroutine may continue to use objects in static memory, but that is usually limited to values that need to persist between subroutine calls — otherwise the use of static memory cannot generally be justified.

There are two machine instructions in Platform 3 that support use of activation records.

```
addsp n
```

and

```
ldsa rn, offset
```

The instruction `addsp` adds the immediate operand to the stack pointer. The former is interpreted as a signed integer in 2's complement representation, so the stack pointer may be given either a positive or a negative increment. The instruction `ldsa` has an effect similar to `ldi` in that it loads a value into a register based on the immediate operand. The difference is that `rn` is loaded with the *sum* of the immediate operand and the current content of the SP. The meaning of the immediate operand is the *offset* to an item contained in the activation record. Given the size of each field of the activation record that offset can be calculated at program development time by summing up the sizes of all fields preceding the one of interest. When the activation record has only a single field, the offset is 0.

Here is an example of how a subroutine may use stack memory. The program below calls a subroutine and passes it a pointer to a NULL-terminated character string. The subroutine calculates the number of times each letter of the alphabet (ignoring the case) occurs in that string and finds the maximum number of occurrences, which it returns in register `r0`. The subroutine requires a table (an array of byte-size cells) to update 26 counters as it scans through the string examining each character in turn. The counters are initialised with zero and then incremented when the corresponding letter is encountered in the character string.

```

1           asect 0
2   # find how many times at most any letter occurs
3   # in a null-terminated string using a subroutine
4   # freq, parameter: r0->string; result returned in r0.
00: d0 05      5       ldi      r0,str          # str is the parameter
02: d6 11      6       jsr      freq
04: d4          7       halt
05: 61 62 72 61  8       str:    dc      "abrákadabra",0      # 5 a's, 2 b's, ...
09: 6b 61 64 61
0d: 62 72 61 00
                    9   # subroutine that finds max freq of any letter
                    10  # ignoring case and non-letters
                    11  # dirty reg discipline
                    12  #
                    13
                    14 freq:
                    15  # allocate stack memory
11: cc e6      16       addsp -26          # make a table for 26 letters of the alphabet
                    17
                    18  # initialise table with 0s
13: d2 1a      19       ldi      r2,26          # r2 holds the downcount
15: 3f          20       clr      r3            # r3 holds initial value
16: c9 00      21       lds a   r1,0          # r1->beginning of table
                    22       do
18: a7          23       st      r1,r3          # initialise current cell
19: 8d          24       inc      r1            # r1->next cell
1a: 8a          25       dec      r2            # decrement the counter
1b: e1 18      26       until eq          # if we have done it 26 times, stop
1d: c9 00      27       lds a   r1,0          # r1->beginning of table again
                    28
                    29  # scan the string and count each letter
                    30       while
1f: b3          31       ld      r0,r3          # r0->current letter, r3=current letter
20: 8c          32       inc      r0            # r0->next letter
21: 0f          33       tst      r3            # r3=NULL? (end of string?)
22: e0 38      34       stays ne          # no, continue
24: d2 df      35       ldi      r2,0b11011111 # clear bit 0x20 ...
26: 4b          36       and      r2,r3          # ... to make r3 content an upper case letter
                    37       if
27: d2 5a      38       ldi      r2,"Z"          #
29: 7e          39       cmp      r3,r2          #
2a: ec 36      40       is le, and        #
2c: d2 41      41       ldi      r2,"A"          # if "A" <= r3 <= "Z"...

```

```

2e: 7e          42      cmp r3,r2      #
2f: eb 36      43      is ge          #
44      then          # ... then
31: 3e          45      sub r3,r2      # r2 = (r3-"A"), table index
32: 16          46      add r1,r2      # r2 -> table[r3-"A"]
33: bb          47      ld  r2,r3      # \
34: 8f          48      inc r3          # - table[r3-"A"] += 1
35: ab          49      st   r2,r3      # /
50      fi
36: ee 1f      51      wend
52
53 # now the frequency table has been completed
54 # find the maximum count:
55
38: 30          56      clr  r0          # r0 will hold current maximum
39: d2 1a      57      ldi  r2,26      # r2 will hold downcount, r1->table still
58      do
3b: b7          59      ld   r1,r3      # r3 = current cell
3c: 8d          60      inc r1          # r1 -> next cell
61      if
3d: 7c          62      cmp r3,r0      # current cell ? current maximum
3e: ed 41      63      is gt
40: 0c          64      move r3,r0      # >, therefore current max=current cell
65      fi
41: 8a          66      dec r2          # count down
42: e1 3b      67      until eq      # to zero
68
69      addsp 26      # deallocate stack memory
46: d7          70      rts           # return to caller
71      end

```

Clearly the table of counters is only required temporarily: every time the subroutine is called, there is a different string to scan and the old counter values are of no use. Stack memory is ideally suited to this, and all it takes to manage it is an allocation instruction on line 16 and a deallocation instruction on line 69.

What if the activation record is contains more than a single field? Perhaps more than one array and some smaller structures need to be accommodated on the stack. When using *static* memory, this is taken care of by the assembler; it calculates individual addresses and substitutes them for symbolic names. Now it seems we have lost this service because our addresses are not fixed, they are relative to the stack pointer and the assembler does not seem to be able to help. Or is it?

In fact even with stack memory we can use symbolic names, only in this case for offsets rather than addresses. For this purpose the CdM-8 macroassembler supports a special type of section, called *template*, which is introduced by the pseudo-instruction **tplate**:

<b>tplate name</b>
--------------------

A template is a *named* absolute section that starts at 0, does *not* allocate any memory, is accessible in the whole source file, but does not contain any entries for the linker to use. The section's text can not be interrupted and continued later (unlike an **rsect**, see next section).

Each label defined within a template is absolute and must be referenced using the prefix *name.*; for example, if the following template (lines 1–5):

```

1      tplate foo
00:    2      dc    "abcde"
05:    3 a:   ds    13
12:    4      dc    "this is it"
1c:    5 b:   ds    7
23:    6

```

```

    7      asect  0
00: d1 05  8      ldi    r1,foo.a
02: d2 1c  9      ldi    r2,foo.b
04: d3 23 10      ldi    r3,foo._
               11      end

```

is included in a program, the names `foo.a` and `foo.b` become available in it, with values `0x05` and `0x1c`, respectively. Notice that `dc`'s are allowed in a template for the purposes of offset calculation, but the memory content defined in them is ignored but for its size. The size of the whole data collection declared in a template is also accessible via the name `foo._`.

One can easily see a template like the one above used to define the structure of a stack area that contains several variables, and then reference an individual field by `ldsa`'ing the field name in a register. Code written in this style is easier to modify if later on it turns out that an extra item is required in stack memory: this can be taken care of by editing the template alone.

**Using stack memory for parameter/result passing.** So far we have passed parameters to a subroutine by placing them in registers and in each of those cases the returned value (if any) was passed back in a register. Clearly this can only be done in special cases when the subroutine being called has few parameters and when those are of the size that fits in a register. Naturally, this restriction can be lessened or entirely overcome by using pointers. Those may point at data structures of arbitrary size and complexity while remaining one byte in size. Such a solution has a disadvantage that the memory allocated for the parameters is held and cannot be reused after subroutine return, as we discussed earlier. The logical solution to this is to use instead the activation record (for parameters and returned values) while allowing the callee to extend it. Here is an example.

```

1      asect 0
2      # compute a+b-c in 16 bits
3
4      ldi    r0,a1
5      ld     r0,r0
6      push   r0
7
8      ldi    r0,a0
9      ld     r0,r0
10     push   r0
11
12     ldi    r0,b1
13     ld     r0,r0
14     push   r0
15
16     ldi    r0,b0
17     ld     r0,r0
18     push   r0          # SP-> b,a (big endian)
19
20     jsr    add16
21     addsp  2          # deallocate b
22                           # a+b stored where we had pushed a
23     clr    r2          # need 0 for 16 bit neg
24
25     ldi    r0,c0
26     ld     r0,r0
27     ldi    r1,c1
28     ld     r1,r1
29
30     not   r0          # invert higher-order byte
31     neg   r1          # negate lower-order byte
32     addc  r2,r0        # add carry-out
33

```

```

1e: c1      34      push    r1
1f: c0      35      push    r0          # SP->(-c),(a+b) (big endian)
36
20: d6 35  37      jsr     add16      # compute a+b-c
22: cc 02  38      addsp   2          # deallocate (-c)
39
24: c4      40      pop     r0          #Â store a+b-c in res
25: d1 33  41      ldi     r1, res0
27: a4      42      st      r1,r0
28: d1 34  43      ldi     r1, res1
2a: c4      44      pop     r0
2b: a4      45      st      r1,r0
2c: d4      46      halt
47
2d: 04      48      a0:    dc      0x04
2e: 22      49      a1:    dc      0x22      # a=0x0422
2f: 05      50      b0:    dc      0x05
30: 13      51      b1:    dc      0x13      # b=0x0513
31: 06      52      c0:    dc      0x06
32: 38      53      c1:    dc      0x38      # c=0x0638
54
33: 00      55      res0:  ds      1
34: 00      56      res1:  ds      1
57
58      add16: #####      # add 16-bit numbers X,Y, res->Y
59          # add 16-bit numbers X,Y, res->Y
60          #Â dirty protocol
61
35: c8 01  62      ldsa   r0,add16_pars.X0
37: b0      63      ld     r0,r0
64
38: c9 02  65      ldsa   r1,add16_pars.X1
3a: b5      66      ld     r1,r1
67
3b: ca 03  68      ldsa   r2,add16_pars.Y0
3d: ba      69      ld     r2,r2
70
3e: cb 04  71      ldsa   r3,add16_pars.Y1
40: bf      72      ld     r3,r3
73
41: 17      74      add    r1,r3
42: 22      75      addc   r0,r2
76
43: c8 03  77      ldsa   r0,add16_pars.Y0
45: a2      78      st     r0,r2
79
46: c8 04  80      ldsa   r0,add16_pars.Y1
48: a3      81      st     r0,r3
49: d7      82      rts
83
84          #Â pars passed via stack
85          tplate add16_pars
00:        86      retadr: ds     1
01:        87      X0:    ds     1
02:        88      X1:    ds     1
03:        89      Y0:    ds     1
04:        90      Y1:    ds     1
91          end

```

This program computes the expression `a+b-c`, where all three numbers are 16-bit integers in big-endian 2's complement representation. The computation is done by repeatedly calling the subroutine `add16`. The subroutine adds two 16-bit big-endian numbers, its parameters, `X` and `Y`. The parameters are pushed onto the stack by the caller and the result replaces `Y` on the stack. Notice that this subroutine does not require stack memory for itself, so it does not allocate an activation record.

Lines 85–91 show the template used to access the parameters on the stack. When the subroutine takes the control, the `SP` points at the return address, above which the higher- and lower-order `X` and `Y` is placed.

On lines 4–18 `a` and `b` are pushed onto the stack and then the subroutine is called for the first time. Line 21 discards `b` and the sum remains at the top of the stack. Following this, lines 30–35 change the sign of `c` and push it onto the stack. Now `a+b` and `-c` are on top of the stack and another `jsr` is issued on line 37. After discarding `-c` (line 38), the result is ready for storing at the top of the stack.

The subroutine is presented on lines 58–82 and is pretty straightforward. The two parameters are read into two pairs of registers, after which they are added and the result is saved in `Y`.

A few concluding remarks. First of all, the subroutine may require stack memory for temporary objects *as well* as parameter/result passing. Naturally, this is possible by issuing a matching pair of `addsp` inside the subroutine. If there are several local objects that need to be placed in stack memory, the template can be extended by adding declarations before the first line (line 86). The resulting template will describe a memory layout which is partly the subroutine's activation record (lines preceding the field `retaddr`) and partly parameters (lines succeeding the field `retaddr`). Nothing will change as far as parameter passing, since the offset to the parameters will remain consistent with the template: the offset and the stack pointer will be added the same values except for the opposite signs, the former due to the declaration of local variables amounting to some  $k$  bytes, and the latter due to the `addsp` with  $k$  as the immediate operand.

It should also be noted that the allocation and deallocation using the instruction `addsp` is shared between the caller and the callee. The callee allocates and deallocates space for its activation record, and the caller only needs to deallocate space used by the parameters/result of the computation. That space can only be allocated by the caller and the callee cannot even deallocate it until after it returns the control, at which point it is no longer active anyway.

**General arrangements for calling a subroutine.** Figure 6.3 sketches an example of quite general subroutine-call arrangements. Here two templates are used: one, `mysub_params`, for defining the parameters and the result of calling a subroutine `mysub`, and another, `mysub_data`, represents the subroutine's view on the parameters/result and local data (here, `x` and `y`). Notice that the parameter part of `mysub_data` corresponds to the parameter/result template `mysub_params`. This correspondence is fundamental: in any programming language, including Levels 4 and above, the caller's and the callee's interpretation of the data passed to and from the subroutine must be consistent. One might wonder whether such consistency can be established automatically, thus avoiding the repetition of the same declarations twice. The answer is that, as will be shown in chapter 7, the caller and the callee are quite often compiled separately, in which case the compiler needs to see those declarations in both sources. That does not automatically mean the same text should be written twice by the programmer, since one file can merely be *included* in two different programs. Those as well as many other issues of software engineering do not interest us from the platforms' point of view, we leave them at that.

Continuing with the figure we note the use of the template size `(_)` in the first `addsp` as well as the minus before the template reference. Both are allowed syntactically and are quite convenient for moving the stack pointer in the right direction by an automatically calculated distance. Once this happens, subroutine parameters can be treated the same as static data (except, of course, having to use `ldsa` instead of `ldi`) which continues even after the subroutine has been called and returned the control. That is how an arbitrary-sized result may be passed back to the caller. Finally the whole `mysub_param` record is deallocated by the second `addsp`. The workings of the subroutine itself should be straightforward after the previous example.

**Further stack-related instructions.** For the sake of completeness let us list here the rest of the stack-related instructions. They are not generally required for writing application programs, but may be of use in special circumstances.

<code>setspl addr</code>
--------------------------

```

        asect 0
# caller
...
addsp  -mysub_params._          # allocate parameters
ldsa   r1,mysub_params.p1
...
ldsa   r2,mysub_params.p2
...
jsr    mysub
ldsa   r1,mysub_params.res
...
# process returned result using r1 as pointer
addsp mysub_params._          # discard pars and result

...
# callee
mysub:
    addsp  -mysub_data.rtadr      # allocate x,y
    ...
    ldsr  r0,mysub_data.p1       #Â r0 points to p1
    ...
    ldsr  r0,mysub_data.p1       #Â r0 points to p2
    ...
    ldsr  r0,mysub_data.y        #Â r0 points to y
    ...

    addsp mysub_data.rtaddr      # discard x,y
    rts

    tplate mysub_data.rtadr
x:    ds     2
y:    ds     10

rtadr:ds      1

p1:    ds     1
p2:    ds     3
res:   ds     5

    tplate mysub_params
p1    ds     1
p2    ds     3
res:   ds     5
    end

```

Figure 6.3: Stack memory usage for calling a subroutine

This instruction sets the SP to the value given by the immediate operand *addr*, which must be a number, but can be presented in any format. For example,

```
setsp -16
```

loads 0xf0 in SP, this making the top 16 memory cells available for non-stack data (or code).

```
pushall
```

This instruction pushes registers from 3 to 0 (in this order) onto the stack. It is equivalent to

```
push r3  
push r2  
push r1  
push r0
```

There is a matching `popall` instruction as well:

```
popall
```

which is equivalent to:

```
pop r0  
pop r1  
pop r2  
pop r3
```

Interestingly, two push instructions take the same time to execute as one `pushall`, and take twice as much memory space. It follows that in most cases if a subroutine wishes to follow a clean protocol, and needs more than one register, the pair `pushall/ popall` makes more sense than a repeated push/pop.

#### 6.4.3 Passing a subroutine as a parameter

Quite often an algorithm implemented as a subroutine is generic enough to work for a number of cases. When one case differs from another by some data object's value, the object can be passed down to the subroutine as a parameter (by providing in a register the value or a reference to it if the value does not fit in the register size). Some times the cases to which the algorithm is applicable differ by a *function* rather than a data value as such. For example, the simplest algorithm of finding the maximum of an array is as follows

```
*** array_max ***  
initialise x with the first element of the array  
do  
    read next array element  
    if it is GREATER than x, set x to that element  
until all elements have been read
```

Notice that the adjective GREATER is capitalised to indicate that the algorithm does not care what it means. For two unsigned numbers, it would mean comparing them as binary digit-strings, digit by digit, left to right. For two numbers in 2's complement representation, the comparison is based on the sign bit of the difference. If it is the absolute values that we compare, this involves making the numbers positive first and then comparing them as unsigned. Array elements may not even be numbers as long as we know the size of each. It would be convenient to use GREATER as a subroutine-parameter of the subroutine `array_max`.

There is a software engineering aspect to this problem. The implementation of the comparison may be prepared by one programmer, and the subroutine for maximisation (and not necessarily as simple as the one above) by another. There is the freedom of what to maximise (an array, a list, a set, etc. of items) and there

is the freedom of *how* to compare items in the process of maximisation (depending on their nature). Object-orientated programming and functional programming offer different (but quite powerful) tools that ensure that interfaces between the two pieces of code can be assuredly consistent and in some sense provably correct. None of it concerns us, the students of platforms. What we are interested in is the low-level mechanisms that support the very idea of an algorithm parametrised by a subroutine.

CdM-8 Platform 3<sup>1/2</sup> has an instruction that makes it possible to call a subroutine at an address contained in a register:

Instruction	Synopsis	Formula	Flags Affected
jsrr rn	jump to subroutine [pointed to by] rn	push PC; PC:=rn	none

**Example.** Let us take a closer look at the example of maximisation. It can be implemented using `jsrr` :

```

1  asect 0
2      # main prog: find the maximum of an array by calling subroutine max
3      # giving it a subroutine-parameter in r2
4
5      # find the maximum in the sense of SIGNED numbers
6      ldi r0,array
7      ldi r1,8          # size of array
8      ldi r2,take_greater
9      jsr max
10     ldi r0,max_signed
11     st r0,r3
12
13     # find the maximum in the sense of UNSIGNED numbers
14     ldi r0,array
15     ldi r1,8          # size of array
16     ldi r2,take_higher
17     jsr max
18     ldi r0,max_unsigned
19     st r0,r3
20     halt
21
22 # subroutine max. dirty protocol
23 max:    ld r0,r3      # r3 holds current max
24     do
25     save r1
26     ld r0,r1
27     inc r0
28     jsrr r2    # clean subroutine: if r1 ">" r3 then r3:=r1
29     restore
30     dec r1
31     until z
32     rts
33
34 take_greater:
35     if
36         cmp r1,r3
37         is gt
38         move r1,r3
39     fi
40     rts
41
42 take_higher:
43     if
44         cmp r1,r3
45         is hi

```

```

46      move r1,r3
47      fi
48      rts
49
50 array: dc 2,8,3,6,-1,2,5,9
51 max_signed:
52     ds 1
53 max_unsigned:
54     ds 1
55 end

```

The main program computes two maxima of the example size-8 `array`: one in the sense of signed numbers (the answer is 9), and one as unsigned numbers (the answer is 255). The former is done on lines 5–11 and the latter on lines 13–19 by calling the same subroutine `max`. In the former case it is given the address of the subroutine `take_greater` and in the latter case `take_higher`. The subroutine `max` itself (lines 23–32) is straightforward, and it uses `jsrr` to call the subroutine whose address is passed down to it in `r2`.

Notice that the actual passing of the parameter-subroutine in our example was implemented in the simplest possible fashion, by placing its starting address in a register just before calling subroutine `max`. Everything that was said in the previous section about passing parameters via the stack applies here just as well. A subroutine parameter can be made a field of the activation record, since all such fields can be completely arbitrary objects, including a pointer to a subroutine.

#### 6.4.4 Coroutines

Subroutines are a master-slave structuring mechanism whereby the parameters of a service are defined by the caller and the subroutine itself is a slave that provides that service. In some situations, in particular when a producer-consumer relationship exists between two branches of an algorithm, the interaction between two encapsulated entities is required to be much more symmetric. For example the producer (i.e., the provider of some data) may determine at some point that the data is complete and should be consumed by the other branch, which requires a subroutine call, but equally the consumer may consume the data and in the process of doing so may discover that more data is needed before any result is returnable. So in a way the consumer has to “call back” the producer for some service, requiring the control to be returned to the consumer after that service has been provided — so that the consumer may continue to work on the original call.

In a way the producer and consumer call *each other* as a subroutine (the control is returned to the caller upon completion of the service requested), that is why the actors engaging in this control pattern are called *coroutines*, see figure 6.4. Notice that if A called B as a subroutine and B called A as a subroutine as well, this would cause an indirect recursion, which is an entirely different control pattern: a *subroutine* always executes from the top, whereas coroutines allow each other to *continue* execution from the point at which it was suspended.

Notice that a coroutine is *also* a subroutine. When it is first called, the technique is the same as that of subroutine calling: the return address is left on top of the stack and the control is transferred to the starting point of the coroutine. When the coroutine has completed all its work it executes a normal subroutine return at which point the continuation data disappears — the job is done. What makes a coroutine different is its ability to *yield* to another coroutine, i.e. to allow it to continue from the last point that it had the control at, while at the same time saving its own current address for subsequent continuation.

There can possibly be more than two coroutines. The control ping-pong shown in figure 6.4 is more complex in such a situation; a coroutine could in principle choose which other coroutine to yield to. Normally this involves a special intermediary, a coroutine usually called *dispatcher*. All coroutines yield to it, and the dispatcher “decides” which other coroutine to re-activate by yielding to it. Complex *cooperative multitasking* schemes of this nature are beyond the scope of the present book; they belong in the area of parallel computing, which is rather advanced.

The implementation of the yield instruction (and yes, it is a single machine instruction in most platforms) is as follows:

exchange the content of the PC with the top of the stack

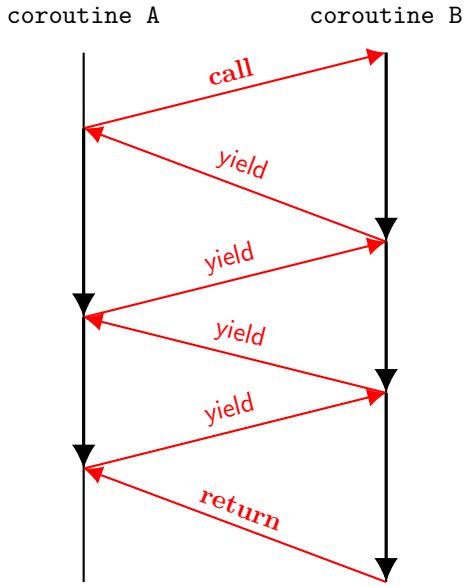


Figure 6.4: Coroutines

Indeed, the initial call to a coroutine from another leaves the continuation address on top of the stack. Exchanging the top of the stack with the current continuation address (the content of the PC) makes the coroutine transfer the control to the continuation of its caller whilst at the same time leaving its own continuation address on top of the stack. One can think of `yield` as a two-phase transaction: first pop the continuation address off the stack, then push the content of the PC onto the stack. Even though the effect of this sequence is exactly the same as exchanging the top of the stack and the PC, in reality the platform ensures that the action of a `yield` is *indivisible*. Nothing can happen between the imaginary pop and push. But what could possibly happen? As we will see in section 13.5 much later, the platform's environment has the ability to interrupt the execution of a program, utilising the stack for this purpose. The details of such events will be discussed in due course; for now let us just say that the `yield` operation can never be interrupted. In CdM-8 Platform 3 `yield` is implemented as a single machine instruction

`crc`, a mnemonic for *CoRoutine Call*.

Let us now turn to an example coroutine application.

**Comma separated values (csv).** A fundamental representation of a table (including active tables such as spreadsheets) is a string of comma separated values each represented as text. A sequence of csv strings (or csv lines terminated by the newline symbol) looks like general text, but it could be converted back to the table it represents.

The process of converting abstract values to text and back is well understood. We ourselves spent some time in the earlier chapters converting between different representations of numbers, and strings of any digits are easily converted to text as well. Yet there is an interesting structural problem specific to csv, which is as follows. A text representation of an abstract object may contain one or more commas, which makes a comma-separated list of values ambiguous as far as its structure. Does a given comma separate two items on the list, or is it part of an item? The answer depends on the representation of items, and we wish to have a solution that works for any representation, or better still, which does not depend on it at all.

The first thing that comes to mind is to delimit each item on a comma-separated list by enclosing it in quotes:

"Joe Bloggs", "Adams, Duglas", "George Brown, Jr.", ...

This does away with the comma identification problem, but introduces a new one: what if the textual representation of an object has quotes in it? For example,

"Joe Bloggs", "General James "Mad Dog" Mattis", "George Brown, Jr.", ...

Now the middle item has quotes which can be interpreted as a closing and an opening one, or as the opening and a closing one. Of course one could think of replacing quotes by asymmetric characters, such as the grave ‘ and the apostrophe ’, but that complicates matters even further. A simple solution is to “escape” each quote inside an item with a special character, for example the backslash \. This way a quote immediately following a backslash will be interpreted as an internal quote, and a quote not following a backslash as an item delimiter. Well, what about an item requiring a backslash as part of it? Would that backslash not be confused with the one that escapes the quote? Not necessarily. Let us agree that *any* character following a backslash is interpreted as this character in a non-special sense, i.e. belonging to an item. So if we needed a backslash in the item representation, we would replace it with \\\. And obviously if we wanted \" to be part of an item, it would be represented as \\\\".

The above example string would look like

```
"Joe Bloggs", "General James \"Mad Dog\" Mattis", "George Brown, Jr.", ...
```

according to the rules. Notice that we could relax our rules a little: items that contain neither commas nor quotes in them could still be written unquoted. By looking at the first character after the comma, the reader would immediately and reliably determine whether the item is presented as plain or quoted/escaped.

Now the problem: how do we write a set of subroutines that convert a single csv string into an array of pointers to NULL-terminated strings representing each item?

**Implementation.** We see a solution as a combination of a main program and two routines: `copy` and `parse`. Coroutine `copy` partitions the original csv string into a sequence of NULL-terminated strings by replacing commas by NULLs, one item at a time. Coroutine `parse` copies an item from the source string to its internal memory buffer, removing the quotes and converting escaped characters on the fly. Both routines yield to the main program to get the next character. The main program is responsible for supplying one and is at liberty to do ancillary work in the process: at the very least to check if the first character is a quote and call `parse` rather than `copy`, but maybe also removing white space after a comma, etc.

Here is the main program:

```

1 asect 0
2     ldi r1, line    # r1 scans the input text string
3     ldi r2, ptrs    # r2 scans array of pointers to result strings
4     while
5         ld r1,r0
6         tst r0
7         stays ne
8         if
9             ldi r3,0x22      # 0x22 is ASCII for quote "
10            cmp r0,r3
11            is eq
12            jsr parse
13            else
14            jsr copy
15            fi
16            do
17                inc r1
18                ld r1,r0
19                crc
20                tst r0
21                until ne
22                st r2,r0
23                inc r2
24            wend
25            halt
26 #           joe,mary,"john donne,Sr",betty<NULL>
27 line:   dc "joe,mary,", 0x22,"john donne,Sr",0x22, ",betty",0
28 ptrs:   ds 10

```

On lines 12 and 14 the initial calls of the subroutines `parse` and `copy`, respectively. is made depending on the initial character of an item. The subroutine yields rather than returning, after which the until-loop (lines 16–21) repeatedly reads the next character from the csv line and yields to the coroutine (which one depends on the previous `if`). The coroutine responds with either a null-pointer (if it needs more input) or a valid pointer to a NULL-terminated string containing the textual representation of the current item. The main program and the coroutines adopt a dirty protocol whereby `r3` is recognised as a work register and need not be saved/restored. Registers `r0` and `r1` are used cooperatively and register `r2` is private and needs to be saved/restored if used.

Here is the coroutine `copy`:

```

1      # r0 = current character, when coroutine yields, r0=0 signals "give me more"
2      # ... otherwise r0->result string and the coroutine ends (returns)
3      # r1-> current character
4
5  copy:      # r3 under dirty protocol
6      ldi r3,ptrc          # save ->beginning of item
7      st r3,r1
8      crc                  # next char
9
10     while
11         tst r0            # while not end-of-line
12         stays ne
13         if
14             ldi r3 ","
15             cmp r0,r3
16             is eq
17                 clr r0
18                 st r1,r0
19                 crc          # consume comma
20                 break
21         fi
22         clr r0
23         crc
24     wend
25     ldi r0,ptrc
26     ld r0,r0
27     rts
28 ptrc:   ds      1

```

When the coroutine is called for the first time, it saves the pointer to the beginning of the current item in `ptrc` and yield to the main program (lines 6–8). The coroutine is initialised and primed for cooperation. As soon as the control returns, if the character received from the main is a comma (lines 13–16) then overwrite it with NULL (lines 17–18) and signals to the main to read next character, i.e. skip the comma discovered (line 19), after which the search terminates. If, otherwise, the received character is not a NULL then skip it and continue on the loop. The loop terminates when we reach the csv line NULL terminator, if no comma is encountered. Finally, retrieve the pointer to the beginning of the current item `ptrc`, load it in `r0` and `rts` back to the main; the job is done.

Finally the coroutine `parse`:

```

1  parse: # r3 under dirty protocol
2      # r0 = current character, when coroutine yields, r0=0 signals "give me more"
3      # ... otherwise r0->result string and the coroutine ends (returns)
4      # r1-> current character
5
6      # initialise:
7      push r2
8      ldi r2,ptr
9      ld r2,r2
10     ldi r3,res

```

```

11      st r3,r2
12      pop r2
13      crc      # yield to coroutine
14
15      while
16          ldi r3,0x22  # 0x22 is ASCII for quote "
17          cmp r0,r3
18          stays ne
19          if
20              ldi r3,0x5c      # 0x5c is ACII for backslash \
21              cmp r0,r3
22              is eq
23              clr r0
24              crc           # just accept next char without checking
25          fi
26
27          ldi r3,ptr
28          ld  r3,r3        # read ptr
29          st r3,r0        # store current char at ptr
30          inc r3         # adv ptr
31          ldi r0,ptr
32          st r0,r3        # save ptr
33
34          clr r0         # skip to next char
35          crc
36      wend
37
38      clr r0
39      crc           # skip closing quote
40      clr r0
41      crc           # skip ","
42
43      ldi r3,ptr
44      ld  r3,r3
45      clr r0
46      st r3,r0        # store NULL at ptr
47
48      inc r3         # adv ptr
49      ldi r0,ptr      #           and
50      st r0,r3        #           store it
51
52      ldi r0,res      # get res ptr
53      ld  r0,r0      #           into r0
54      rts            # job done, no more coroutining
55
56  ptr:          dc buffer
57  res:          dc 0
58  buffer: ds 30

```

The coroutine uses a persistent pointer `ptr` which points to somewhere within its internal `bbuffer`. Initially it points to the first byte of the `buffer`. When called for the first time, the coroutine copies `ptr` to `res` to save the pointer to the beginning of the free space in the buffer (lines 3–8). From that address a copy of the current item will be stored in memory. The initialisation ends by yielding to the main (line 9).

The rest of the work is done in a while-loop (lines 15–36). The loop continues while the current character is not a quote. If the character is a backslash it is ignored but the main is yielded to again to read the next character directly (lines 29–35) without checking whether it is a quote, etc. That next character is stored in the `buffer` according to `ptr`, and the latter is advanced and saved (lines 27–32). Finally the coroutine signals that it needs a new character (line 34) and yields to the main for it, concluding the loop body.

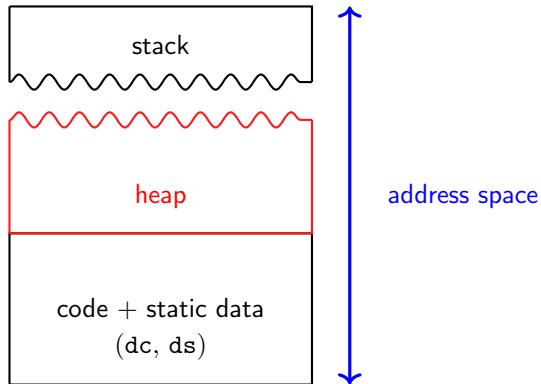


Figure 6.5: Placement of the heap

We leave the loop when the second quote is encountered, by which time the whole item is copied to the `buffer` with any escaped characters properly dealt with. Lines 43–46 take care of the NULL termination of the item string, lines 48–50 advance the `ptr` and finally lines 52–54 ensure that the coroutine executes a return without continuation.

## \*6.5 Heap

Stack memory works fine for scoped access, where a fixed amount of memory is allocated for a while and then deallocated/freed when no longer needed. This is quite sufficient for programs that allocate individual arrays and records for use in subroutines, but the LIFO discipline imposed by the stack proves too restrictive when memory needs to be allocated and deallocated to support expansion and contraction of linked data structures, which are data dependent and can happen at any point in the program.

Consider for example a subroutine that produces a list of unique English words that occur in a long character string passed down to it as a parameter. First of all, it is unclear how many elements the result list will have. Also, the subroutine has to allocate memory for individual character strings representing unique words, and it is not immediately clear how many words and how long they may turn out to be. The programmer cannot even estimate and maximise those characteristics for the purpose of memory allocation, since such estimates cannot be relied upon in the absence of text statistics. Also, the memory allocated for the answer must survive the return from the subroutine, since the list in question is the subroutine's result. That memory can only be deallocated when the result is fully consumed, e.g. when the five longest words are found on the list by a different subroutine.

Neither of the memory mechanisms we have seen so far (static memory, stack memory) can provide an efficient solution. We need an altogether different memory mechanism. Let us start with

### 6.5.1 Requirements

The program resides in the low addresses and the stack grows upside-down from the top of the address space. Between the highest allocated address of the program and the lowest anticipated address of the stack lies a range of addresses that are not yet allocated to anything. What we would like to do is reserve a range of addresses in that area for a special memory pool, which we, following the convention, will call the *heap*, fig 6.5. It is something new to us in that it is a proper *active object* of the kind found in modern high-level programming languages. Not only does it have storage allocated to it, it also provides a set of pre-implemented subroutines via which that storage is managed. In fact those subroutines can be left *abstract*, i.e. defined by their behaviour and interface (parameters they need and the result being returned) without worrying about *how* exactly their functionality is implemented. For us, students of computing platforms, the fact that they are implementable must be sufficient, since the specific implementation will vary from platform to platform. For our platform of choice a useable implementation is, sadly, quite impossible due to the restriction of the address space to a meagre 256 bytes, including the program, static data, stack and any pre-implemented subroutines.

That will not deter us, since the specification of heap memory is interesting by itself as it involves cost considerations (which are fundamentally “platformic”) and new algorithms. Here is a list of subroutines that we require:

**allocate(size)** This subroutine has one parameter: the size of the data object to be allocated in heap memory. The subroutine returns a pointer to a memory area of `size` bytes allocated for the data object. If the returned value is a null-pointer, then there is not enough heap memory left to allocate an object of the requested size.

**free(pointer)** This subroutine deallocates (frees) the memory occupied by the object pointed to by the `pointer`. A call to `free` is only legal if the parameter `pointer` is the value returned by a previous call to `allocate`. The subroutine `free` itself does not return a result.

The above two are all we need besides the assurances that we can use them repeatedly without any restrictions save for the total available amount of heap memory (which may also vary in the course of program execution). For the avoidance of doubt and for the purposes of formal proof of correctness the following additional requirements are often made explicit:

1. It is guaranteed that if  $p$  is the pointer returned in response for `allocate( $x$ )`, then
  - (i)  $L \leq p < H$ , where  $L$  and  $H$  are the low and high ends of the heap memory address range
  - (ii) no subsequent call to `allocate` can return a pointer  $p'$  in the range  $p \leq p' < p + x$ , until the subroutine `free` is called with the value  $p$  as its parameter.
2. if at some point in time all non-null pointer values ever returned by `allocate` have been passed to `free` as a parameter, in other words, if all allocated memory is subsequently deallocated, the heap becomes *empty* and will respond to the next call to `allocate` as if it were the very first call. In particular, the maximum available size would be exactly as it was when the heap was first used.

The requirement 1(i) means that the heap only provides a legal allocation; 1(ii) that the allocated segments of memory never overlap; and requirement 2 means that heap memory does not *leak*, i.e. *all* allocated memory can eventually be deallocated and used again.

Memory leak is a well-known hazard in systems fundamentally based on heap memory (which is to say, nearly all of them). Often this happens because the program “forgets” (i.e. overwrites or unlinks from a linked data structure) some of its pointers received from `allocate`. The corresponding data objects become completely inaccessible; a side effect of it is that they cannot be given as a parameter to `free`, so the memory allocated for them becomes lost for the rest of the execution time. Requirement 2 states that the correct heap object should not leak by itself (even though the program using it might leak heap memory).

Notice that our requirements are quite loose. They do not guarantee that *all* available heap memory can be allocated; the subroutine `allocate` can refuse to allocate a data object (by returning a null pointer) even when the sum of the sizes of all currently allocated objects is significantly less than the amount of heap memory currently available. None of the above requirements forbids that. Nor do they prescribe how data objects should be placed in the heap memory range. Both issues belong to the domain of memory allocation algorithms; there are several available and they differ in the amount of computation they require and the proportion of the heap address space they reserve for themselves. There is a time cost to allocation; and there is a space cost to the auxiliary infrastructure built around the allocated segments to manage them. In the next section we will take a cursory look at one of the practical algorithms.

### 6.5.2 Buddy allocation algorithm

This is not the most economical allocation algorithm in terms of memory space but it is one of the fastest ones. It is based on a simple and elegant idea of space partitioning by dividing in half. Let us assume, for simplicity, that the heap memory address range spans exactly one kilobyte, i.e. 1024 bytes<sup>2</sup>. The allocation algorithm allocates space in the form of contiguous blocks of various sizes. The blocks have a size that is a

---

<sup>2</sup>The figure these days tends to be expressed in gigabytes, rather than kilobytes, but even this tiny heap is 4 times the amount of memory available to a Cdm-8 platform.

```

*** allocate(s), s is the size in bytes ***
up := True
rank_of_s := the smallest k such that s ≤ M × 2k
k=rank_of_s
repeat
    <<<<<
    if k=rank_of_s AND directory[k] is not empty
        remove the first element from directory[k], which contains an offset T
        return the value H+T, where H is the starting address of the heap.
        add pair (H+T,rank_of_s) to the alloc list.
    if k=r_max
        return null pointer (insufficient memory)
    if up
        k:=k+1
        if directory[k] is not empty
            up := false
    if not up
        remove the first element from directory[k], which contains an offset T
        directory[k-1]:=[T,T+M × 2k-1]
        k:=k-1
    >>>>>

```

Figure 6.6: Buddy algorithm: allocation

power of 2, from some minimum size, say 64 bytes up to the whole heap, in our case 1024 bytes. So in our example the heap can have up to 16 64-byte blocks.

Consequently, blocks can be size-64, -128, -256, -512 and -1024. The smallest ones are called rank-0 blocks. A block twice as large is rank-1, four times as large rank-2, etc, until the maximum rank  $r_{\max}$ , in our example,  $r_{\max} = 4$ , which corresponds to the whole heap as a single block. We will denote the minimum block size as  $M$ . In our example  $M = 64$ .

Notice that if we split a rank- $k$  block (where  $0 < k \leq r_{\max}$ ) down the middle, the result will be two adjacent rank- $(k - 1)$  blocks. A pair of blocks that together cover the footprint of a single block of the next rank, are called *buddies*. The buddy algorithm regards heap memory as a collection of blocks produced by repeatedly splitting the rank- $r_{\max}$  block. Notice that apart from the largest block, which is always single, each block has a buddy.

The algorithm allocates a single block for an `allocate(x)` call by searching for a free (either deallocated earlier, or never allocated) rank- $r$  block, such that  $M \times 2^{r-1} < x \leq M \times 2^r$ , i.e. large enough to accommodate the request but not so large that, if split, either buddy would also be large enough. If such a block is not available it looks for the smallest available rank- $R$  block (any one of them if more than one is available), where  $R > r$ . If no such block is found, then the heap memory is out of space for allocations of size  $x$ . Otherwise the algorithm progressively splits the rank- $R$  block into halves, quarters, etc., until it gets two free rank- $r$  blocks, one of which is marked as occupied and the pointer to it is returned to the caller.

This is a crude approach, since the algorithm might allocate just under 50% more memory than necessary, but the advantage is that it *never* wastes more than 50%. Clearly because only free blocks are allocated and only ones that originate from the largest block, the requirements 1(i,ii) are satisfied.

To implement the allocation algorithm, it is convenient to arrange a size- $(r_{\max} + 1)$  array of lists, which we shall call the *heap directory*. Each array element corresponds to a specific rank, from 0 to  $r_{\max}$ , and is a pointer to a linked list of offsets to free blocks of that rank. Initially, the heap directory in our example looks as follows:

rank	0	1	2	3	4
heap directory	[]	[]	[]	[]	[0]

```

*** free(p), p is a pointer to the block to be freed ***
find a pair (p,r) in the alloc list
if not found
    ERROR
    fail
remove (p,r) from alloc
T := p-H, where H is the starting address of the heap
k := r
repeat
    <<<<<
    v:=T/(M × 2k)
    if v is even
        buddy:=T+M × 2k
        B:=T
    else
        buddy:=T-M × 2k
        B := buddy
    found := False
    for each t in directory[k]:
        if t=buddy
            remove t from directory[k]
            T:=B
            k:=k+1
            found := True
            break
    if not found or k=r_max
        put T on directory[k]
        return
>>>>>

```

Figure 6.7: Buddy algorithm: deallocation

which means that there is a single rank-4 (1024 byte) free block with offset 0 from the beginning of the heap memory area. The buddy allocation method in algorithmic form is presented in figure 6.6. It is not very complicated. First of all, the target rank `rank_of_s` is determined and the algorithm is set in the upward direction by the variable `up` initialised to `True` and the current rank `k` is set to target. If the target rank has free blocks we remove one from the rank's list of free blocks, return it and terminate.

Otherwise the algorithm steps up until it finds availability, at which point the direction is switched to 'down' (`up` becomes false).

When the algorithm steps down, it takes one block from the current rank (which is guaranteed to have availability or else the direction would still be 'up'). It splits it in two and makes the free-block list for the previous rank out of them. Indeed the previous rank is guaranteed to have no availability (since we have passed it on the way up) so there is no need to extend the current list there, since it is empty.

The process terminates on the way up immediately if the target rank has availability; otherwise when the maximum rank has been reached, if there is no availability; otherwise on the way down when the algorithm reaches the target rank, which it definitely will since it stepped up from it. When a pointer to the newly allocated block is returned, that pointer value coupled with the target rank is also added to the allocation list `alloc`, which is empty initially. This is useful for deallocation, as it allows to check that the block was allocated in the first place, and since the rank of the block associated with the pointer can be retrieved as well.

**Freeing memory.** Where the buddy algorithms comes to its own is in *deallocating* space following a `free(p)`. Recall that `p` points to the block to be freed. The trick is that since every block has a buddy, except the largest block, which represents the whole heap memory, the subroutine `free` can check whether the buddy of `p` is free as well. If it is, the subroutine merges the two buddies back into the block of the next rank from which they were made when first allocated. The subroutine then checks whether the buddy of that larger block is free, and if so, merges it in as well. The procedure is repeated until a higher-rank buddy happens not to be free, or when the maximum rank is reached. In the latter case (as well as when the largest block of the heap is freed), the heap becomes completely empty, i.e. comes back to its initial condition. This takes care of requirement 2 (no leaks). The algorithm for deallocation is shown in figure 6.7.

It first checks the validity of the pointer by looking it up on the list of allocated blocks `alloc`, at the same time determining the corresponding block's rank `r` and then offset `H`. The variable `buddy` holds the offset to the beginning of the current block's buddy. It is calculated based on the parity of `T` in units of the current block size  $M \times 2^k$ . If the parity is even, the parameter block is the low half of its immediate progenitor, otherwise it is the upper half. The offset of the next rank block is stored in the variable `B`. The rest of the algorithm is straightforward.

**Example.** We continue with our small heap, which we assume to be 1024 bytes in size placed at address 100000 decimal. Figures 6.8 and 6.9 represent the history of calling heap management subroutines a total of 12 times. The first four data objects allocated are A, B, C and D sized 30, 200, 70 and 80 bytes, respectively. First of all, the buddy algorithm splits the biggest block into five: a rank-3, a rank-2 and a rank-1 block, and also two zero-ranked blocks. One of those rank-0 blocks is returned for use as A, and the rest are entered in the heap directory. Data object B grabs the rank-2 block, C the rank-1 block and then the allocation of D causes a further split. A free rank-3 block with offset 512 is divided into one rank-2 and two rank-1 blocks, one of the latter being returned for use as D.

Now we have the first deallocation. We free C, which causes the rank-1 directory list to expand as it now has two vacant blocks: the one with offset 128 (which was allocated to C) before, and the buddy of D, offset 640. They are not buddies to each other, so cannot be merged.

Finally E grabs the only vacant rank-0 block and proceed to figure 6.9.

At this point we free A, which gives us one vacant rank-0 block (offset 0), and B, which goes to the rank-2 list, where another block (offset 768) has been staying since we split the initial block of the heap. They are not adjacent and cannot be merged.

Now we grab the free rank-0 block for F, and deallocate D. Now rank-2 blocks with offsets 512 and 640 can merge to produce a rank-3 block with offset 512. We deallocate E, creating a free buddy for the block currently held by F, and then deallocate F which precipitates a cascade of mergers: two rank-0 blocks into a rank-1, two rank-1 blocks into a rank-2, etc., until two rank-3 blocks merge back to the original single rank-4 block.

Figure 6.8: Operation of heap memory (Part 1)

Figure 6.9: Operation of heap memory (Part 2)

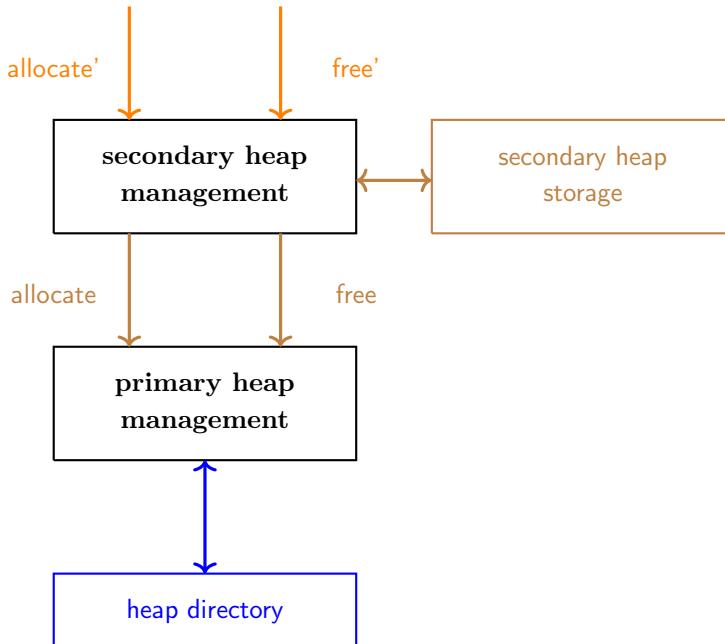


Figure 6.10: Multilevel heap

**Cost analysis** In our analysis of platforms we are always cognisant of the solution cost. Proper complexity analysis goes far beyond this book, but some simple considerations of speed and space are inescapable when considering a technical solution and especially when more than one is available.

We have presented an approach to “dynamic” memory, i.e. a storage system which does not require the knowledge of data object sizes prior to use, and which permits an arbitrary order of memory allocation/deallocation between several data objects. Two questions remain unanswered:

1. How fast does the algorithm find a free block if one exists?
2. How much memory does it require *for itself*, i.e. for accommodating its internal data structures, such as the heap directory.

Let us start with the speed question. In order to allocate memory for an object, the algorithm has to find a block of the right size and if one is not available, keep looking for blocks larger than what is required. Each availability check costs a fixed number of instructions; those are the instructions for reading one array element from the directory array and checking whether it is a null pointer. On the way down the algorithm also spends a fixed number of instructions to split a single block into two and place them on the directory at the previous index. Consequently, the amount of time required to allocate a block is proportional to the distance in ranks between the allocation request and the smallest available block. At most this distance is the number of ranks supported by the heap, or the binary logarithm of the ratio of the heap size to the smallest block size. For a large heap (gigabytes) the unit of allocation is typically not too small either, perhaps kilobytes, which gives a 1M ratio and approximately  $20 \times 2 = 40$  steps (stepping up and down), which works out as a few thousand machine instructions. More importantly, if we wished to use the buddy algorithm to allocate individual bytes, it would only slow us down by 50%! Indeed the difference between kilobytes and bytes is a factor of  $2^{10}$  which is 10 more steps in either direction for the algorithm bringing the total to just 60 steps.

Mathematicians and computer scientists know that the logarithm is an extremely *slow* function of its argument. The cost of buddy allocation is logarithmic in heap size, which means that the algorithm is in fact quite fast for large and small amounts of memory alike.

Deallocation also involves search but this time we are searching for the buddy at a specific directory index. The length of the list of free blocks linked to the directory entry can be as long as the maximum number of blocks, which is again the ratio that we keep coming back to, and which we assumed to be at most 1M. Traversing a list of this length would indeed be prohibitively expensive; one might even think that this is the

price we have to pay for the fast allocation process. However, the list does not have to be a list, since the order of the elements (offsets to free blocks) is not important. Consequently, the algorithm can utilise a better data structure that supports fast search. A good book on algorithms and data structures will immediately point to several; one example is a binary tree, in which the search in most situations is logarithmic in cost, and in which the space overhead (additional memory required to accommodate the binary tree infrastructure) is about additional 50% compared to the list. We shall not dwell on this any further.

Now to the rest of the space overheads. As we already mentioned, the size of the directory array is logarithmic in the size of the heap (because it corresponds to the number of supported block ranks). We can reserve only 100 array elements and be perfectly sure that this would be sufficient for 100 ranks, which would require  $2^{100}$  words of heap memory, which is more than currently available on this planet counting all forms of digital storage. We conclude that the storage requirements of the directory are negligible.

There is also an allocation list, which keeps record of all active allocations. We must not see the space taken by this list as a *platform* overhead, since the length of the list is proportional to the number of blocks held by the *client* program. In principle, it is possible to leave the information about the rank and perhaps even some validation bit strings right in the block provided to the client in response to the `allocation` call. Often the space for this is sliced off the beginning of the block by advancing the returned pointer and using negative offsets to store that information before returning the control to the caller. That way the caller will not be able to legitimately overwrite the element of the allocation list. The subroutine `free` can back up its pointer parameter and read the rank, etc. from the hidden part of the block. In this approach the overhead of the allocation list is at most one or two machine words per allocated block, i.e. perfectly negligible and not dependent on the size of the heap. Note that if the allocation list is distributed over the heap blocks as suggested, the search for the rank information and the pointer validation of the subroutine `free` are extremely fast (a few instructions) and independent of the heap size.

Finally, the availability lists pointed to by the directory elements are the main space cost. The maximum number of available blocks is achieved when all of them are rank-0. For a heap with  $M=1K$  and 1GB heap memory, there can be at most a million free blocks. So we require the same number of list elements kept in reserve in case we need them for the heap directory. In an address-consistent 64-bit machine (a modern desk-top computer) a list element takes two 64-bit words, or 16 bytes. Consequently, we must suffer a 16M byte space overhead if we are to use a fine-grained (i.e. minimum 1K allocation) gigabyte heap. That causes less than 2% of the space to be lost. Two per cent corresponds to the worst-case allocation, but if it is affordable, the space can be used immediately with zero time penalty. If reservation of list space is undesirable, the alternative is for the subroutine `allocate` to request blocks for use as list storage when and as needed, for the subroutine `free` to deallocate them when all list elements in such blocks have been deleted.

**Multilevel heap.** The above analysis brings us to an important observation. Just as platforms can be arranged in a hierarchy of levels using virtualisation mechanisms, such as interpretation, so can parts of a platform. This is especially true of all kinds of memory; indeed these days any computer user would have heard of *main memory* and *primary* and *secondary caches* even though not everybody immediately recognises a multilevel organisation there. Each level of memory is an abstraction that holds complexity underneath and offers services to the level above. Let us turn to figure 6.10 where an example of such an arrangement is sketched out.

One of the disadvantages of the buddy algorithm is its relatively high space overhead if the requested allocation size is almost a power of two multiple of  $M$  or, worse still, it is much smaller than  $M$ . Another disadvantage is the flip side of the main advantage: the ability to accommodate a variety of allocation sizes. The latter necessitate search, which would not be needed for repeated allocation of same-size data objects. Consequently a heap based on the buddy algorithm is far from ideal for allocating masses of small objects, for example, list elements or records used in more complex linked data structures. They tend to be much smaller than  $M$  and of a fixed, predetermined size, thus hitting both weak points of buddy allocation simultaneously.

The solution could be in terms of a two-level heap. At the bottom level we place the classical buddy algorithm with relatively large blocks, say 1MB for a 1GB heap. This would require an 11 word directory array and no more than 1000 list elements for keeping tabs on available blocks. This virtually nullifies the space overheads of heap management in relative terms. We will call this a *primary* heap. It is completely generic, good to use for allocating any large objects with unpredictable size and lifetime and can be utilised by the client program directly.

On top of the primary heap let us organise a heap for small fixed-size objects, which we will call the *secondary* heap. Without going into too many details, imagine a pair of subroutines `allocate'` and `free'` that operate as follows. Using the primary heap's `allocate` the secondary heap management requests a large-size block initially, which it fills in with a linked list of fixed-size *slots*. The slot size can be known in advance, or a whole block of them can be prepared when the first slot of a given size is requested by the client. Either way, given the linked list, a slot can be allocated by the secondary subroutine `allocate'` at a cost of a few instructions (deleting a slot from the linked list and returning its address to the client), and it can be similarly rapidly deallocated. Compare that with the unavoidable search in the primary heap and you will see that any application that involves frequent use of the heap for processing linked data structures (typically every few tens of machine instructions for most kind of uses) will suffer hardly any speed degradation due to the secondary heap. By contrast primary heap allocation for those purposes in the absence of a multilevel heap organisation would dominate the execution time.

If one large block received from primary `allocate` proves insufficient, further blocks can be requested and linked in. Different size slots would need to be kept in different large blocks, but that is the trade-off between speed, memory overhead and flexibility that the multilevel memory introduces. Also, deallocating a slot may cause the containing large block of them to become empty, at which point the block itself can be deallocated using the primary heap subroutine `free`. However, there is no mechanism in the secondary heap for avoiding holding several nearly empty large blocks before any one of them is completely empty and can be recycled elsewhere by the primary heap. Again, this shows the trade-off between the high leak-resistance of the primary heap and the superior speed and memory efficiency of the secondary one.

# Virtualisation

## 7.1 Object modules and linking

So far we have regarded subroutine calls and returns from a subroutine purely as a control mechanism, not dissimilar to conditionals and loops. Such a view is justified by the nature of `jsr` as an instruction that passes the control to an identified point. That point in the program also marks a *reusable* code segment, i.e. a piece of code which can be used repeatedly on different data at different points in the calling program. The nature of a conventional machine instruction is very similar to this: an instruction is an action that can be used repeatedly on different data at different points in the program. In this sense a subroutine can be regarded as a rather complex virtual instruction. Similar to a real platform instruction, a subroutine has an op-code (i.e. its starting address or a label associated with it) and “operands”, i.e. subroutine parameters received under some calling convention. From this perspective, subroutines realise an *intensive* virtualisation mechanism. Indeed, the only extensive part, which has to be programmed every time a subroutine is called, is the parameter passing preparation and the `jsr` for the transfers of control. The former could be very low-cost (here we mean execution time) if parameters are passed using a constant array of pointers: this requires perhaps a few instructions to arrange on the part of the caller and a similar number for parameter access inside the subroutine. The cost of the `jsr/rts` pair is exactly two instructions. If the subroutine realises a substantial algorithm (i.e. executes tens of instructions or more), the cost of calling it is, in fact, negligible.

In order to use a subroutine for virtualisation, one would need to decouple its body from the rest of the program and make it part of a new platform. The platform in question extends the one on which the program is supposed to run by offering the program an assortment of readily available subroutines that it *may* call. The subroutines that *are* called are included in the program and *linked* with it automatically: by “included” we mean allocation of a fixed range of memory addresses to each subroutine that the program intends to call, and by “linking” we refer to the process of informing the program before execution about the actual addresses that must be used in the `jsr` instructions that do the calling. More generally, linking is the process of informing separately compiled parts of some of each other’s addresses, namely those associated with the labels that are used jointly.

When subroutines occur in an assembler source, the job of allocating memory for machine instructions and assigning specific addresses to any labels is done by the assembler. If a subroutine is to be included in the platform rather than program, it needs to be assembled separately and stored in compiled form ready to be allocated and linked<sup>1</sup>.

We have now identified the following technological requirements:

1. Technology to compile an assembly language program part by part (this process is called *separate compilation*).
2. A mechanism of assigning symbolic names (labels) to certain points in a part of the program and publishing them for use by other parts despite separate compilation.
3. A mechanism for adjusting the compiled machine code of a part to make it work correctly in a range of addresses unknown at compile time.

---

<sup>1</sup> Strictly speaking, that is not a necessary requirement, since virtualisation can conceivably be done by publishing the source of each platform subroutine and by getting the programmer to cut-and-paste it into her program, but that, besides being inconvenient and error-prone, would break the platform abstraction.

Let us consider these in detail. Since the assembler already supports labels as a form of symbolic address, and that is used broadly to label data in `dc` and space in `ds` pseudo-instructions, what is required is the ability to declare some of the labels *entry points*, or ents for short, and to allow them to survive compilation for use as addresses by other separately compiled parts. In other words, the result of compiling, which will be called the *object file* from now on, will, under separate compilation, have to include a list of ents with their associated addresses.

Here we encounter the first difficulty: the actual memory addresses of the entry points are unknown to the assembler, because it only compiles a single part of the program at a time and has no access to the rest. It is the totality of the parts that will eventually define which part goes where in memory, not a single part on its own. The only information about an ent that is available to the assembler is its position *relative* to other address in the same contiguous block of code. In other words, the assembler knows offsets associated with labels, not their actual memory addresses (ones we called ‘absolute’ addresses in aspects).

### 7.1.1 Relocatable sections

Consequently, it is common practice to arrange a separately compiled part of a program into a collection of *relocatable sections*, or r-sects for short. Each r-sect represents a contiguous address segment which is also named. R-sect names are used as anchoring points for addressing but are also useful for diagnostics purposes, i.e. to be able to identify a specific r-sect in which a linking error has occurred. Each r-sect employs a *relative address space*, which starts from 0. Any label in that address space is categorised as *relocatable*, i.e. one to be adjusted when the r-sect is finally allocated a specific range of memory addresses. In the process of allocation, the starting memory address of the range is added to every relative address in the section to obtain its actual address in memory.

Externally visible labels, the ents, are included in the object file as a table listing labels against relative addresses in their r-sect. If a part contains several r-sects, the object module will contain several such tables<sup>2</sup>.

Ents are syntactically defined as labels terminated by the angular bracket `>` rather than a colon. Here is an example of two r-sects in a part:

CdM-8 Assembler v2.1 <<<multdiv.asm>>> 13/08/2015 22:38:26

```

1 ##### section mul, contain 2 subs
2      rsect mul
3 mul>
4 #      computes product of r0 and r1, result goes in r1
5
00: c2       save r2
01: 3a       clr r2
             while
02: 00       tst r0
03: ed 09    stays gt
05: 16       add r1, r2
06: 88       dec r0
07: ee 02    wend
09: 09       move r2,r1
0a: c6       restore
0b: d7       rts
             17

```

---

<sup>2</sup> Absolute sections may also declare ents, but those are marked as absolute and are treated separately.

```

18  smul>
19  #      same as mul, but numbers are signed
0c: c2
0d: 3a
0e: 00
0f: ea 13
11: 84
12: 8e
13: 05
14: ea 18
16: 85
17: 8e
18: d6 00
1a: 9a
1b: e3 1e
1d: 85
1e: c6
1f: d9
00: c2
01: 02
02: 30
03: 85
04: 79
05: ed 0b
07: 16
08: 8c
09: ee 04
0b: 09
0c: c6
0d: d7
40
41
42 ##### section div, can be used on its own
43     rsect    div
44  div>  save r2
45  #      divides r0 by r1, quotient in r0, remainder in r1
46
47      move r0,r2
48      clr r0
49      neg r1
50      while
51      cmp r2,r1
52      stays gt
53      add r1,r2
54      inc r0
55      wend
56      move r2,r1
57      restore
58      rts
59      end
=====
```

#### SECTIONS:

Name	Size	Relocation offsets
mul	20	02 04 08 0e 10 15 19 1c
div	0e	02 06 0a

#### ENTRIES:

Section	Name/Offset
\$abs	<NONE>
mul	mul:00  smul:0c
div	div:00

The above is the assembler listing of a source file containing two r-sects, `mul` and `div`. A program that links to this part may reference one to three entries: `mul`, `smul` and/or `div`. The assembler session summary found between the double lines above shows the entries in the aforementioned form: the name of an r-sect coupled with a hexadecimal offset. Notice that the current address (shown at the beginning of the line in the assembler listing and separated from the rest of the line by a semicolon) drops from `1f` to `00` on line 43 as we enter a new r-sect. Also notice that the main program can limit its use of the other part by only referring to `div` or to one or both subroutines for multiplication. In such a case the memory needed to accommodate the unused r-sect will be available for other uses. It is, however, impossible to utilise the space allocated to `smul` if only unsigned multiplication is required, `mul`, since an r-sect can either be linked to the main program as a whole or not linked at all.

Another interesting feature of r-sects is the fact that the address constants in one contain relative addresses (for the simple reason that the eventual memory addresses are not known at the compilation stage). This means that the object file should include information about where these constants are located in the object file bit-string so that when the section is allocated space for, it can also be adjusted by adding to those constants in it the starting address of the section. This information is presented for the programmer's inspection in the assembler summary above, under the heading `SECTIONS`<sup>3</sup>. For example, the instruction `wend` in the section `mul` at relative addresses 07-08 includes a pointer to relative address 02. The pointer points to the `while` instruction and is used for closing the loop. However, the address 02 shown in it is a relative address. If the section is allocated at address, say, 34, the pointer should point at address  $34+2 = 36$ . The adjustment will be done at the allocation/linking stage using the information provided by the assembler in the object file, which is also shown in the assembler summary. The process of adjusting the pointers after allocating a range of addresses for an r-sect is called *pointer relocation*, or just relocation for short. Let us now turn to a separately compiled main program, the user of the above subroutines. Here is one example of how it can be written:

```
CdM-8 Assembler v2.1 <<<test.asm>>> 18/06/2017 21:58:41
```

```

1 # compute -3x+7,
2      asect 0
3 smul:   ext          # declare smul as an external label
4                      # to be defined by an ent elsewhere
00: d0 0b      5 ldi    r0,x
02: b0         6 ld     r0,r0
03: d1 fd      7 ldi    r1,-3
05: d6 00      8 jsr    smul
07: d0 07      9 ldi    r0,7
09: 11         10 add   r0,r1
0a: d4         11 halt
0b: 11         12 x:    dc    17      # example value for testing
13             end
=====
```

#### SECTIONS:

Name	Size	Relocation offsets
------	------	--------------------

#### ENTRIES:

Section	Name/Offset
\$abs	<NONE>

#### EXTERNALS:

Name	Used in
smul	\$abs+06

---

<sup>3</sup> Bear in mind that all addresses and data displayed by the assembler in addition to the source code are presented in hex

---

Notice the declaration of `smul` as an external symbolic address. On line 8 it is used by a `jsr` to jump to the corresponding subroutine and at this stage the label is compiled as `0x00`, since the assembler does not know the actual address of the subroutine. Interestingly, `cocas` fully supports external labels, even in the offset form, for example `foo+3`. The offset is communicated to the linker, which takes it into account in determining the absolute address that should be substituted for the offset label.

### 7.1.2 Linking object files

Finally, let us look into how linking is done *across* object files. As we already mentioned, on line 3 above the label `smul` is declared as external. Recall that it is also an *ent* that belongs to the r-sect `mul` included in the other separately compiled part. Having made the declaration, the main program uses the label for calling a subroutine in that separately compiled part, see line 8. The assembler summary records that usage: in the absolute section at address 06 the external `smul` is used. This location will be linked.

Now let us see how the linking is realised. The platform's toolkit includes the object linker called `cocol`, which implements allocation and linking. The linker first allocates space for all absolute sections found in the object files given to it, according to the numerical addresses specified in them. It checks that the allocation is conflict-free along the way (i.e., that the same address is not claimed by more than one absolute section). As soon as the allocation is finished, the linker is left with a set of holes in the memory space, i.e. memory not allocated yet and consequently available for further allocation. The holes are used to accommodate any r-sects.

The linker does not know at this point which of the supplied r-sects are required for the application. In order to determine that, the linker builds a link relation  $L$ , which is a set of pairs  $(x, y)$ , where  $x$  and  $y$  are section names. The absolute section<sup>4</sup> is given a special name (which cannot clash with any other as it violates the naming rules of `cocas`) `$abs`. For any  $x, y$ , the pair  $(x, y)$  is included in the link relation  $L$  if and only if the r-sect  $x$  contains an ext which is defined as an ent in the r-sect  $y$ . In other words,  $x$  is link-related to  $y$  if  $x$  "calls"  $y$ .

To determine which r-sects must be included, the linker first builds the relation  $L^*$ , the transitive closure<sup>5</sup> of  $L$ . The latter contains all pairs of r-sects that are related directly or indirectly, via intermediate r-sects, for instance: if  $x$  is link-related to  $z$  and  $z$  to  $y$ , then the linker considers  $x$  and  $y$   $L^*$ -related, etc. Finally the linker identifies all r-sects that the absolute section is  $L^*$ -related to. **Only** those r-sects end up being allocated space for and linked.

The linker produces a report for the programmer if the `-l` option is specified. Here is the report on the application we are using in this section as a running example:

```
$ cocol -l test multdiv

SECTION '$abs'
    ENTRY points

SECTION 'mul' from file:muldiv.obj
    ALLOCATION start: 0c size: 20
    ENTRY points
        'mul': 0c
        'smul': 18

SECTION 'div' from muldiv.obj
    << not deployed >>

ABSOLUTE SEGMENTS ALLOCATED:

from file: test.obj start: 00 size:0c
```

<sup>4</sup> Strictly speaking there could be any number of absolute sections, since the `asect addr` pseudo-instruction introduces a contiguous section of code starting at the absolute address `addr`. However, for the purposes of linking all asects are regarded as a single noncontiguous absolute section. The fact that that section is not contiguous does not present a problem since asects do not require relocation.

<sup>5</sup> We refer the reader to the theory of relations found in any good textbook on discrete mathematics

The report is self-explanatory. Notice how the linker found a suitable allocation for one of the r-sects, while leaving the other, `div`, unallocated, since no reference to its ents has been detected by the assembler and passed to the linker in an object file. Also notice the physical addresses determined for the ents.

To conclude this section, let us restate a few subtle points, which assembler programmers often forget about.

- An r-sect is an autonomous unit, which does not “see” other r-sects even if they are included in the same file. If the instruction `rsect name` is used repeatedly in the same separately compiled part, the chunks of code following that instruction (until the start of another r-sect or an absolute section) are concatenated by the assembler, which produces a single r-sect named *name*. If an r-sect with the same name occurs in a different separately compiled part, the linker will consider this a name clash and will complain.
- R-sect names are not labels. They consequently cannot be used for subroutine calls and they do not clash with ordinary labels, hence it is quite possible (and was indeed the case in our example earlier on) to give an ent in an r-sect the same name as the r-sect itself.
- Since every r-sect is autonomous, it is not sufficient to declare a certain name to be an ext in *some* section. Every ext used in a section **must** be declared in *this particular* section by being placed as a label to the `ext` pseudo-instruction, otherwise an error of “label not found” would arise. The reason for this requirement is that if a label is used in a section but it does not label any instruction in it, it is possible that the label was, in fact, mistyped, rather than being one that genuinely represents an ext.
- The use of the angular bracket to identify ents in a section is necessary: the assembler has no way of knowing which of the labels are meant to be externalised and which are for the section’s use only. The angular bracket makes it certain.

### 7.1.3 Structure of object file

At this point we must remind the reader yet again that the CdM-8 platform was constructed for teaching purposes and so it makes certain efficiency sacrifices to aid learning. The object file in any practical system is a complex binary structure. That is, the object file is a data structure represented as a bit-string, bit-sliced and stored in a file. It is possible to examine the bit string to see important features of the data structure that it represents, but it is not easy. CdM-8 object files are in fact text files that represent the bit-string in question in structured text<sup>6</sup>. As a representation, this form is in one-to-one correspondence with the intended original, but it does not require any tools other than a text editor to see what is inside the file.

Naturally, an object file represented as text is at least twice as large as the bit-string that contains all the information. Indeed, a byte requires two hexadecimal digits, which, represented as text, take a character each. Since characters require a byte each to be represented in ASCII, this results in 2 bytes per byte of original data. As a matter of fact, the ratio is worse: the text is structured for ease of reading: bytes are separated by spaces and punctuation; new lines and textual headers are added to mark structural units inside the data structure, etc. All of that is avoidable and the linker would be better served by a binary encoding of the object file as a variant record, see section \*3.15. Yet given the smallness of any potential outcome (256 bytes only are available on the platform for any linked program, and consequently no more than that for an individual object file), the efficiency of tools and file storage is not at all important.

**Object file records** The object file is represented as a sequence of *textual records*<sup>7</sup>, which can be examined by a human. Each record is a single line of text and has a category, which is indicated by the first four characters of the line. The categories are:

**‘ABS\_’** Memory content. The record has the format `ABS  aa:  data`, where *data* is a space-separated list of hex pairs defining the memory content byte-by-byte starting from the absolute address *aa*.

**‘NTRY’** Entry declaration. The record has the format `NTRY  label  aa` and declares *label* to be an entry in the current section. The hex pair *aa* defines the absolute (if the current section is an `asect`) or relative (if the current section is an `rsect`) address of the *label*. The relative address is counted from the lowest location of its section.

<sup>6</sup> Needless to say — but we shall nevertheless! — that text in its turn is represented as a bit-string, but that representation is for the platform’s use, not human inspection

<sup>7</sup> not to be confused with records as standard data structures

**'NAME'** Declares the start of an **rsect**. The record has the format **NAME<sub>U</sub>id**, where *id* is the name of the section.

**'DATA'** Defines the memory image of the current **rsect**. The record has the format **DATA<sub>U</sub>data**, where *data* is a space-separated list of hex pairs.

**'REL<sub>U</sub>'** Declares the locations within the current **rsect** that require adjustment, i.e. to which the actual memory address of the start of the section must be added before run-time. The format is **REL<sub>UU</sub>data**, where *data* is a space-separated list of hex pairs defining those locations relative to the start of the section.

**'XTRN'** Declaration of an external label. The record has the format **XTRN<sub>U</sub>label:<sub>U</sub>xlst**, where *xlst* is a space-separated list of space-separated *id-adr* pairs where *id* is a section name and *adr* the address within the section (relative to its starting point) at which the external reference occurs. When the linker determines the actual memory location of the entry *label* it adds this value to every *adr* in the section *id* mentioned in *xlst*. The reason why that value is added rather than copied is because the assembler allows external references to be offset by a constant and it is that constant that it will place in the corresponding location in the object code. If the external reference occurs in an **asect**, the special name **\$abs** is used for *id* to indicate that *adr* is an absolute address.

Here is the object file produced by the assembler for the main part **test.asm** shown earlier:

```
ABS 00: d0 0b b0 d1 fd d6 00 d0 07 11 d4 11
XTRN smul: $abs 06
```

and this is the same for the file **multdiv.asm**

```
NAME mul
DATA c2 3a 00 ed 09 16 88 ee 02 09 c6 d7 c2 3a 00 ea
13 84 8e 05 ea 18 85 8e d6 00 9a e3 1e 85 c6 d9
REL 04 08 10 15 19 1c
NTRY mul 00
NTRY smul 0c
NAME div
DATA c2 02 30 85 79 ed 0b 16 8c ee 04 09 c6 d7
REL 06 0a
NTRY div 00
```

#### 7.1.4 Memory image file

The purpose of a linker is to allocate space and link up a program. More specifically, the **CdM-8** linker must determine the initial content of each of the 256 bytes of memory when the program is loaded in it. We have described in detail what is involved in allocating and linking each section. When the job of the linker is done, it records the result in a file with the **.img** extension, which has the following structure:

**Line 1:** contains the string **v2.0 raw**. This is a marker that indicates that what follows on the subsequent lines is a Cdm-8 memory image.

**Line *k*,  $2 \leq k \leq 257$ :** two hex digits representing the content of the memory cell at address *k-2*.

For the same reasons that CdM-8 object files are text, the memory image is also encoded as text (and of course a larger platform would use binary files for storage efficiency). A memory image file is always 257 lines long. Any memory that is not allocated to a section is automatically filled with zeros by the linker. In a larger system the unused memory is simply not included in the memory image file.<sup>8</sup> Here is the image file the linker created for our example program. Since the file has very little content in each of the lines, we

<sup>8</sup> which, obviously, requires the latter to have some structure in order to define which ranges of addresses are unused when the program is not contiguous in memory.

show it against line numbers in table form:

N	content	N	content	N	content	N	content	N	content
1	v2.0 raw	11	11	21	ee	31	84	41	e3
2	d0	12	d4	22	0e	32	8e	42	2a
3	0b	13	11	23	09	33	05	43	85
4	b0	14	c2	24	c6	34	ea	44	c6
5	d1	15	3a	25	d7	35	24	45	d9
6	fd	16	00	26	c2	36	85	46	00
7	d6	17	ed	27	3a	37	8e	47	00
8	18	18	15	28	00	38	d6	48	00
9	d0	19	16	29	ea	39	0c	49	00
10	07	20	88	30	1f	40	9a	50	00

The rest of the lines contain 00.

## 7.2 Instruction aggregation

The diligent reader, who has solved a large number of the assembly language exercises that accompany the book, has no doubt developed by this point her favourite paraphernalia of code patterns. Those are “idioms” of the assembly language, or stable combinations of instructions that are used almost automatically by an experienced programmer faced with a standard task. Let us take a look at a tiny example.

Loading a byte from a labelled memory address, say `foo` into a register, say `r0` is usually a two-phase affair. First an `ldi` is employed to get the address into some register, and then an `ld` command is engaged to actually fetch the data. The register could be a different register from the destination of the data (if the address is to be reused soon) or it could be the same, thus reducing the pressure on the register set. So in our case we write:

```
ldi      r0, foo
ld      r0, r0
```

This is the kind of thing that one tends to write many times in a CdM-8 assembler program, yet it is a tiny pattern, only two instructions, and it does not merit virtualisation as a subroutine. There are two reasons for it; one is obvious: the overhead of the subroutine calling will exceed the economy of any kind; the other one is subtle: the parameters here are a *register* and a label. The latter can be easily made variable: one could think of introducing variable labels associated with ordinary labels by a pseudo-instruction. The former less so; a variable register cannot easily be envisaged in an ordinary assembler source due to the smallness of the register set. Instances of such a variable would potentially clash with registers assumed to hold important data.

Yet any stable pattern should be seen by a computer scientist as a clear call for virtualisation. There is a *single* action here, albeit an aggregate one, namely getting a value from `foo` to `r0`. That is how it is perceived by an algorithmically inclined mind, and that is how it should feature in the program if we can help it. The technology we seek is not procedural, it is not a lightweight subroutine that is required, rather a textual substitution. We wish to be able to write something like:

```
ldv      r0, foo
```

and get the assembler to produce the previous two-liner in response. Here `ldv` is a virtual op-code, and `r0` and `foo` are virtual operands. So if, alternatively, we wished to load the value of `bar` into `r1`, we would write

```
ldv      r1, bar
```

and get the assembler to substitute

```
ldi      r1, bar
ld      r1, r1
```

for it. Clearly we have two requirements: the ability to define a pattern and the ability to configure it using a certain number of parameters. The CdM-8 assembler `cocas` allows the programmer to do just that, using

its *macro* facilities. A macro is an aggregate with parameters defining a virtual instruction implemented by substitution/configuration. In our example the following macro definition would solve the problem:

```
macro  ldv/2
    ldi    $1, $2
    ld     $1, $1
mend
```

Here the pseudo-instruction **macro** introduces the name of the aggregate, **ldv**, and the number of parameters used for its configuration, 2. The parameters are denoted as **\$1** , **\$2** , ... , **\$9**, so there can be at most 9 parameters. In practice a much smaller number, typically one or two, proves sufficient. Importantly the number of parameters is part of the macro definition. We could still define a version of the macro **ldv** with three parameters (and we will in a moment), and the assembler would treat it as completely separate macro despite the fact that it has the same name. Given the above macro definition, the (virtual) instruction

```
ldv    r3, rex
```

*expands*, i.e. is replaced by, to

```
ldi    r3, rex
ld    r3, r3
```

which is produced by substituting the macro body for the **ldv** and associating **\$1** with **r3** and **\$2** with **rex**. Now imagine that we wish to reserve the option of reusing the address that has been loaded first in order to get the value. We could introduce an additional macro definition as follows:

```
macro  ldv/3
    ldi    $3, $2
    ld     $3, $1
mend
```

Now we could write the last **ldv** instructions as a three operand one, for example to do the following:

```
ldv    r3,rex,r1
inc    r3
st     r1, r3
```

which would increase the value of the memory cell labelled by **rex** by 1. At this stage we might fancy turning this newly programmed pattern into a virtual instruction in its own right. All that it would take to achieve it is yet another macro definition:

```
macro incmem/1
    ldv    r3, $1,r1
    inc    r3
    st     r1, r3
mend
```

which would let us code

```
incmem rex
```

and avoid the tedium of writing 4 machine instructions in an entirely obvious manner. Isn't it wonderful?

As often happens in computing, the moment of triumph is spoiled by the devil-harbouring detail. The virtual instruction **incmem**, happily introduced by us via aggregating some simple instructions twice, has now lost the explicit indication of register use. By looking at it, we are unable, without reference to two definitions, to discern that **r1** and **r3** get corrupted by its execution. So for example

```
ldi r1, 15
do
    incmem rex
    dec r1
until eq
```

would not increment `rex` four times, but would instead result in nontermination. To remedy that, one could add a couple of register scopes:

```
macro incmem/1
    save r1
    save r3
        ldv r3, $1, r1
        inc      r3
        st       r1, r3
    restore
    restore
mend
```

Now the instruction `incmem` is completely safe, but we have introduced 4 extra instructions (i.e. the overhead of 100% on the original sequence) which in many cases would be unnecessary, e.g.:

```
incmem rex
incmem rex
```

would save and restore the registers twice, while this would at worst make sense once (and at best be completely unnecessary). Moreover, the attentive reader would already notice that other instructions in the macro would be redundant as well. Here is the expansion of the above:

```
# incmem rex
    save r1
    save r3
        ldi  r1,rex
        ld   r1,r3
        inc  r3
        st   r1,r3
    restore
    restore
# incmem rex
    save r1
    save r3
        ldi  r1,rex
        ld   r1,r3
        inc  r3
        st   r1,r3
    restore
    restore
```

All together 16 instructions! This simplifies to

```
# incmem rex
    save r1
    save r3
        ldi  r1,rex
        ld   r1,r3
        inc  r3
        st   r1,r3
# ...
    ldi  r1,rex
    ld   r1,r3
    inc  r3
    st   r1,r3
    restore
    restore
```

Observe that the `ldi` that executes as part of the `ldv` macro need not be present twice in the code either , since `r1` is not written into in the interim:

```

# incmem rex
    save r1
        save r3
            ldi r1,rex
            ld   r1,r3
            inc  r3
            st   r1,r3
# ...
    ld   r1,r3
    inc  r3
    st   r1,r3
    restore
    restore

```

The `st` followed by `ld` in the middle cancel each other (and the memory write is not needed, as that location is written into again at the end):

```

# incmem rex
    save r1
        save r3
            ldi r1,rex
            ld   r1,r3
            inc  r3
# ...
    inc  r3
    st   r1,r3
    restore
    restore

```

which leaves 5 instructions plus 4 save/restores, the latter of questionable utility. We have established that the cost of virtualisation was a factor of 3 in the worst case, or just under a factor of 2 in the best case (when the save/restore is indeed required).

This simple example is very instructive indeed. It shows the power of macro aggregation as a virtualisation mechanism, its flexibility and its potential cost. The removal of the redundant instructions is what a “proper” Level 4 compiler does, for example one compiling from the language C, which is one of the mainstream choices at Level 4. This is known as *peephole optimisation* owing to the fact that the removal is done based on looking at the code through a small window (a few machine instructions) and detecting actions that cancel each other, and which are therefore redundant. A macro processor of an assembler cannot do such a thing, it is too mechanical and low-level! All it does is dutifully substitute and associate.

This is a convincing enough demonstration why higher levels of abstractions, such as an L4 platform, are not realisable via aggregation. Even though at times it could be abstract *and flexible* enough, an assembler lacks intelligence to ensure efficiency when the translation mechanism that acts between platform levels needs to analyse the program *nonlocally*. To a certain extent this deficiency could be alleviated by more powerful macro techniques (and we will see a few below). However, any analyses involved in proper L4 programming languages, especially if not just effectiveness but high efficiency is required, are much more complex and elaborate than what the most sophisticated macro processor of an assembler can support.

Yet macro expansion, if used judiciously, is a powerful *low-level* virtualisation technique. We shall see how it is used to build Platform 3<sup>1/2</sup> on top of CdM-8 Platform 3 next.

# Down the Rabbit Hole

## 8.1 Platform 3

We are about to begin our descent to the ground level of computer organisation. We noted earlier, in section 2.1, that Platform 3½ is based on CdM-8 Platform 3, and that the virtualisation technique used for supporting the former is software aggregation of the latter, using macro facilities. As we are studying this virtualisation in the downward direction, the strategy is to establish the minimum instruction-set requirements of Platform 3 needed to support Platform 3½ and to show how the rest of the functions is covered by macro definitions. We shall start with ALU instructions.

### 8.1.1 ALU instructions

Looking at figure 5.1 on page 109, it is easy to see that the Platform 3½ instruction set has a certain degree of redundancy. Instructions can be re-introduced as macros involving other instructions. When this does not involve extra cost, i.e. when the total number of instructions does not increase, we can minimise the platform functionality without affecting either its effectiveness or its efficiency<sup>1</sup>. Such abstraction is known in literature as “zero cost abstraction” and represents a transformation of convenience, i.e. one that improves programmability without changing the core functionality.

The ALU instruction set includes the instructions `tst` and `clr`, which have often been used in programming seen so far as well as the programming exercises that accompany the course. The purpose of `tst` is to set the flags Z and N according to the data contained in the operand register. Since the flags are set by all ALU instructions anyway, we could use any instruction that does not change the operand in question for flag setting. For example, we could add 0 to the register, but it would not be without cost: we would need to use yet another register and ensure that it contains 0. Alternatively the `move` instruction with two identical operands that are the register in question would perform the required flag setting action exactly as required. Importantly the use of `move` instead of `tst` is zero-cost, since the instruction count (1) does not change. Accordingly, we introduce the following definition:

```
macro  tst/1
      move $1,$1
mend
```

and exclude the instruction `tst` from the Platform 3 instruction-set requirements.

On to the instruction `clr`, we observe that its effect is identical to subtracting a register from itself. This also has the right effect on the flags: C and Z up, and V and N down. Consequently we define:

```
macro  clr/1
      sub $1,$1
mend
```

and exclude `clr` as well. We have now 7 instructions left out of 9 presented in the table in figure 5.1, which still require Platform 3 support.

<sup>1</sup> Resources and abstractions do not live happily with each other: the engineer has to be constantly aware of the cost hidden under the shroud of abstraction. In this case the abstraction of machine instruction is not sufficiently uniform: depending on the underlying Platform 2 some instructions can take more memory/execution time than others. For now we will adhere to a simplifying assumption that the instruction count is a good indication of cost, but this will be revised in the next volume.

Let us now proceed to data movement operations shown in figure 5.3 on page 113. The instruction `shl` is defined as the equivalent of multiplication by two plus the value of flag C (0 or 1), with the flags set for overflow and carry in the standard way. This can be achieved via the `addc` instruction thus:

```
macro  shl/1
       addc $1,$1
mend
```

That eliminates `shl` and reduces the Platform 3 support requirement to 5 data movement instructions.

The logic instructions on figure 5.2 on page 110 are quite distinct. None of them can be replaced by an aggregate, which leaves us with 4 instructions to be supported.

In summary we have  $7+5+4=16$  ALU instruction for which we require support from Level 3. The rest can be aggregated from them using macros. The attentive reader will have noticed that the trick we employed to implement the Platform 3 $\frac{1}{2}$  instruction `shl` would also work with `shla` using a similar definition:

```
macro  shla/1
       add $1,$1
mend
```

which would enable us to reduce the load on Platform 3 even further to 15 basic instructions. Two considerations prevent us from following this route. Firstly, there is a certain granularity to instruction sets based on bit-strings. If the ALU instructions are to be a distinct group (and they should be for reasons that will be quite clear when we look at the Platform 2 implementation in terms of Platform 1), a certain number of bits is to be allocated to it in the opcode of the corresponding machine instructions. That number is the same for 16 and 15 instructions, i.e. 4 bits, so no saving would be made by removing one instruction, whereas our efforts to reduce the number from the present Platform 3 $\frac{1}{2}$  set of 20 down to 16 saves us a whole bit. For a platform whose machine word consists of only 8 bits such a saving is surprisingly very significant. We will discuss that in more details in the next section.

The second consideration is that our chosen cost metric is in fact not very accurate. There are two resources relevant to the concept of instruction efficiency: memory space and execution time. While the substitutions we have made do not require extra memory space (we replace ALU instructions by ALU instructions, and those are all the same size in CdM-8), the secondary aspect, the aspect of *time*, is more subtle. At this point we have no knowledge of CdM-8 Platform 2 and do not appreciate that two-register ALU instructions require two “ticks” of the platform clock<sup>2</sup>. By contrast single-register instructions require only one tick. It is for that secondary consideration CdM-8 implements one of the shifts in hardware; it cannot do both, since it would run out of bit-string to encode its instructions, but there is some gain in supporting at least one.

An alternative strategy for Platform 2 could be to support both left and right rotation (at present only `rol` is supported) in hardware, but the use of rotation is limited to some cryptographic algorithms which are either too complex for a beginner’s course or two demanding for this tiny platform.

The reason why we dwell so much on the pros and cons of various instruction-set decisions is that those are a good illustration of the principles of platform design. The reader will see that no decision is absolute, that any choice is essentially a tradeoff and as such is strongly dependent on the requirements and intensions of the designer.

### 8.1.2 Control

Control instructions of Platform 3 $\frac{1}{2}$  present a much greater challenge than ALU minimisation. There is a considerable variety of them, they are scoped and have parameters that represent events (such as conditions) rather than quantities, registers or addresses. Yet none of them feels primary to a discerning examiner. Indeed the action of a control instruction is fundamentally mixed: on the one hand they evaluate a condition and prepare a decision point. On the other, the decision is made always about one thing: whether or not to transfer control to a known point in the program. The difference between an `if` and a `while` is in that the former always passes the control “down”, i.e. to higher addresses in the program, whereas the latter has

---

<sup>2</sup> Any known Level 2 platform operates in discrete time steps, known as clock cycles or ticks. Your trusted desktop computer may proudly display a processor characteristic known as GigaHertz rate, e.g. 2.3GHz, which means that the Level 2 hardware receives 2.3 billion ticks every second

the ability to “close the loop”, in other words to return the control back, i.e. to a lower address in order to ensure repeated execution. Yet there is no *fundamental* difference between them: both are two-phase transactions, evaluation of a condition (which is external to the control instruction and which is done by any other instructions inside the scope) and the actual control transfer. Consequently, one would assume that a less abstract platform, such as Platform 3 would only need to support the latter, a conditional transfer of control in a simple, unstructured way, whereas the rest can be achieved by aggregation mechanisms.

That is, in fact, exactly how things stand with Platform 3<sup>1/2</sup>. The aggregation scheme is based on the Level 3 instruction family called *branches*. The simplest version of branch is the unconditional branch **br**

```
br addr
```

which transfers the control to *addr*. Put simply, the next instruction that will be executed after the instruction **br** is the one at address *addr*. What makes branches universally powerful as a control mechanism is their ability to sense a condition. The conditional Platform 3 branch is as follows:

```
bcc addr
```

where *cc* is one of the familiar abbreviations shown in fig 5.4 on page 117. What happens here is that the control is passed to *addr* if the condition is realised. For example,

```
    cmp r0,r1
    bgt    else
    move r0,r2
    br     done
else:  move r1,r2
done:  halt
```

This is a Platform 3 program fragment that copies to **r2** the lesser of the values contained in registers **r0** and **r1**. Indeed the **cmp** instruction sets the condition. If it is *gt*, i.e. register **r0** contains the greater number, the control is passed to the address **else**, where the **move** instruction copies **r1** to the result register **r2**. Otherwise (namely when the condition is not *gt*, hence **r0** is less than or equal to **r1**) it is register **r0** that gets copied to **r2**, and then the unconditional branch brings us to a **halt**. Clearly that is a Platform 3 implementation of the following Platform 3<sup>1/2</sup> code segment:

```
if
  cmp r0,r1
  is le
  move r0,r2
else
  move r1,r2
fi
halt
```

We learn a few lessons from this example. First of all, notice how much more elegant the upper platform’s code is. The programmer does not need to invent labels that merely designate “then” and “else”, but which cannot always be the same lest the program exhibit a name clash (i.e. different addresses marked with the same label). Also the code is much more readable as we can see where the alternative code sequences begin and end. Using branches could result in “spaghetti” code, i.e. code that is hard to trace and hard to understand the logic of.

Another lesson learned here is that the correspondence between the two platforms’ instructions can be established purely mechanically, without understanding either the logic or the intentions of the program. Indeed, we have three parts:

```
if
```

which does not seem to require any translation,

```
is le
```

which should be translated as

```
bgt else
```

We also require

```
else
```

which translates as

```
br      done  
else:
```

and

```
fi
```

to be translated as

```
done:
```

Of the four Platform 3<sup>1</sup>/<sub>2</sub> instructions, the instruction **is** presents the main difficulty in implementation via aggregation. It is due to the fact that the parameter, in this case **le**, is the conditional marker that must be notionally negated *before* making it part of a conditional branch. The parameter **le** must somehow be transformed into **gt** before being inserted as **\$1** into a branch. There exists however a simple solution to this problem. Instead of using textually unrelated alternatives **lt/ge**, **eq/ne**, **le/gt**, etc., let us define “regular” three-letter synonyms: **lt/nlt ge/nge**, **eq/neq**, etc. and then introduce macros for them:

```
macro bnlt/1  
    bge    $1  
mend  
macro bnge/1  
    blt    $1  
mend  
macro bneq/1  
    bne    $1  
mend
```

and so on. Assuming similar definitions have been introduced for all 16 conditions in the aforementioned figure 5.4, we can easily define the four macros required to code the conditional construct:

```
macro if/0  
mend  
  
macro is/1  
    bn$1    else  
mend  
  
macro else/0  
    br      done  
else:  
mend  
  
macro fi/0  
done:  
mend
```

Let us submit our findings to the macroassembler **cocas** to see what happens. It has the option **-lx** which causes it to show all its macro expansions:

```
$ cocas -lx testm1  
CdM-8 Assembler v2.1 <<<testm1.asm>>> 19/08/2015 00:05:50
```

```

1 macro if/0
2 mend
3
4 macro is/1
5           bn$1      else
6 mend
7
8 macro else/0
9       br      done
10 else:
11 mend
12
13 macro fi/0
14 done:
15 mend
16
17 macro bnle/1
18       bgt      $1
19 mend
20       rsect    prog
21       if
22 # >>>>
23 # <<<<<
00: 71           cmp r0,r1
25       is le
26 # >>>>
27       bnle    else
28 # >>>>
01: ec 06       bgt    else
30 # <<<<<
31 # <<<<<
03: 02           move r0,r2
33       else
34 # >>>>
04: ee 07       br      done
36 else:
37 # <<<<<
06: 06           move r1,r2
39       fi
40 # >>>>
41 done:
42 # <<<<<
07: d4           halt
44       end

```

In this mode, `cocas` comments the beginning and end of an expansion by right and left chevrons. Although it requires some effort to see it, one can satisfy oneself that exactly the Level 3 instructions that we had intended were produced by macro expansion. All lines that contain Level 3 instructions (as opposed to pseudo-instructions or macros) are highlighted in red; also they have memory content shown on the left. Compare them with the code segment on page 176.

This seems to completely define how Platform 3<sup>1/2</sup> is realised via aggregation from Level 3 but in fact the technique is somewhat more involved. To start with, the above macro expansion contains two labels, which, if we used another `if` in the same program would lead to a name clash. The use of labels is unavoidable since the macro has no way of knowing where the consequent and alternative begin and end. This kind of name clash is avoidable using the technique that we consider next.

### 8.1.3 Nonce and unique

We need the ability to generate unique names to use inside a macro to avoid a name clash. Here is how it can be done in a macro that copies a NULL-terminated string from one place to another using exclusively Level 3 instructions (the operands are pointed to by registers **r0** and **r1**):

```
$ cocas -lx strcpym.asm  
CdM-8 Assembler v2.1 <<<strcpym.asm>>> 19/08/2015 00:35:07
```

```
1  macro strcpym/0  
2      push r2  
3      push r0  
4      push r1  
5  loop':      ld r0,r2  
6      inc r0  
7      st r1,r2  
8      inc r1  
9      move r2,r2  
10     bne loop'  
11     pop r1  
12     pop r0  
13     pop r2  
14 mend  
15  
16     asect 0  
00: d0 10    ldi      r0,0x10  
02: d1 20    ldi      r1,0x20  
03:          strcpym  
04:          # >>>>  
04: c2        push r2  
05: c0        push r0  
06: c1        push r1  
07: b2        loop1:   ld r0,r2  
08: 8c        inc r0  
09: a6        st r1,r2  
0a: 8d        inc r1  
0b: 0a        move r2,r2  
0c: e1 07    bne loop1  
0e: c5        pop r1  
0f: c4        pop r0  
10: c6        pop r2  
11:          # <<<<  
11: d1 30    ldi      r1,0x30  
12:          strcpym  
13:          # >>>>  
13: c2        push r2  
14: c0        push r0  
15: c1        push r1  
16: b2        loop2:   ld r0,r2  
17: 8c        inc r0  
18: a6        st r1,r2  
19: 8d        inc r1  
1a: 0a        move r2,r2  
1b: e1 16    bne loop2  
1d: c5        pop r1  
1e: c4        pop r0  
1f: c6        pop r2  
1g:          # <<<<  
19:          end
```

Notice the *apostrophe* placed after the label `loop` in the macro definition on line 5, and the use of `loop'` for referring to this point in `bne`. Inside a *single* macro the apostrophe acts like a fixed decimal number, say 15. One can think of the label `loop'` as being `loop15` or `loop7` for this matter, it makes no difference. Two things are important: the label is syntactically correct (letters followed by digits) and it is the same label everywhere inside one macro. So if the apostrophe represents 15, then the label on line 5 is `loop15` and the label on line 10 is also `loop15`. If the apostrophe represents 7 then both occurrences stand for `loop 7`. However, if a macro expands again, *cocas guarantees* that the value of the apostrophe *will be different!*

The idea of the apostrophe is that of freshness. The value is specific to each occasion (macro expansion). A word coined for one occasion is called a ‘nonce’ in English, and so we will call the apostrophe in a macro expansion the nonce. Nonces occur elsewhere in computer science, for example in cryptography, in exactly the same meaning. See how the nonce works in lines 24/29 where it took the value 1 and in lines 40/45 where its value turned out to be 2. Using the nonce we can perfectly safely and effectively use any number of labels in all kinds of macros.

Before we return to the issue of control macros, let us first observe that if we were to write a string copy macro, we would wish it to have registers as parameters. That way we could write instructions such as `strcpy r1,r2`. Surely we could modify the example above to make this happen? Say, like this:

```
macro strcpy/2
    push r2
    push $1
    push $2
loop':      ld $1,r2
            inc $1
            st $2,r2
            inc $2
            move r2,r2
            bne loop'
            pop $1
            pop $2
            pop r2
mend
```

Surprisingly it would not work! The reason is that the above macro silently assumes that none of the arguments is in fact `r2`. Worse still, there is no way that the macro *user* may be made aware that `r2` is special, if the macro is part of the platform rather than the program. If the user wrote

```
strcpy r2,r3
```

it is easy to see that this would overwrite the output address with the input data and the result would be incorrect (or even nonterminating). Consequently, it is not safe to use specific registers inside a macro if the parameters are also expected to be supplied in registers. We need a facility that assigns a *free* register, i.e. a register not used by the parameters, to some kind of variable that we may then use as a work register designation. The macro-assembler provides such a facility in the form of `unique` macroinstruction. Here is how it is used:

```
$ cocas -lx strcpyr.asm
CdM-8 Assembler v2.1 <<<strcpyr.asm>>> 19/08/2015 01:24:48
```

```

1  macro strcpys/2
2          unique $1,$2,temp
3          push ?temp
4          push $1
5          push $2
6  loop':      ld $1,?temp
7          inc $1
8          st $2,?temp
9          inc $2
10         move ?temp,?temp
11         bne loop'
12         pop $1
13         pop $2
14         pop ?temp
15 mend
16
17         asect 0
18         strcpy r1,r2
19 # >>>>
00: c0          push r0
01: c1          push r1
02: c2          push r2
03: b4  loop1:   ld r1,r0
04: 8d          inc r1
05: a8          st r2,r0
06: 8e          inc r2
07: 00          move r0,r0
08: e1 03        bne loop1
0a: c5          pop r1
0b: c6          pop r2
0c: c4          pop r0
0d: # <<<<<
0e: c0          push r0
0f: c3          push r3
10: b1  loop2:   ld r0,r1
11: 8c          inc r0
12: ad          st r3,r1
13: 8f          inc r3
14: 05          move r1,r1
15: e1 10        bne loop2
17: c4          pop r0
18: c7          pop r3
19: c5          pop r1
47: # <<<<<
48          end

```

The instruction on line 2 tells `cocas` that its operands *must be unique registers*. Macro expansion first substitutes specific registers for `$1` and `$2`. The `unique` instruction will then scan its operands and encounter a *macro variable* `temp`. It knows that `temp` is a variable because it is not a register: the item does not start with the letter `r` nor is it followed by a ternary digit. The instruction `unique` then acts very much like the nonce: it takes any register not mentioned in the list, and it does not matter which, and associates it with the variable `temp`. It is easy to see that at most four items may be supplied to a `unique`. From zero to

four variables may occur on the list. When zero variables occur, the `unique` just checks that the registers are pairwise distinct, and reports an assembler error otherwise. When all items are variables, the `unique` simply associates them with arbitrary, but distinct registers.

The object associated with a macro variable can be explicitly recalled by placing the question mark before it, as is the case on lines 3,6,8,10 and 14. Also note that macro variables are *not* labels, and hence do not name-clash with those. It is possible to name a macro variable the same as an opcode, label, etc. (but, naturally, not a register name for obvious reasons)

Looking at line 18 we see that the macro `strcpy` is given `r1,r2` as parameters. Line 20 tells us that the `unique` chose `r0` as a free register not clashing with the parameters; that register was then used on lines 23,25,27 and 31 in places where the macro variable is recalled. On line 33 `strcpy` is used with parameters `r0,r3`. Now the `unique` chooses `r1` as a fresh register, since `r0` is used by the first parameter. Accordingly lines 35, 38, 40, 42 and 46 show `r1` where the macro variable is recalled.

Here is the same piece without macro expansion:

```
$ cocas -l strcpyr.asm
```

```
CdM-8 Assembler v2.1 <<<strcpyr.asm>>> 19/08/2015 01:46:19
```

```

1  macro strcpy/2
2          unique $1,$2,temp
3          push ?temp
4          push $1
5          push $2
6  loop':      ld $1,?temp
7          inc $1
8          st $2,?temp
9          inc $2
10         move ?temp,?temp
11         bne loop'
12         pop $1
13         pop $2
14         pop ?temp
15 mend
16
17         asect 0
00: c0 c1 c2 b4 18     strcpy r1,r2
04: 8d a8 8e 00
08: e1 03 c5 c6
0c: c4
0d: c1 c0 c3 b1 19     strcpy r0,r3
11: 8c ad 8f 05
15: e1 10 c4 c7
19: c5
20             end

```

Looking at lines 18 and 19 one can easily believe one has written a program for a “real” Level 3 machine that has string processing register instructions, whereas by sliding one’s glance up to lines 1-15 one is brought back to reality. The reader will agree that, if it were not for resource considerations, aggregation as a virtualisation technique would work perfectly with assemblers!

#### 8.1.4 Control revisited

Equipped with the nonce we are about to revisit the conditional construct to see if we can now safely code it as a macro using exclusively Level 3 instructions (branches). The code below has been our best effort so far:

```
macro if/0
mend

macro is/1
    bn$1      else
mend

macro else/0
    br        done
el:
mend

macro fi/0
done:
mend
```

Clearly we could use the nonce with the two labels that we require to avoid repeated use of the same label in different contexts. The difficulty we would have in doing so is that the nonce is stable within a single macro, but as soon as a new expansion starts, its value changes. So if we merely added the apostrophe in this manner:

```
macro if/0
mend

macro is/1
    bn$1      else'
mend

macro else/0
    br        done'
el':
mend

macro fi/0
done':
mend
```

the result simply would not work. The nonce in the macro **is** would have a different value compared to the nonce in the macro **else**. To make matters worse, consider the situation where two conditional constructs are used one inside the other:

```
if
<...>
is ...
<...>
if
<...>
is ...
<...>
else
<...>
    fi
else
<...>
fi
```

Since the macro `else` expands into a branch instruction whose target is inside a `fi` instruction, which `fi` instruction should it be for the `else` marked in red? Clearly the answer is the first one, not the second one. The last `else` in the nest of `ifs` should couple with the first `fi`. This is yet another instance of the LIFO discipline, which we have seen so many times already. It calls for two things: macro variables for connecting different macros together and a macro stack to preserve the LIFO scoping discipline.

Here is how it is achieved in the macro language of `cocas` (we present the source first to avoid an unwieldy listing) :

```

1 macro if/0
2         mpush '_'
3 mend
4
5 macro is/1
6         mpop    id
7         mpush   alt?id
8         bn$1   alt?id
9 mend
10
11 macro else/0
12         mpop    where
13         mpush   new?where
14         br     new?where
15 ?where:
16 mend
17
18 macro fi/0
19         mpop    term
20 ?term:
21 mend

```

Let us go through the macro definitions one by one and illustrate what happens there with an example.

**the macro if** defined on lines 1–3 is now nonempty. It engages the nonce and pushes `_'` on top of the *macro stack*. The macro stack is the same kind of LIFO stack that we have seen before, but it is used by the macro processor of the assembler during the compilation, not the platform during the program execution. The pseudo-instruction `mpush` (not to be confused with the instruction `push`, which is a Level 3 machine instruction) places its operand (after macro expansion) on top of the macro stack. For illustration purposes, let us assume that the nonce had the value 7.

**the macro is** defined on lines 5–9 has three lines of code. First (line 6) it pops the stack into the macro variable `id`. In our illustration it will be associated with `_7`. Then (line 7) it will push onto the stack `alt?id`, i.e. `alt_7`, then it will produce the branch with the target `alt_7`. The net effect of it is that the `is` instruction will expand into a branch that triggers by the opposite condition and which bypasses the consequent to a label `alt_7`. The label itself will be placed by the `else` or `fi` instruction, which we will look at next.

**the macro else** defined on lines 11–16 pops off the stack what the instruction `is` left there. The stack in our illustration has `alt_7` at the top, which will be associated with the variable `where` (line 12). The top of the stack will have `newalt_7` after line 13 and the macro will expand into an unconditional branch with the target `newalt_7`.

**the macro fi** defined on lines 18–21 finally pops the value (in our illustration the label `newalt_7`) that was the target of the unconditional branch and places it as a label.

Here is the summary of how this will be expanded according to the above analysis:

Source	Expansion
<pre> if   &lt;...&gt; is ne   &lt;...&gt; else   &lt;...&gt; fi </pre>	<pre> # nothing &lt;...&gt; beq alt_7   &lt;...&gt; alt_7:   br newalt_7   &lt;...&gt; newalt_7: </pre>

In the case when `else` is not used the expansion will proceed as follows:

Source	Expansion
<pre> if   &lt;...&gt; is ne   &lt;...&gt; fi </pre>	<pre> # nothing &lt;...&gt; beq alt_7   &lt;...&gt; alt_7: </pre>

Expanding the macros in this by hand is left as an exercise to the reader. Notice how the `else` instruction replaces the label `alt...` by `newalt...` at the top the stack to make sure that the negated branch of the `is` instruction passes the control to the alternative that follows below and not the `fi` instruction, which will use the label `newalt...` for exiting the `if`.

One can conclude at this point that the use of the macro stack, nonce and the variables allows the programmer to construct a correctly working conditional macros. The question that still remains is whether or not this will work when conditionals are nested, i.e. occur inside the consequent and/or the alternative of another.

Notice that the macro stack is left exactly in the same state as it was before `if`. The macros collectively popped as as many items off the stack as they pushed on it. This means that if the consequent and/or the alternative used further conditionals in them, the above illustration would still be strictly valid: the content of the top of the stack would still be as we indicated. Moreover, there would be no name clash with any labels generated by the conditionals inside the consequent/alternative, since all the labels we have produced in our illustration contain the nonce, and since any other conditional construct would use a different nonce value in the `if` macro.

Let us see how the following be expanded by the macroassembler:

```

macro bnle/1
    bgt    $1
mend
    rsect   prog
    if
        cmp r0,r1
    is le
        move r0,r2
    else
        move r1,r2
        if
            tst r1
        is le
            neg r1
        fi
    fi
    halt
end

```

Here is the relevant part of the listing:

```

26          rsect    prog
27          if
28 # >>>>
29 # <<<<<
00: 71          cmp r0,r1
30          is le
31          # >>>>
32          bnle    alt_1
33          # >>>>
01: ec 06          bgt     alt_1
34          # <<<<<
35          # <<<<<
03: 02          move r0,r2
36          else
37          # >>>>
04: ee 0b          br      newalt_1
38          alt_1:
39          # <<<<<
06: 06          move r1,r2
40          if
41          # >>>>
42          # <<<<<
43          # <<<<<
44          move r1,r2
45          if
46          # >>>>
47          # <<<<<
48          tst r1
49          # >>>>
07: 05          move r1,r1
50          # <<<<<
51          is le
52          # >>>>
53          bnle    alt_5
54          # >>>>
08: ec 0b          bgt     alt_5
55          # <<<<<
56          # <<<<<
57          # <<<<<
58          # <<<<<
0a: 85          neg r1
59          fi
60          # >>>>
61          alt_5:
62          # <<<<<
63          fi
64          # >>>>
65          newalt_1:
66          # <<<<<
67          # <<<<<
0b: d4          halt
68          end
69

```

The listing dispels any doubt one might still harbour about whether or not the macro stack works as it should. Last observation: we used prefixes `alt` and `newalt` to name the labels, and the nonce made sure the full labels were all pairwise distinct; that does not eliminate *all* possible name clashes. It is conceivable that the programmer may want to name `alt_3` a piece of data in a `dc`, which may then clash with a macro if the nonce value should happen to be 3. In order to guarantee that such things never happen, macro packages tend to use obscure prefixes such as multiple consonants, dollars (when the dollar is considered a letter in the syntax of the assembly language, which is not the case here), or underscores. We would be much safer to use two underscores as the prefix of all generated labels and to inform the programmer that she should never name her labels that way<sup>3</sup>.

---

<sup>3</sup> which is exactly how the CdM-8 standard macro library, which the reader has used so far without knowing it, deals with its generated symbols

### 8.1.5 Loops

The implementation of Platform 3½ loop instructions in Level 3 using macro facilities is generally very similar to the conditional. Let us take a look at the `while` construct:

```
while
    <...>
stays CC
    <...>
break
    <...>
continue
    <...>
wend
```

Our first observation here is that we cannot use the same macro stack for loops as we did for conditionals. If we are to keep information about labels at the top of the stack, any intervening `if` would shield it from a termination instruction (i.e. `break/continue`). As a result a termination instruction would not be able to interact with the enclosing loop to see where to pass the control. The solution is to use a separate macro stack for loops, and in fact `cocas` provides a small array of stacks for use in macro definitions. The default stack is stack 0, which is the one we used to define the macros for conditionals. Now we are going to use stack 1 to define all the above loop instructions:

```
1 macro while/0
2     1mpush __b',__e'
3 __b':
4 mend
5
6 macro stays/1
7     1mpop end,beg
8     1mpush ?beg,?end
9     bn$1 ?end
10 mend
11
12 macro break/0
13     1mpop end,beg
14     1mpush ?beg,?end
15     br ?end
16 mend
17 macro continue/0
18     1mpop end,beg
19     1mpush ?beg,?end
20     br ?beg
21 mend
22
23 macro wend/0
24     1mpop end,beg
25     br ?beg
26 ?end:
27 mend
```

To implement the loop we maintain two labels at the top of the macro stack 1: the label that marks the beginning of the loop and the label that marks the end. On line 2, the `while` macro creates two nonce-differentiated labels `__b` and `__e` for this purpose, pushes them onto the macro stack 1, and marks the beginning of the loop with the former (line 3). The decision point, the macro `stays`, reads these labels into the macro variables without affecting the stack (it pops them off and pushes them back on). Then it expands to a branch to the end on the negated condition. The `break/continue` instructions do exactly the same thing with the stack and expand into unconditional branches to the end and beginning points of the loop, respectively (lines 15/20). Finally the instruction `wend` pops both labels off the stack, thus leaving it

balanced, expands to an unconditional branch to the beginning of the loop and places the end-label (lines 25–26). We shall compile the following piece with the above macros:

```
rsect test
while
    dec r0
stays gt
    add r1,r0
    if
        dec r1
        is eq
        continue
    fi

    inc r2
    if
        add r2,r2
    is vs
        break
    fi
wend
end
```

Here is the listing. We only show the relevant lines, highlighting macro-expanded lines in red. It is left as an exercise to the reader to check that the logic of the above piece is preserved, and that all branches are correct and are matched with labels placed appropriately:

```
66  rsect test
67  while
69  __b1:
00: 88      71  dec r0
              72  stays gt
01: ed 0f      76  ble __e1
03: 14      79  add r1,r0
              80  if
04: 89      83  dec r1
              84  is eq
05: e1 09      88  bne     alt_4
              91  continue
07: ee 00      93  br __b1
              95  fi
              97  alt_4:
09: 8e      100  inc r2
              101  if
0a: 1a      104  add r2,r2
              105  is vs
0b: e7 0f      109  bvc     alt_9
              112  break
0d: ee 0f      114  br __e1
              116  fi
              118  alt_9:
120  wend
0f: ee 00      122  br __b1
              123  __e1:
              125  end
```

Definition of the until loop macros presents little challenge after the while loop has been fully understood; it is left as an exercise to the reader. More obscure macros: **save/restore** and the case of **break/continue** terminating multiple loops/iterations, as well as the issues of intelligent error reporting, all of which implemented in the Platform 3½ standard macro library, go beyond the scope of this introductory book. The

interested reader is referred to the CdM-8 Ecosystem Manual available online. The rest of the Platform 3<sup>1/2</sup> instructions in fact belong to Level 3 and are supported by `cocas` directly, without recourse to the standard macro library. It is they that must be considered collectively as CdM-8 Platform 3.

### 8.1.6 Less common control structures

For the sake of completeness we should note that all control instructions seen so far have a labelled target: all branches pass the control to the address provided by the immediate operand and the `jsr` likewise calls the subroutine whose address is contained in the immediate operand. There are situations in which the programmer may wish to pass the control to an address computed in the process of program execution rather than one provided by the compiler. CdM-8 platforms no simple solution to this when the branch is conditional; indeed conditional branches are there to support scoping control structures only<sup>4</sup>. However, an *unconditional* branch to a computed address can be achieved indirectly thus:

```
push r1  
rts
```

Here no subroutine is being called, and one may wonder how an `rts` can be meaningful without a `jsr`. However, the effect of the instruction `rts` is that the top of the stack is popped off and put it in the PC. If the address of a machine instruction is pushed onto the stack just before an `rts` the effect of this will be exactly the same as that of an unconditional branch `br` to the instruction. In the above segment the machine instruction that will be executed after the `rts` is the one pointed to by `r1`. What would this be useful for?

High-level programming languages often have the construct `case` whereby a variable is compared to a set of values, and if it holds one of those, the corresponding piece of code is executed. In a way this is similar to an `if` with more than one alternative. A Level 2 implementation of such a construct could well be code that evaluates the condition and fetches the address of the next instruction from an array. Using the above pattern, this is easily achievable. The cost of not having a dedicated machine instruction for it is small: the two instructions together are the same size as a conventional branch, but the execution requires two stack operations, which takes significant time due to memory access. Yet the fact it is possible to achieve completeness without this type of branch being a dedicated machine instruction at Level 2 helps to tighten up the instruction set. Fortunately, useful examples of *conditional* branches to a computed target are too exotic to worry about how they may be supported efficiently. Naturally, a conditional branch can be followed by the above `push/rts` pattern thus delivering an effective solution.

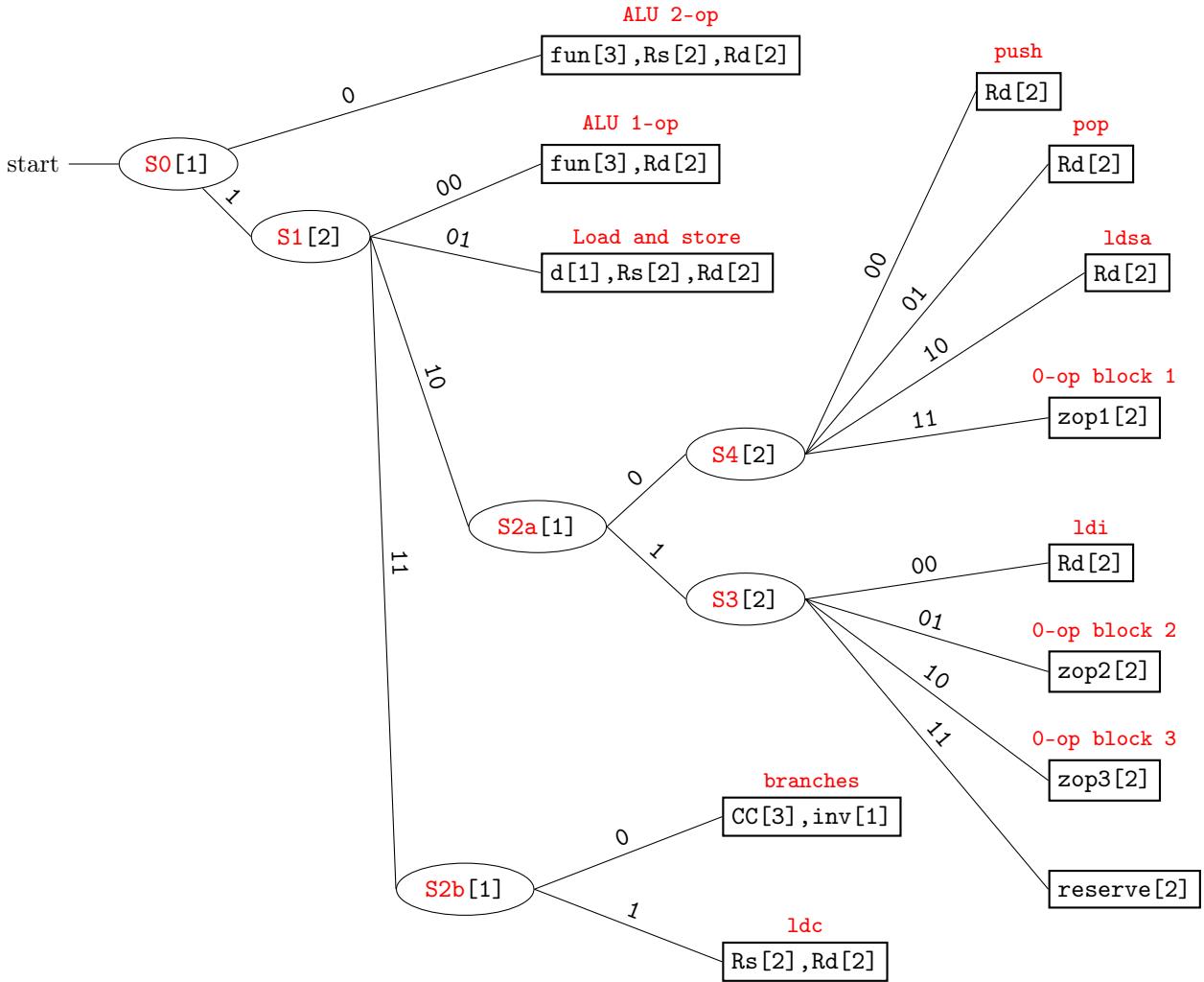
Another case of a computed target is the already familiar Level 3<sup>1/2</sup> instruction `jsrr` (see section 6.4.3). It is implemented as the following macro:

```
macro jsrr/1  
push $1  
crc  
mend
```

It is easy to see that the coroutine call will pop the address off the stack (pushed on it by the previous instruction using the macro operand) and that it will push the return address on the stack, which is indistinguishable from a `jsr` to the address given by the macro operand `$1`. Again the overhead here is purely in terms of execution time, since stack operations are introduced without need, but the economy of a dedicated Level 2 instruction is quite useful given the deficit of space in an 8-bit machine word.

## 8.2 Platform 2

We have now successfully dealt with all of the Platform 3<sup>1/2</sup> aggregation facilities that make it half-way between Levels 3 and 4. We can now focus on the next step down, from the pure Level 3 CdM-8 assembler (no macro facilities of any kind) to the world of bit-strings in which we need to arrive to be able to proceed to the hardware levels that lie beneath. In this effort we will be assisted by our earlier studies of data



bit-string (dec equiv)	fun (ALU 2-op)	fun (ALU 1-op)	d	zop1	zop2	zop3	CC(inv=0) branches	CC(inv=1) branches
0	move	not	st	addsp	halt	ioi	beq	bne
1	add	neg	ld	setsps	wait	rti	bhs/bcs	blo/bcc
2	addc	dec		pushall	jsr	crc	bmi	bpl
3	sub	inc		popall	rts	osix	bvs	bvc
4	and	shr					bhi	bls
5	or	shla					bge	blt
6	xor	shra					bgt	ble
7	cmp	rol					br	nop

Immediate operands for: **ldi**, **ldsa**, **jsr**, **branches**, **addsp**, **setsps**

Figure 8.1: Structure of the CdM-8 instruction set (field bit sizes are shown in brackets).

structures, see section 3.14; the reader is advised to revise that section before continuing with the current one.

The Level 2 platform introduced here is the CdM-8, which is a machine that has an 8-bit instruction set. This means that all Level 3 assembler instructions (which excludes any Level 3 pseudo-instructions, since they do not belong to Level 2) must be packed into exactly 8-bits each, although some may require an extra machine word (i.e. another 8 bits) for the *immediate operand*, such as the `ldi` data byte or the `jsr` address. The extra word in all cases follows the instruction's byte without a gap. Needless to say that none of the following facilities: separate compilation, linking, relocation and object libraries belong in Platform 2. That platform deals with a single bit-string that contains compiled code and data for the whole application.

It is rather surprising that the large variety of machine instructions that we have encountered, including any choice of register operands, can be represented in only 8 bits (in fact even with some information capacity to spare), yet it is not only possible, but can be done efficiently. The key idea here is to use variant records, which we have seen before. A variant record has a selector field at the beginning, which determines the interpretation of the rest of the bit-string. Each value of the selector corresponds to an individual “variant”, i.e. bit string format. Another contributing idea is that of nesting, where a variant record contains in its individual variants further variant records. The nesting is hierarchical, which means that there is a *selector sequence* that leads to a specific bunch of fields corresponding to a specific machine instruction.

We now turn to the graphical representation of the instruction set shown in figure 8.1. The nesting used in the instruction set is quite special. While in a general record any field can be defined as a variant record, in instruction sets it is typically the first one. As a result, the various selectors form a contiguous prefix to data fields. In order to get to the latter (which in the case of the instruction set consists of opcodes, register numbers, function/condition codes, etc.) the Level 2 executing agent, the processor, has to recognise the combination of selectors at the beginning of the machine instruction. The number of selectors is not fixed, since values of earlier selectors determine the format of the rest of the word, including the size and meaning of any subsequent selectors. However, when all selectors have been located and identified, the rest of the instruction contains only data of the aforementioned sorts in certain fixed positions.

The diagram has some important properties:

1. read left to right, all paths along the lines go through selectors (shown inside ovals) and arrive at exactly one plain record (shown in a rectangular box), which is the terminal point of the path.
2. the total number of bits shown on the lines and declared in the box is always 8.
3. the total prefix written on the lines that lead to the box is unique and does not occur even as an initial part of the prefix on another path.

Because the prefix is unique, it is possible to uniquely determine to which Level 3 group of instructions (shown as a heading in the table under the diagram) any 8-bit string belongs. Since the record in the box consists of register numbers and values, whose symbolic representation is shown in the table, we can uniquely identify the Level 3 instruction.

Conversely, since any possible Level 3 instruction has its opcode and other values in the table, we can, given an instruction, fill in the corresponding record box. When we prefix its content with the bits shown on the path leading to it, we get exactly the 8 bits of the corresponding Level 2 machine instruction.

Notice that there are unused instructions in the instruction set, such as the one corresponding to the value `0b11` in the group `zop3`. Also some of the opcodes in the table, e.g. all of the `zop3` members except `crc`, have not been discussed yet. Most of them are required to support the interaction of the Platform 2 with the environment and/or used to support an operating system. We included them in the table for the sake of completeness, and will come back to them later.

Let us first try to translate individual instructions by hand. We start with a simple ALU instruction

```
sub r1,r2
```

---

<sup>4</sup>there is no conditional subroutine or coroutine call either, so the design is consistent in that sense. However the reader should not draw far-reaching conclusions from our simple examples. There are sufficiently common and very practical platforms in which *every* instruction is conditional, but the set of engineering tradeoffs they address is by no means basic.

The Level 3 op-code here is `sub`. We locate it in the table under the column `fun` for ALU 2-op. Looking at the diagram we see that ALU 2-op is a box that can be reached from the starting point (far left) via one selector `S0`. The record has a 3-bit opcode and two 2-bit register numbers. Reading the selector value shown on the line, we get the prefix 0. From the table we get the value of `fun` for `sub` as 3, or 011 in 3-bit binary. The registers `Rs` and `Rd` (the first and second register operands, respectively, or, as engineers like to call them, the source and destination registers) are, in 2-bit binary, 01 and 10, respectively. Gathering all bits together, we get 0-011-01-10, or 0x36.

Another example:

```
jsr 0x17
```

We find `jsr` under the zero-operand group 2 (`zop2`) in the table. That group is reachable from the starting point of the diagram via a path labelled as 1-10-1-01. The translation of `jsr` from the table is 2, or in 2-bit binary 10. The result is 1-10-1-01-10, which is 0xd6. Remembering the immediate operand, we get the desired translation: `d6 17` in hex.

Notice that there is space for another zero operand group, which would be called `zop4` and have the prefix 1-10-1-11, if it existed, but the CdM-8 processor simply has no need for 4 more instructions.

Let us have a go at the branches: translate `blt 0x20`. The opcode is found in the table under `inv=1`, bit-string value 5 (3-bit binary 101). The box content is consequently 1011, the prefix is 1-11-0-, so the instruction in hex (remembering the immediate operand) is `eb 20`.

Let us now try it in the opposite direction. Suppose by peeping in the platform memory we have discovered that a cell contains the content `0xc7` and we wish to know what instruction it is. First convert to binary: 1100 0111. Starting from the left we observe that `S0` requires 1 bit, and we split it off: 1-1000111. Since `S0=1`, we take the lower route on the diagram and come to `S1`. Here we must split off 2 bits: 1-10-00111, following the third route from the top to `S2a` which needs one bit: 1-10-0-0111. Accordingly we follow the upper line to `S4`, splitting two further bits off: 1-10-0-01-11. The selector value `S4=01` indicates the `pop` instruction and the remaining two bits lead us to `pop r3`.

Another example: `0x35`. Binary conversion: 00110101. Upper route from `S0`. Structure: 0-011-01-01. Translation: `sub r1,r1`.

Finally, translate: `0xe6`. Binary conversion: 11100111, structure: 1-11-0-011-0, opcode: `bvs`. Need the immediate operand. Suppose the next byte in memory is `0x03`. Then the full instruction is `bvs 0x03`.

It should be noted that the instructions `ldi` and `ldsa` are rather special as they are fully determined by their path prefixes 1-10-1-00 and 1-10-0-10, respectively, and have no distinct opcodes. All they have in their boxes is a single register number. Those two instructions do not occur in the table in any form. One must bear in mind that despite the lack of opcode, they have immediate operands and occupy each two consecutive bytes!

### 8.2.1 How does the assembler work?

We have seen how individual instructions are assembled and disassembled. This covers the format conversion between Levels 3 and 2, but there is one important aspect that we have swept under the carpet: assembling the program from instructions. In fact this topic harks back to the earlier section on linking, since the principles of assembling programs from instructions are not dissimilar from those of building applications from sections. In both cases two steps, or as they say *passes* are made over the collection of things being brought together: the *allocation pass* and the *resolution pass*. Compiler designers usually refer to them as pass 1 and 2, but we will keep the full name to remind ourselves what is involved in each.

**Allocation pass.** The assembler reads the source program line-by-line, from beginning to end. It creates and maintains the *allocation pointer* (which is the equivalent of the PC in the platform): it initialises it with the lowest available address (this can be established by a pseudo-instruction) and increments it every time a Level 3 instruction (or a `dc/ds`) has been allocated. For each Level 3 machine instruction the assembler only tries to determine two things:

1. what label, if any, the instruction is labelled with;

2. the size in bytes of the bit-string that corresponds to the instruction.

Item 1 above is used to take note of the pair (label, current allocation pointer). Item 2 is used for advancing the allocation pointer to make it point to the address from which the next machine instruction will be accommodated.

**Resolution pass.** At the end of the allocation pass all labels that are used in the program are collected into a table in which they are listed against the value of the allocation pointer at their position in the program, i.e. the allocation address of the instruction that they label. Now the assembler reads the source from the beginning one more time. The difference between the first and second reading is that whereas during the allocation pass the assembler did not know what addresses were behind their symbolic names (labels), now that all labels have been collected and translated into addresses any instruction can be compiled into a bit-string. The assembler analyses the structure of each Level 3 instruction, determines all the fields of the corresponding box in figure 8.1 and generates the bit-string. In doing so it creates and maintains the allocation pointer in exactly the same way as it did then.

It should be noted that some of the labels will be exts and some ents (see section 7.1), while the rest will be labelling instructions and data in the source file. Also there will be not one but several allocation pointers, one of which used with the asects, and the rest with the r-sects. These details may also vary from assembler to assembler, but the two-pass method of compilation described above is absolutely fundamental.

The cornerstone assumption laid in the foundations of the 2-pass assembly process is that **it is possible to determine the size of the bit-string corresponding to a machine instruction or constant-allocation pseudo-instruction without knowing allocation addresses for any labels.** This only works if the format of such instructions is sufficiently stable, independent of the physical address. One example of how this might *not* be the case comes from the observation that branches perform a dual role. On the one hand there are “near” branches, those that transfer the control around a small section of the program (an *if* or a small loop); on the other hand they could be used for long jumps to different sections of separately compiled parts, such as subroutines, etc. One can imagine a flexible format of a branch whereby the *difference* between the current and destination address is stored in the machine instruction. Moreover, if that difference is small, the branch follows format 1, and allocates, say, 4 bits for it with the whole instruction fitting a single byte. If the difference is large, then the instruction indicates that it uses format 2, under which an immediate operand (an extra byte) is allocated. This seems a clever way of saving memory (and speeding things up, too), but it breaks the fundamental assembly principle: in order to compile such a branch, the assembler must know whether the address is far or near, but in order to determine that, it needs to complete the allocation pass first to get the actual addresses. The latter is not possible, since some of the instruction have an unclear size.

When the fundamental principle does not hold, assembling may still be possible by using iterative approximation, i.e. making assumptions and corrections until the translation becomes consistent. This is a long and protracted process which platform designers usually seek to avoid. It is therefore the case in all modern platforms, that address-specific versions of instructions, if available, are selected by the programmer (or a Level 4 compiler on her behalf), and the assembler merely checks that her assumptions (or Level 4 allocation decisions) are borne out by the allocation pass and reports errors, but it does not attempt to change the instructions.

We conclude this section with an example of assembling a simple asect by hand:

```

1      asect    0
2      ldi      r0,x
3      ld       r0,r0
4  loop:   dec     r0
5          bne  loop
6          halt
7  x:      dc      17
8      end

```

Pretending to be an assembler, we do the first pass:

source	AP	size	labels	comments
asect 0	0			set the AP
ldi r0,x	0	2		ldi has an immediate op
ld r0,r0	2	1		
loop: dec r0	3	1	loop=3	
bne loop	4	2		a branch has an immediate op
halt	6	1		
x:dc 17	7	1	x=7	one byte specified in dc
end				

Here AP is the allocation pointer and we assume hex everywhere just as `cocas` does in its listing. Notice how in every row of the table the AP is increased by the value in the size column to obtain the next AP value.

On to the second pass:

source	AP	size	code	comments
asect 0	0			set the AP
ldi r0,x	0	2	d0 07	x=07 from pass 1 table
ld r0,r0	2	1	b0	
loop: dec r0	3	1	88	
bne loop	4	2	e1 03	loop=03 from pass 1 table
halt	6	1	d4	
x:dc 17	7	1	11	
end				

The translation is done using figure 8.1 as before. Now let us see if `cocas` agrees:

CdM-8 Assembler v2.1 <<<tst.asm>>> 22/08/2015 14:56:06

```

      1      asect    0
00: d0 07      2      ldi      r0,x
02: b0          3      ld       r0,r0
03: 88          4  loop:   dec     r0
04: e1 03        5      bne     loop
06: d4          6      halt
07: 11          7  x:      dc      17
                  8      end

```

Clearly it does!

# On the ground climbing up: Platform 0

In section 3.1 we introduced the idea of discontinuity on which the computing world is based. Digital data arise either as a result of dealing with a discontinuous phenomenon, or when it is possible to identify with sufficient certainty discrete values for a continuous real-world phenomenon. For example, the presence or absence of a material object, the truth or falsehood of a statement, and the count of items present, are all discontinuous phenomena. We may also use discrete values to represent continuous phenomena, such as length, or light frequency (colour), or sound pressure level when the underlying technology introduces *forbidden intervals* in a continuous quantity's range of variation. In such a case, even though the quantity remains continuous and can potentially take any values within its range, the *probability* of its finding itself within a forbidden interval is so low as it needs to be to ensure reliable discretisation. For example, if we wish to use light intensity as a discrete quantity (think of light telegraph between ships at sea), we must use the light source that provides high light intensity when on and no light at all when off. Any ambient light makes the reading at the receiving end continuous, but if the probability of high-intensity ambient light is low (e.g. sun reflection off a low-flying drone) then our discrete abstraction on/off is adequate.

Descending to the bottom of the Tanenbaum classification (Level 0), we see a *physical* implementation of digital data in the form of electrical *signals*. Electrical signals are, in fact, *continuous* phenomena of the kind mentioned above, but we devise circuits that divide these signals into two distinct classes (those representing zero and those representing one) with a forbidden interval between them that corresponds to the signal intensity in between. These two classes form the basis of data representation at Level 0. Even though there is some variety of platforms at Level 0, in this book we focus on the silicon technology known as CMOS (Complementary Metal-Oxide-Semiconductor), which we will refer to as Platform 0 with a capital “P”.<sup>1</sup>

---

<sup>1</sup>Notice that despite the style of the name, this level has nothing to do with CdM-8. It is completely general-purpose and is a basis of the majority of modern computing machines except perhaps some very special-purpose ones.

## 9.1 Representing ones and zeros in Platform 0

An electrical signal is an observable characteristic of a conductive *wire*. By applying instrumentation to the wire we can tell what electrical signal is present in the wire at a given time. A wire cannot sustain a signal, all it can do is carry it from one end to another. This happens so fast that for our introductory analysis of signals we can assume that the whole wire, both ends and any intermediate point, exhibits exactly the same observable signal at any time. In other words, signals do *not* move along wires in an observable way; the whole wire behaves electrically the same at every point along its length.<sup>2</sup>

Physically, signals in platform design tend to be voltages. A voltage is an electrical characteristic which indicates the potential of two electric terminals for creating an electric current between them. The *resistance* of the conductor between the two terminals tells us how much current will flow from one end to the other for a given voltage (Ohm's law): the higher the voltage between two terminals and the lower the resistance of the conductor the more current will flow.

Voltages are normally measured in Volts, abbreviated to V, which are standard (SI) units that one can see everywhere in electrical and electronic equipment. In the UK the mains voltage is 240V, but in the USA the mains are at 110V; a lithium battery in a mobile phone produces a voltage of 4.3V between its terminals, a USB port in a computer (or a wall-wart power supply unit) supplies 5V to the USB plug. The elementary physics of electricity is covered in the secondary school curriculum, but for the sake of completeness we include some preliminaries below.

In systems design, voltages are observed with respect to a common reference point of the whole system, called the *ground*. When we say that a wire is at +5V, we mean that the voltage (also known as *the potential difference*) between the wire and the ground is 5 Volts. Because computing platforms are based on discontinuous discrete systems, the *exact* voltage levels we use are not very important. What *is* important is that there is a readily detectable difference between voltages used to represent 0 and those used to represent 1, i.e. that there is a reliable forbidden interval enforced by all components of the system.

### 9.1.1 Some definitions

**Definition 14 : Voltage Levels** Platform 0 signals are voltages that are compared with a certain threshold voltage that depends on silicon technology. Voltages that are *above* the threshold are said to be at a *high* level, or high, and those that are *below* the threshold are said to be at a *low* level, or low.

**Definition 15 : Binary signals** Platform 0 is neutral to the binary interpretation of high/low: we may use low signals to represent 0 and high signals to represent 1, or low to represent 1 and high to represent 0. For either choice of interpretation circuits are available to implement the same digital functionality as the other. We call the interpretation of binary signals low as 0 and high as 1 the *natural interpretation* and will stick to it when considering the operation of Platform 0 and Platform 1

The Platform 0 machine supplies a wire that is held at 1 by the platform's power source or the ground. Also supplied is a wire likewise held at 0. Both wires may be extended by the designer to any point in the design and are thus universally available. Other common names for the two wires are  $V_{DD}$  (voltage above the transistor drain) and  $V_{SS}$  (voltage below the transistor source), or PWR and GND (PoWeR and GrouND).

Having established the signal levels we are ready to discuss the properties of Platform 0 devices.

---

<sup>2</sup> In reality there is a tiny delay due to signal propagation along a wire; silicon engineers are aware of it and there is a standard technique for neutralising its negative consequences. That is why we can ignore it with impunity.

**Definition 16 : terminal** Platform 0 devices are silicon structures equipped with input terminals, output terminals, or input/output terminals. A terminal is literally an end-point, a point where silicon meets with wire. Wires can only be connected to silicon structures at their terminals.

An **input terminal** can have a digital signal (0 or 1) applied to it (we say *asserted* on it). An input terminal which is not asserted is said to be *floating*.

An **output terminal** can apply (we say it *asserts*) a signal level (0 or 1) on its connecting wire. Some output terminals must apply a signal level at all times, others have the ability to *disconnect*, i.e. behave as if the terminal is not connected to the wire to which it is attached.

In some cases a single terminal can behave as an input terminal at one point in time and as an output terminal at another. Such a terminal is said to be an **input/output terminal**.

**Definition 17 : wire** A wire is a piece of conductive material that can connect any number of electrical terminals belonging to Platform 0 devices. Only one device connected to any wire can assert on the wire at any given time, otherwise the wire becomes *conflicted*<sup>a</sup>. Any number of devices may receive inputs from the wire at any given time<sup>b</sup>

All terminals connected to a wire are kept at the same voltage (and consequently same signal level) by it, which is the voltage asserted by the device currently providing output to the wire. The function of the wire is to assert that signal level on every input to every device it connects. If a wire has no device asserting on it, or if it is connected only to terminals currently acting as inputs, the wire is said to be *floating*.

Two connected wires are indistinguishable from a single wire connecting all the devices attached to either wire.

---

<sup>a</sup>strictly speaking, the wire may not be conflicted if two devices assert the same signal at the same time, but it is safer to work on the assumption that no wire may be asserted by more than one device at any time

<sup>b</sup>in practice the number of inputs is limited by the ability of the devices to produce sufficiently strong currents, but we will not worry about this here.

**Definition 18 : switch** A switch is a device that behaves as a wire connecting two terminals at some moments in time whereas at other times it renders them disconnected. The choice is determined by an external agent operating the switch. Thus, switches may be used to make the output of one device asserted on the input of other devices at specific times.

When a switch between two devices is open the devices are disconnected. When the switch is closed they are connected to one another. This is the terminology used in electrical circuits that employ *mechanical* switches. A mechanical is operated directly by a human being using moving objects such as buttons, plungers, etc.

Platform 0 employs *electronic* switches in the form of a silicon transistor. In reality they act more like valves in plumbing - either allowing current through or blocking it off. The effect is the same, so the term *switch* is often used to avoid requiring people to think about circuits in two different ways. However, when we look *inside* a transistor we view it as if it were a valve, so a transistor that allows current through is said to be *open*, whilst a transistor that blocks the flow of current is said to be *closed*.

The purpose of platforms at Level 2 and above is to run programs. The purpose of Platform 0 is to implement *designs*. Each design is an aggregate, a compound device of the same sort as the basic Platform 0 devices but with a more complex functionality.

**Definition 19 : design** A Platform 0 design is a combination of wires, stand-alone terminals (terminals that are not connected to any silicon directly) and Platform 0 devices such that:

1. some wires are labelled for input, some for output and some for input/output, the third category being optional;
2. each labeled wire is connected to a distinct stand-alone terminal, which becomes an output, an input or an input/output terminal of the design according to the label of the wire.
3. input wires have no output terminals connected to them in the design;
4. the removal of any of the devices changes the relationship between the signals on input and output terminals.

A representation of a Platform 0<sup>a</sup> design in diagrammatic form showing how terminals are connected with wires is called a *circuit diagram*.

---

<sup>a</sup>Platform 1 uses exactly the same representation, except its circuit elements are aggregates of indivisible Platform 0 devices, rather than being indivisible devices in their own right.

In plain language, a Platform 0 design is a collection of Platform 0 devices connected to one another by wires attached to their terminals. Some — or all — of these terminals become the terminals of the design, turning it into a new (Level 1) device. Universal Platform 1 devices are usually called gates and buffers, while Platform 0 devices are normally referred to as transistors (or transistor pairs). Consequently a Level 0 design is a transistor aggregate representing a gate or a buffer. Platform 0 has a small number of other elements as well, but none as important as the transistor, and we will discuss them when the need arises.

## 9.2 Transistor

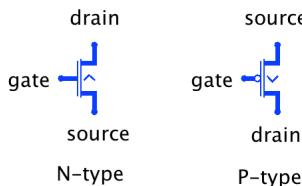
From the inception of conventional computer architecture that dates back to the first general-purpose ENIAC computer in the 1940s a *controlled switch* has firmly held its place as the basic building block of all computer hardware.

A simple example of switch functionality is the push-button used to control a door-bell. This has two terminals and is operated by hand. When it is pressed down it connects the two terminals together and completes the circuit. When the button is released, a spring pushes the terminals apart and the circuit is broken.

An electronic switch is different from a mechanical switch in one important aspect: it is controlled by a *signal*, rather than by a human hand. An electronic switch has a third, control, terminal, which, depending on the binary value asserted on it, connects or disconnects the other two. The control terminal must not be allowed to float, since under such conditions, the "door bell" will ring sporadically, and at unpredictable times. The value 1 could be used to close the switch and 0 to open it, or the other way around: both kinds of switch are used in the technology, and besides they are used in pairs in the same design.

We have just described in simple terms the working of a modern MOSFET transistor, the basic building block of the silicon chips found in current computers, mobile phones, washing machines, spacecraft — anywhere that digital data is to be manipulated.

In our circuit diagrams we will represent transistors using the following symbols<sup>3</sup>:



There are two types of transistor: the N-type and the P-type. Each type of transistor has three terminals known as the *source*, the *gate* and the *drain*. The source and the drain are the input and output terminals for the switch respectively. The gate is a terminal used for the control signal. The two types of transistor are distinguished by the way they respond to control signals. The N-type transistor is *opened* by asserting 1 on the gate: this connects the source to the drain. When the signal on the gate is 0, the route from source to drain is blocked. We say that the N-type transistor is open when the gate is up (i.e. equals 1) and closed when the gate is down (i.e. equals 0). A P-type transistor acts in a similar fashion, except that it is open when the gate is down and closed when the gate is up. The situation is completely symmetrical.

In circuit diagrams the names of the terminals are not normally written down. The symbol leaves no doubt as to which of the three terminals of a transistor is connected to the gate, and the little arrow between the two other terminals always points from source to drain. To see which type the transistor is, we observe that the P-type has a little circle on its gate terminal.

The transistor acts as a pure function of its inputs: the signal at the drain depends on the signals on the gate and source at the same moment in time<sup>4</sup>. To better understand how the output from a transistor depends upon the signals asserted on its two inputs we may use truth tables familiar from elementary logic, listing the output of a device against all possible combinations of the input values. Here is one for the N-type transistor:

gate	source	drain
0	0	Z
0	1	Z
0	Z	Z
1	0	0
1	1	1
1	Z	Z

<sup>3</sup>these symbols are not the most common ones, but we will use the notation employed by the Logisim logic simulation package, because all of our examples will be run on Logisim. If you are curious about the international standard symbols for devices you may find them in public domain reference materials.

<sup>4</sup>ignoring for now any delay in opening or closing the transistor, and in propagating a signal from source to drain

Our “truth table” is a little special as it includes the extra value  $Z$  which represents *floating*. The transistor’s gate cannot be allowed to float under any circumstances, because the output would then be unpredictable. However, the other input — the source — can. The principle remains the same: if the gate is open, the source is connected to the drain with an effective wire, and so the output simply follows the input, whatever value.

Here is the truth table for the P-type transistor:

gate	source	drain
0	0	0
0	1	1
0	$Z$	$Z$
1	0	$Z$
1	1	$Z$
1	$Z$	$Z$

As one can see, the effect of the control signal on the gate of a P-type transistor is the opposite to its effect on the gate of an N-type transistor.

### \*9.3 Transistor imperfections

The above “truth tables” for transistors are in fact even less conventional. Not only are there three possible results (0,1, and  $Z$ ) the discretisation that the transistor depends on, i.e. the existence of the forbidden interval between 0 and 1, has a *quality* to it. Indeed, to be treated abstractly as a discrete device the transistor should enforce the *same* forbidden interval on its output, i.e. drain, as it receives on its control input, i.e. its source. If that is the case, we are able to connect transistors any way we choose, without worrying that the signal propagating across the circuit might cease to be identifiable with sufficient certainty. Unfortunately such a guarantee is not available in the CMOS technology. In fact, the quality of different lines of the truth table varies. Here are less idealised tables:

gate	source	drain	gate	source	drain
0	0	$Z$	<b>0</b>	<b>0</b>	<b><math>0^*</math></b>
0	1	$Z$	0	1	1
0	$Z$	$Z$	0	$Z$	$Z$
1	0	0	1	0	$Z$
<b>1</b>	<b>1</b>	<b><math>1^*</math></b>	1	1	$Z$
1	$Z$	$Z$	1	$Z$	$Z$

$\leftarrow$ -N-type      P-type- $\rightarrow$

Both transistors are perfect enough when they are closed. The N-type transistor is perfect when it is open and pulling *down*, i.e the source is connected to the ground. The transistor acts like a wire from the drain to the ground, and the latter can sink any necessary current to bring the drain down towards zero safely outside the forbidden interval. A reliable position for an N-type transistor is to have a path through the circuit to the ground. The source can float, in which case the drain will also float in all circumstances, or be asserted with 0, in which case an open N-type transistor is a good approximation to a wire. The P-type transistor has exactly the same peculiarity: it is good at pulling up towards the power supply and its reliable position is when the source has a path to it through the circuit. What we say below about the N-type transistor can be re-stated for the P-type one in an obvious way, so we will only do it once.

Problems start when an N-type transistor is used in a circuit where its source has a path to the power supply. When the transistor is open and if the source is high, it will attempt to connect it to the drain, and an N-type transistor cannot quite do that. What happens is the drain will be pulled up but not as far out of the forbidden interval as in the case of pulling down towards the ground. The signal will be somewhat *degraded*. For example if the forbidden interval is between 1.5 and 3.5 Volts under a 5V power supply, an N-type transistor would be able to raise the drain up to say, 4V, whereas pulling down would achieve less than 0.1V. Electronic circuits are susceptible to electromagnetic interference, which causes small signals to appear on wires spontaneously over time. A 1.4V gap between the lower end of the forbidden interval and the transistor output is quite sufficient to make the probability of misidentification infinitesimal. On the other hand, a mere 0.5V at the top is less so.

Worse still, signal degradation of this sort can accumulate. If a second N-type transistor in a similar position (high source) is asserted with a degraded signal, when that transistor opens the drain will exhibit a combined effect from both transistors, bringing the “high” level even closer to the higher end of the forbidden interval. Consequently, the regimes indicated in red in the above tables tend to be avoided. There may be advantages in connecting transistors in this precarious way as well, and we will mention some later, but then signal degradation must be prevented before it has a chance to build up and cause digital errors. How this may be done will become clearer in the next chapter.

# Universal Platform 1

Having discussed the ground-level technology, if only in a cursory manner, we are prepared to make the first step up on our return journey to CdM-8 Platform 2, which we left at the end of Chapter 8. As we explained in Chapter 2, the transition from Platform 0 to the Universal Platform 1 is by means of *aggregation*: bringing together the building blocks of Platform 0 to create compound Level 1 devices that can be combined (by further aggregation) to eventually build a functional implementation of Platform 2.

We will show and briefly discuss an implementation for each of these Level 1 devices in our chosen Platform 0. Sometimes it will be the only possible implementation, but most of the time it will not; however, a detailed analysis of the pros and cons of different implementations is well beyond the scope of this book.

## 10.0.1 Logic gates

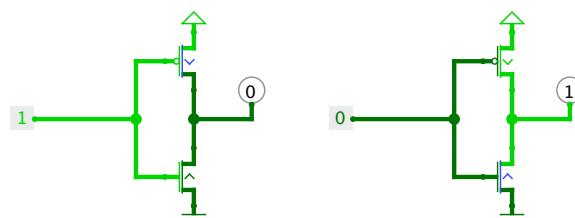
The simplest Level 1 blocks are commonly referred to as “logic gates”, or just “gates”. Each such block creates an output signal by opening and closing its component transistors in a manner that serves to assert either the power or the ground of Platform 0 on its output, serving as a gate for the signal to pass through. The term “logic gate” derives from the fact that these blocks may be thought of as implementing logic functions if we view one signal level as representing *true* and the other as representing *false*. It is routinely assumed that the signal representing 1 also represents *true*, and the signal representing 0 also represents *false*, so designers stick to 1 and 0, rather than referring to true and false.

There is an unfortunate opportunity for confusion between the term “gate” used to refer to a Level 1 unit from which Level 2 designs may be constructed, and the gate terminal of a transistor, which is called a gate for a good (and similar) reason. We will do our best to avoid such ambiguity by making sure that we qualify the term *gate*: we shall talk of a *transistor gate* when we need to explain why a transistor is open or closed, and specify which named logic gate we are referring to when talking about Level 1 units of design.

Now it is time to demonstrate how something useful may be made out of transistors.

## 10.1 NOT-gate

As few as two transistors can be aggregated to form a useful design known as the “NOT-gate”, also commonly referred to as *inverter*



The NOT-gate is formed by connecting two transistors as shown in the diagrams above. The left and right drawings are of the same circuit, except that on the left the input presented to the NOT-gate is 1 and on the right it is 0. The straight lines between transistors represent wires, and the round dots at the intersection of wires means that they are connected (otherwise they merely intersect in the geometric sense, being otherwise

two separate wires). In the diagrams above the pale-colour wires are those carrying the signal 1 and the dark ones are those carrying the signal 0, but this is not a convention; we will ignore the colour of wires in subsequent circuit diagrams as it is generally not informative.

Notice that

- The bottom transistor is an N-type device
- The top transistor is a P-type device
- Both transistors' gate terminals are connected to the NOT-gate's input signal
- The two transistors' drain terminals are connected together to form the NOT-gate's output
- The bottom transistor has its source connected to the Platform 0 ground (0), which is depicted on circuit diagrams as a triangle filled with horizontal strokes and pointing down
- The top transistor has its source held at 1 by the Platform 0 power supply (depicted by a hollow triangle pointing up)

In the **left-hand circuit**

- The input signal is held at 1
- The bottom transistor is open (because it is an N-type device), so its drain is asserting the signal 0 (from its source) on the output terminal of the NOT-gate
- The top transistor is closed, so its drain is disconnected from its source

We conclude that under the input signal 1, the NOT-gate's output is pulled down by the Platform 0 ground.

In the **right-hand circuit**

- The input signal is held at 0
- The bottom transistor is closed, so its drain is disconnected from its source
- The top transistor is open, so its drain is asserting the signal 1 (from its source) on the output terminal of the NOT-gate

So when the same circuit is given an input of 0 the situation changes symmetrically, and the NOT-gate's output is pulled up by the Platform 0 power supply.

Consequently, our two-transistor aggregate has the ability to invert the input signal: the output is 1 when the input is 0 and vice versa. In logic under the standard interpretation (1: true, 0: false) this behaviour is associated with the NOT operator, hence the name of the circuit, the NOT-gate.

### 10.1.1 A brief note about timing considerations

The behaviour of the NOT-gate on Level 0 presented here is somewhat idealised. In reality, when the signal applied to a transistor gate changes (from 0 to 1 or vice versa), an electric current flows into or out of the gate area of the transistor to charge or discharge it to its new level. The gate has some capacitance, meaning that it requires a small amount of electric charge to flow in order for it to reach its new voltage.

The power supply can provide a limited current into a transistor, so it takes some time for the charge to flow, just as it takes time to charge up a mobile phone battery, in fact the greater the capacitance and the higher the voltage the longer it takes. That is why modern computers prefer low-voltage processors and why smaller transistors are faster than larger ones.

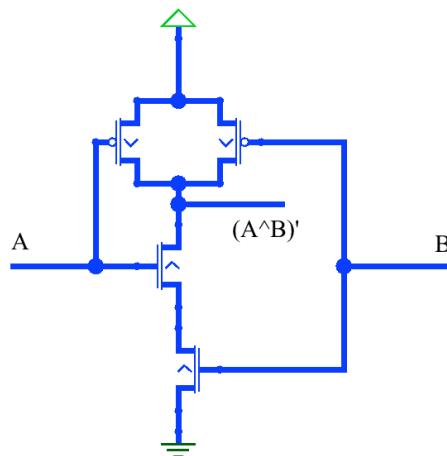
So the output of a NOT-gate is not *immediately* equal to the logical negation of the input. There is always a small delay in propagating input changes to the output. In modern silicon technology the delay is in the region of 5 picoseconds ( $5 \times 10^{-12}$  sec, or five millionths of a microsecond). Such delays are characteristic of all Level 1 logic gates and have to be taken into account in Level 2 architecture; we will come back to it later. Suffice it to say that delays of this kind limit the speed at which a processor may execute an instruction, and a great deal of effort is put into minimising them.

## 10.2 NAND-gate

The NAND-gate has two inputs, which we shall call  $A$  and  $B$ , and one output. The output asserts the value  $(A \wedge B)'$ , where  $\wedge$  is the logical connective “and” (also known as conjunction) and the apostrophe stands for logical negation. The truth table of NAND is as follows:

A	B	$(A \wedge B)'$
0	0	1
0	1	1
1	0	1
1	1	0

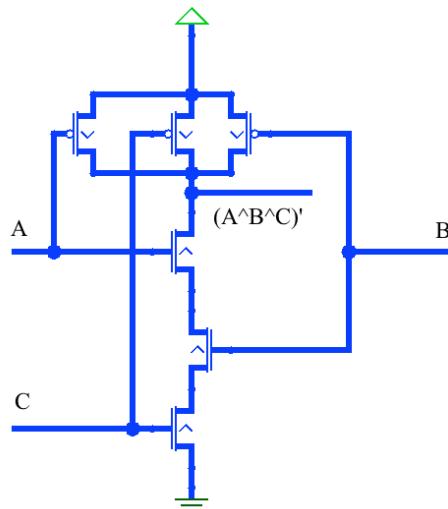
Here is a Platform 0 implementation of the NAND-gate:



When  $A$  and  $B$  are both at 1, the two P-type transistors at the top are closed and the two N-type transistors below them are both open, connecting the output to the ground. The result is 0. Any other levels at the input would result in at least one of the top two transistors being open and at least one of the bottom two being closed, so the output would be connected to the power supply and disconnected from the ground.

We conclude that the output conforms to the truth table shown above. Notice that the above solution shares a distinctive characteristic with that for the NOT-gate: transistors are used in pairs, one N-type and one P-type. This will be the case in *all* subsequent designs and is an important feature of the silicon technology called CMOS, which stands for *Complementary* MOS transistors.

Level 1 gates can have any number of inputs. For example a 3-input NAND-gate looks as follows<sup>1</sup>:




---

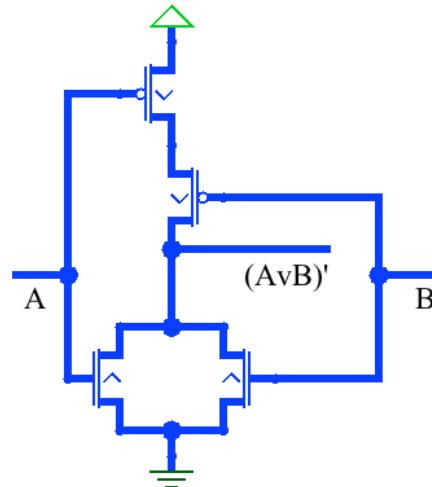
<sup>1</sup>As we mentioned earlier, when two line intersect, the two wires they represent connect if and only if the intersection is marked with a dot

### 10.3 NOR-gate

The NOR-gate is a circuit with two inputs  $A$  and  $B$ , and one output. The output terminal asserts the value  $(A \vee B)'$ , where  $\vee$  is the logical connective “or” (also known as disjunction) under the standard interpretation of signals. The truth table of NOR is as follows:

A	B	$(A \vee B)'$
0	0	1
0	1	0
1	0	0
1	1	0

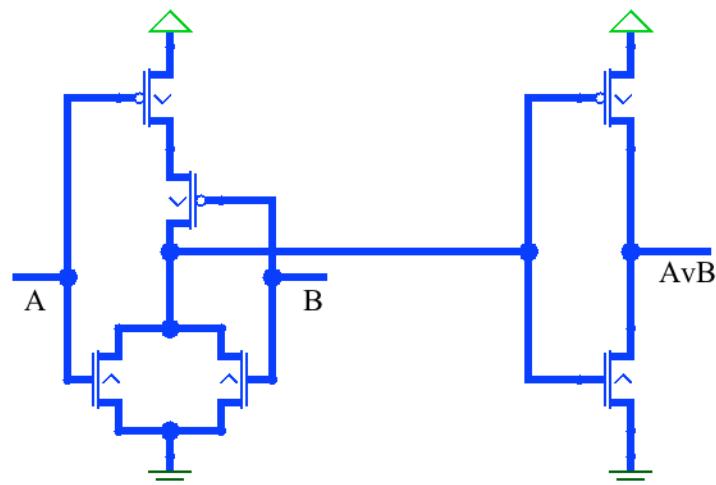
The Platform 0 implementation of the NOR-gate is a transposition of the NAND design:



If both inputs  $A$  and  $B$  are at 0, the bottom transistors are closed and the top ones are open, asserting 1 (power supply) on the output. Any other combination of inputs leaves at least one of the top two transistors closed and at least one of the bottom two open, asserting 0 (ground) on the output.

A NOR-gate with more than two inputs can be constructed in a manner similar to the NAND-gate.

NAND and NOR can be made into AND and OR by inverting (i.e. negating) their output. All this takes is another CMOS pair aggregated into a NOT-gate and fed with the output of a NAND or a NOR. Here is an OR-gate:

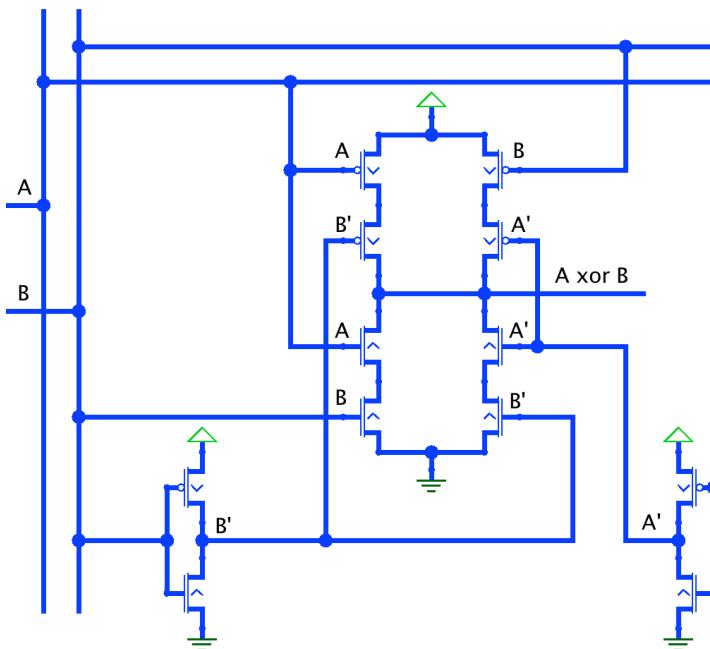


## 10.4 XOR-gate

The XOR-gate has two inputs,  $A$  and  $B$ , and one output. The output terminal asserts the value  $A \oplus B$ , where  $\oplus$  is the logical connective “exclusive or” (also known as addition modulo 2) under the standard interpretation of signals. The truth table of an XOR-gate is as follows:

A	B	$A \oplus B$
0	0	0
0	1	1
1	0	1
1	1	0

There are several implementations of the XOR-gate in Platform 0. Let us start with the most straightforward one.

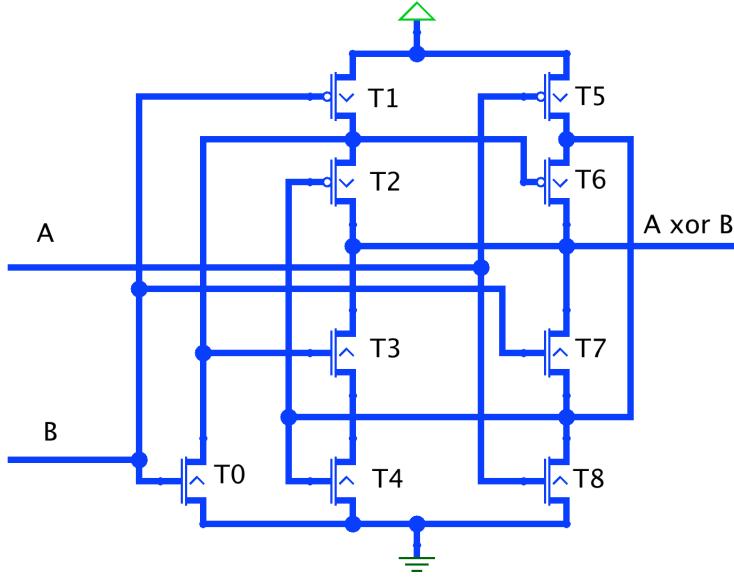


Surprisingly this gate takes three times the number of transistors we needed for NAND/NOR. The circuit is well structured and easy to understand. The two pairs of transistors at the bottom are connected to form by now familiar NOT-gates, the intention being to produce inverse signals  $A'$  and  $B'$  so that we may use both the original  $A$  and  $B$  and their inverses. The top four transistors are P-type, pulling up to the power supply, i.e. 1. The output wire ( $A \oplus B$ ) is at 1 when either the left pair or the right pair are open. The former happens when  $A = B' = 0$ , and the latter when  $B = A' = 0$ , which give us the two middle rows of the truth table.

The other four transistors are N-type. They pull down to the ground, i.e. 0. The output wire ( $A \oplus B$ ) is at 0 when either the left pair or the right pair are open. The former happens when  $A = B = 1$ , and the latter when  $A' = B' = 1$  which give us the last and the first rows of the truth table, respectively.

Notice that all four combinations of inputs are mutually exclusive, so if a pair of P-type transistors pull up, no pair of N-type transistors is open to pull down and vice versa; there is no conflict on the output line.

This straightforward implementation is neither unique nor the most economical one, even among well-behaved circuits that do not cause any signal degradation (mentioned in section \*9.3). The following design fully implements the XOR-gate in only 9 transistors:



Here surprisingly the number of N-type transistors is greater by one than that of P-type transistors. One way of reading this circuit and understanding how it works is to set  $B = 0$  and simplify it for this case. Transistor T0 becomes a gap, T1 a short-circuit (source to drain), T6 a gap (due to T1) and T3 a short-circuit (also due to T1) and finally T7 a gap. The remaining four transistors form two inverters connected in series, which deliver  $A$  to the output, thus taking care of the first and third rows of the truth table (where  $B = 0$ ).

Now set  $B = 1$ . T0 becomes a short-circuit, T1 a gap, T6 a short-circuit (due to T0), T3 a gap(also due to T0) and T7 a short-circuit. What remains is a single inverter for  $A$  made up by T5 and T8, whose output is shunted by T2 with a grounded source. We might think for a moment that it is problematic, since T2 is P-type and prefers a high source, but notice that the transistor gate of T2 is asserted by the output of the T5/T8 inverter, which produces  $A'$ . So if  $A' = 1$  T2 is closed and therefore perfect, and when  $A' = 0$ ,  $A'$  is already grounded by the T5/T8 inverter and therefore T2 sees both its source and drain at 0. Strictly speaking, this represents a weak conflict on the output wire: T2 is pulling its drain up slightly above the ground due to its inability to assert a clean 0, while the N-type T8 unleashes the full power of the ground on the same wire, just as any N-type transistor would. As a result, T8 wins and the output signal is a clean 0. We conclude that T2 plays no role when  $B = 1$  and the output is simply  $A'$ , which takes care of the remaining two lines of the truth table (where  $B = 1$ )<sup>2</sup>.

The design is as clever as it is instructive. It shows how hard it is to predict the minimum number of components needed for a Level 1 design even when the circuit is as simple as an implementation of exclusive-or. In any practical circuit-building, the designer usually defines *a* solution, while its transformation into a more optimal one is handled by hardware optimisation tools based on complex combinatorial algorithms — any manual optimisation is limited to tiny circuits only.

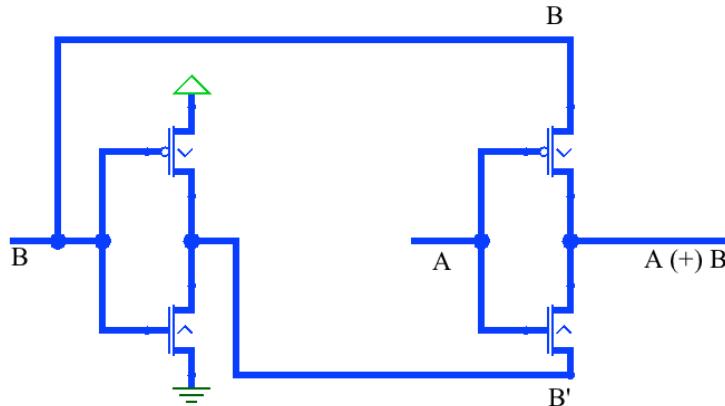
As a final remark, the XOR-gate has its dual, the XNOR-gate, which works as an XOR-gate followed by an inverter. XOR is often used to determine whether two signals are different. Naturally an XNOR-gate is there to determine if two signals are the same.

## \*10.5 Pass Transistor Logic (PTL)

And yet we need a minimum of 9 transistors to implement a simple logical connective! Fortunately there are other ways of building circuits which tend to be more economical. If we are prepared to tolerate some signal degradation a solution with as few as four transistors is readily available:

---

<sup>2</sup> The best way to make sure this circuit is well understood is to actually draw it in a circuit simulator and delete and replace components by wires as suggested by the last two paragraphs



Here is how it works. The left-hand pair of transistors is wired up as a NOT-gate. The input to this part of the circuit is  $B$ , so the wire that runs across the bottom of the circuit carries the signal  $B'$  to the right-hand pair. The wire across the top carries the signal  $B$ . If we now look at the right-hand pair of transistors we see that the source of the top one is asserted with  $B$ , and the source of the bottom one is asserted with  $B'$ .

The input signal  $A$  is fed to the gates of the two right-hand transistors. When  $A$  is 0, the bottom-right (N-type) transistor is closed and the top-right (P-type) one is open, so  $B$  is asserted on the output of the XOR-gate. When  $A$  is at 1, the bottom-right transistor is open and the top-right one is closed, so  $B'$  is asserted on the output, in full agreement with the XOR truth table.

What makes this so economical is that we have relaxed the requirement that all P-transistors have to have a path to the power supply from their sources, and that all N-transistors have a similar path to the ground. The right-hand pair have neither. Such a design method, i.e. gating signals between input and output directly rather than switching the power supply/ground to the output wires, is known as *Pass Transistor Logic* (PTL) and is used to reduce the number of CMOS pairs in the design. PTL circuits are much simpler and often faster (due to signals having to traverse fewer transistors), but the signal degradation mentioned in section \*9.3 makes them less reliable if measures are not taken to counter it. In our case the remedy could be a simple pair of conventional NOT-gates connected in series and so inverting the output twice. Since each NOT-gate produces its output by gating the power/ground, an imperfect signal will become perfect by traversing the first NOT-gate, while the second one restores the original logical value. Unfortunately, this also increases our transistor count to 8, which is just one fewer transistor than the best conventional design we have shown. Of course if we are to build an XNOR gate which requires an inverted after an XOR circuit, then the above PTL-style XOR implementation would be perfect, since it gives us a low transistor count while any signal degradation will be recovered from by the following conventional inverter which is needed anyway.

## 10.6 Platform 0 revisited: Pull resistors

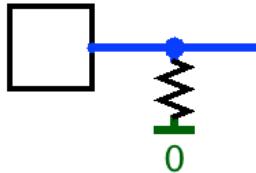
Transistor gates do not work if their inputs float. However, it is convenient sometimes to allow devices to disconnect their outputs under control, so that a default value is used in the absence of an assertive signal. This does not seem possible at first glance, since we have no means to distinguish between an actively asserted value (be it 0 or 1) and a value established at random in a floating wire.

What we are missing so far in the picture is the idea of *power* alongside voltage. We already discovered in the previous section that the idealisation of a transistor may not work if it is put to a severe enough test. This happens because a signal passing from the source of a transistor to its drain loses voltage which may lead to identification as the wrong signal level. Current is also in short supply in the microelectronic world: significant current, and hence electric power, is required to overcome a transistor gate's capacitance.

By the same token, a floating wire attached to the gate will not have sufficient electric power to drive it *all the time*.

On average the power of the electrical noise that occurs in a disconnected wire is very low. To overcome the noise and bring the floating wire to a definite level, be it 0 or 1, requires the amount of power much smaller than the power supply or ground provide via an open transistor. That small power can leak through

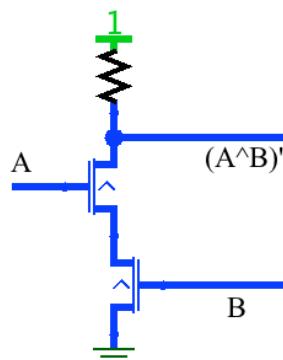
a *resistor* effecting the necessary pull. The term “resistor” means a device that passes only a small current through, as opposed to wire, which has no resistance to current and would pass as much of it as the power supply, or a transistor gating it, could provide and the consuming device at the other end could consume. Consequently when any level *is asserted* on the wire by a transistor, the power sunk into it is far greater and will quickly overcome whatever leak by the resistor.



The circuit represented as a square box on the left has an output connected to a pull resistor (the zig-zag line), the other end of which is connected to a fixed value (0 in this case).

The value towards which the resistor pulls is usually asserted by either the power supply or the ground. In this case it is the ground (0). The box on the left need not assert 0 on such a *pulled* wire. If the box disconnects its output from the wire (by closing the gate of a transistor connected to the output), the random electrical charge on the wire will slowly sip through the resistor into the ground, until the wire reaches the ground voltage. Consequently to have a definite value on the wire, the box only needs to assert 1 on it — or disconnect, and the resistor will ensure that the wire drops to 0 all by itself.

Using a pull resistor may allow us to simplify gate circuits. For instance, here is an alternative implementation of the NAND-gate, which uses a pull resistor connected to the power supply (1) rather than the ground:



The ground (0) is connected to the source of the bottom transistor, the drain of which feeds the source of the top one. It is only when A and B are both at 1 that both transistors are open connecting the ground to the output and bringing it down to 0. In all other cases the ground is isolated and the pull resistor asserts 1 on the output wire in the absence of an active drive. A quick look at the truth table in section 10.2 reassures us that this is exactly how a NAND is supposed to behave.

So why not just use pull resistors everywhere? The reason is that by its very nature a pull resistor is slow. As far as the eventual value of the output is concerned, it will be correct whether the circuit is implemented in CMOS or only one type transistors are used and the rest are replaced by pull-up/pull-down resistors. The CMOS implementation tends to be faster since under CMOS the output is always driven by the power supply/ground and never by a weak driving force such as the resistor. Consequently, the correct values are established sooner, which directly affects the speed with which a Level 2 platform can execute instructions.

The advantage of pull resistors is rarely the reduction in the number of transistors, though it may be important to keep this as low as possible for power efficiency in special circumstances. Their utility stems from the fact that by using “default values” we may break the design into self-contained parts that assert values only when this concerns them, and that allows a group of resistors to maintain useful defaults at all other times. We will look at how this is done later.

## 10.7 Controlled buffer and transmission gate

In the previous section we assumed that a Level 1 component had the ability to assert 1 or disconnect, and we relied on the pull resistor to provide the value 0 when the device actually disconnected. A much more common situation at Level 1 is when several devices can assert a value on the same input, and the designer wishes to choose which one will succeed in doing it (recall that asserting different values on the same input at the same time is forbidden by the rules of Platform 0). This is achieved by using a *controlled buffer* with the following “truth table”:

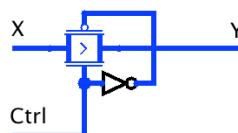
Ctrl	X	Y
0	0	Z
0	1	Z
1	0	0
1	1	1

Recall, that Z stands for “floating”. There is a version of control buffer which is always open (Ctrl=1). It is called simply the *buffer*. It does not use the control signal and is fully analogous to a chain of two inverters.

Here is an (obvious) Platform 0 implementation of the controlled buffer:



The N-type transistor opens when Ctrl is at 1, and lets X through to the output. If Ctrl is 0, the transistor is closed and Y is isolated from X. A set of  $n$  controlled buffers may be used to select which one out of the  $n$  devices is allowed to assert its signal on an input to another device. We could leave the analysis of the controlled buffer at that in the idealised world of electronics, but we must not forget about the limits that any idealisation imposes. An N-type transistor in PTL (which is what we have above) is, as we know, subject to signal degradation described in section [\\*9.3](#) when X is high. The way to avoid it is to use what is known as the *transmission gate*:



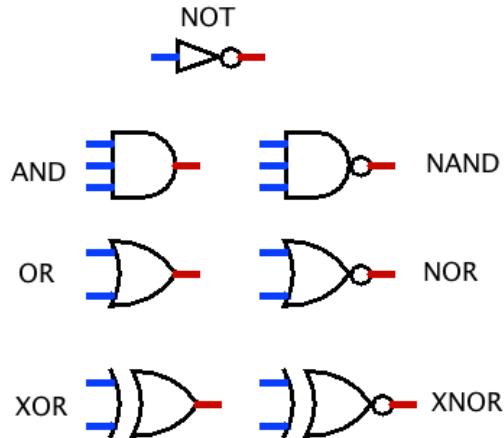
The transmission gate is two transistors, an N-type and a P-type, connected in parallel (sink to sink and drain to drain), with the two gates driven at all times by opposite values. The triangular component at the bottom of the diagram is the standard inverter, which delivers inverted Ctrl to the gate of the P-type transistor.

When the N-type gate is open, so is the P-type and the same is true when the gate is closed. When the transistors are open, any value of X makes one of the transistors pull Y hard towards it and the other transistor does so less effectively. As a result a weak conflict ensues (of the same kind as we described in the analysis of the 9 transistor XOR gate on page [207](#)). The transistor operating in perfect conditions easily wins the weak conflict and the observable output in such a case shows no significant signal degradation.

Our chosen circuit simulator, Logisim, treats transistors, and by extension transmission gates, as unidirectional devices: the signal is copied from the “input” to the “output” only. In real life an open CMOS transistor is *bi-directional*. This means that when it opens it can be thought of (ignoring signal degradation) as plain wire conducting signals between the source and the drain both ways. A transistor/transmission gate will even pass a floating signal in either direction in the absence of an active drive, just as the wire would. This should be borne in mind whenever transmission gates are used in complex designs.

By contrast, we will only require the controlled buffer (and never a transmission gate acting bidirectionally) in the rest of the text, and buffers will only be used with a non-floating input signal. The Logisim’s inadequate simulation of transistors should, however, be born in mind when attempting some of the circuit-design exercises that accompany the book.

## 10.8 Gate symbols and circuit structuring



The above drawings represent standard American notation of circuit elements at Level 1. Logic gates come in pairs, a gate and its output inverse: AND vs NAND and OR vs NOR. The last pair is named slightly differently: the inverse of XOR is usually referred to as XNOR, not NXOR.

The inversion of the output is indicated by the small circle placed directly on the output terminal. The inversion of an input is also possible and is indicated in the same way<sup>3</sup>. It is easy to distinguish gate shapes in circuit diagrams: the AND gate is round, the OR gate is pointy, and the XOR gate looks like the OR gate except the inputs come to an additional arc symbolising exclusion. There can be any number of inputs greater than one to AND/NAND, OR/NOR and XOR/XNOR. We did not discuss three and more inputs to a XOR gate in section 10.4 since the design did not seem to permit a natural extension. Yet there is always an option to express a three input XOR via a circuit that uses two 2-input XORs (by directly implementing  $(A \text{ xor } B) \text{ xor } C$ , and similarly 4- and more input versions, which may be inefficient. Fortunately, we do not require such multiple input XORs anywhere on our journey back to Platform 2 so we shall not dwell on this any longer.

Our goal is to build CdM-8 Platform 2 from the elements of the Universal Platform 1, and we will use logic gates to build parts of Platform 2, which will eventually be joined into a single circuit that supports the Platform 2 instruction set. This task cannot be completed in a single step; we must first construct the Level 1 functional blocks that perform core functions of the platform. These are built from logic gates, and those designs form our next stop. Since our designs will soon become larger let us first take a look at the structuring facilities of the circuit language.

### 10.8.1 Logisim chips

Building circuits is somewhat similar to writing programs. The designer first identifies units of functionality that have to be implemented. Those could be implemented right away or used as placeholders in an enveloping design. The design process may start at the bottom when units that do not contain further units as components are built first (a bottom-up design), or it may start at the top. In the latter case the circuit of the whole design is drawn first, using components not yet available. Then those are built with Platform 1 elements, standard Platform 1 circuits or further units to be implemented next (a top-down design). The designer may even decide to combine bottom-up and top-down strategies in her work. Whichever approach is chosen, for a large enough circuit it is predicated on our ability to use hierarchical abstraction for more than *platform genesis*, i.e. creation of platforms on top of other platforms. We may, and often will, need a *design hierarchy*.

With software, design hierarchies are well supported by the platform itself, primarily by structuring monolithic code into a hierarchy of subroutines, and using platform tools: the assembler for separate compilation and the linker for linking self-contained object files. The same is true of hardware design. Even in simulation,

---

<sup>3</sup>The easy way in which a NOT gate can be added to any gate terminal should not be abused. For example, if not careful one may end up with drawing two circles on terminals of the AND-gate without realising that this would always be equivalent to a NOR-gate, which is easier to read and interpret.

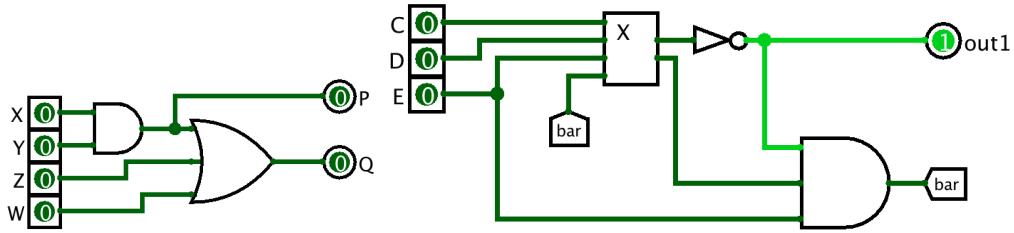


Figure 10.1: example of chips and tunnels. Left: chip X. Right: a design using chip X and employing a tunnel named ‘bar’ for connecting the output of an AND-gate with a chip X input.

Logisim permits definition of self-contained subcircuits of a given circuit, which we loosely call *chips*. Chips are designs in isolation. Newly designed chips can be used in larger circuits as components like gates and other standard elements. Similarly to what the Cdm-8 linker does, Logisim supports linking of circuits, i.e. using one circuit as a library for another. Chips can be fetched from a library repeatedly and used in different parts of the design. The reader would do well to familiarise herself with relevant Logisim documentation, since none of the Logisim specifics will be discussed here, except for chips and tunnels.

### 10.8.2 Logisim tunnels

In programming we take special care to ensure high readability of code. This is achieved by:

- formatting the code carefully to emphasise its control structure
- use of meaningful names for variables to make clear their intention
- making good use of hierarchies.

Turning to the hardware design, we state that the last bullet point is already addressed in the previous section. The other two require special considerations as far as hardware design is concerned. Although circuits are little more than connection schemes for participating components, care should be exercised in placing the components and drawing wires between them. Just as carelessly written programs become incomprehensible to their author a few months (or even weeks) later, to say nothing of an unfortunate external maintainer who is not the author, circuits, too, have the uncanny ability to look like some abstract art criss-crossed with wires of unclear intention and requiring the talent of a cyber-sleuth for determining how they work (and why this one does not).

None of this aggravation is inherent in circuit design. In fact it is relatively easy to maintain sound discipline in drawing rather complex circuits. The placement issue is the easiest to address:

- Group the components that work (mainly) together in one place of the design board. If it helps, display a title nearby to remind yourself of the intention and/or function of the part of the circuit.
- Whenever possible arrange the components so as to avoid unnecessary wire intersections. Nothing impedes readability worse than the need to follow a long wire which turns and twists, and which intersects with many other wires, making a connection with some. Remember that gates and transistors can be drawn in any orientation, as can most of other useful units.

The second suggestion is much easier to follow by exploiting the ability of the circuit editor (Logisim as well as most industrial ones) to hide wires from view. This is made possible by so called *tunnels*. They can be thought of as wire conduits placed under the design board. Since there can be any number of such conduits, each has a name and carries attributes of a wire bunch (wire bunches will be discussed later). At any point in the circuit diagram the designer can call up a tunnel head by drawing the corresponding symbol and supplying the tunnel name. All identically named tunnels are connected together by Logisim underneath the board. Bear in mind that tunnel names are scoped: they are considered by the circuit editor to be local to a chip. Different chips may use identically named tunnels, such tunnels are treated as different by the circuit editor.

Coming back to circuit structuring, tunnels compete with explicit wires as methods of showing connections between points on the design board. The recommended discipline is to use tunnels for semantic rather than graphic reasons: a tunnel is justified if the signal it carries can be given a meaningful name. That in turn suggests that tunnels are more appropriate as elements connecting meaningful units of design (which in a way complements structuring into chips).

We finish this section with an example of chips and tunnels. The left-hand side of figure 10.1 shows a small chip X having 4 inputs and two outputs. It is an isolated subcircuit under Logisim, and its terminals are named for ease of identification. The square ones are input terminals and the round ones are output ones. The right-hand side shows the use of the chip X as a component in a larger circuit. The connection also involves a tunnel named `bar` which connects the output of the AND-gate with an input of the chip. The design is shown here merely as an illustration. It does not perform any useful function.

# Platform 1 circuits

The Universal Platform 1 has logic gates as its building blocks. The purpose of the platform is to support circuits that connect gates into *functional units*, each of which consists of several gates. Such a unit is self-contained and serves a purpose that can be understood without reference to its circuit, for example, a unit that computes the sum of two numbers, or a unit that holds a bit string inside which can be retrieved later. Functional units may be, and often are, connected into larger circuits to build a more complex functional unit and eventually a collection of functional units is connected into a circuit that represents a whole Level 2 platform. From that point on, software kicks in, and much of the rest of the story we know already.

With this in mind, we begin surveying major functional units that we need at Level 1 in order to build our own CdM-8 Platform 2. Two major classes of units will be exposed: combinational and sequential. Those classes are completely universal as well, and they are needed for the construction of any Level 2 platform.

- **Combinational units** are *pure* functions: their outputs follow their inputs in that at any given time the output value may be expressed by evaluating a logical or arithmetic expression on its input values.

To be precise, the outputs follow the inputs with some tiny time delay. This means that if the inputs change at some moment  $t$  in time, the outputs at time  $t$  will still be the same, but eventually (and quite quickly, we are talking sub-nanosecond times in the current technology) at some time  $t + \delta t$  the output will change to reflect the new values of the inputs (assuming that those new values cause the output to change at all).

A functional unit that flips all bits in a bit string is a combinational circuit. A functional unit that combines two bit strings representing numbers and gives a bit string representing the sum of the two as its output, is also a combinational circuit.

- **Sequential circuits** do not compute. They are storage units that hold values until they are changed by new data arriving at the input at a certain time. Sequential circuits are therefore units that possess an internal *state* which can be changed by *commands* received on their inputs. We will study them later. For now we will remark that the assortment of primary components exposed in the previous chapter is not 100% complete.

Platform 1 contains further elements used for data storage and data communication. We will return to our building blocks when we need further elements for our purposes.

## 11.1 Adders

Our first port of call is a functional unit for adding numbers: an *adder*. Addition of binary numbers of fixed length is a core arithmetic function, without which a computing system cannot operate. The rest of the arithmetic derives from addition, more or less efficiently, requiring specialist circuits or not, but addition is completely unavoidable. Let us start with the simplest form of addition: bit-sliced addition of single-digit binary numbers.

As we saw earlier when we dealt with data representation, bit-sliced addition requires three inputs: the carry-in  $C_{in}$  and two operands  $X$  and  $Y$ , and it produces a carry out  $C_{out}$  and the result  $R$ . For one-bit numbers  $X$  and  $Y$ , the relation between inputs and outputs is as follows:

$$R = X \oplus Y \oplus C_{in}$$

and

$$C_{out} = (X \wedge Y) \vee (C_{in} \wedge X) \vee (C_{in} \wedge Y).$$

The first equation states that  $R$  (the output) is the sum of the operands and the carry-in modulo 2. The value of  $R$  is 0 if all three are zero, or if any two of them are ones, but not the third one. The value of  $R$  is 1 when just one of the three inputs is 1 or when all of them are 1. This is quite consistent with what we saw in section 3.5.3, where we discussed addition of binary numbers.

The second equation means that  $C_{out} = 1$  if at least two of the three inputs are ones. Which, again, is consistent with what we know about adding binary numbers:  $1+1=10$  in binary, causing a carry-out of 1.

Rather than implementing the adder directly from the above equations, it is easier and, interestingly, more efficient (i.e. a lower transistor count) to decompose the full adder into so-called half-adders.

### 11.1.1 Half-adder

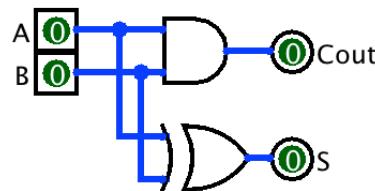
A half-adder has *two* inputs,  $A$  and  $B$  rather than three, but the same two outputs: the sum  $S$  and the carry-out  $C_{out}$ :

$$S = A \oplus B$$

and

$$C_{out} = A \wedge B.$$

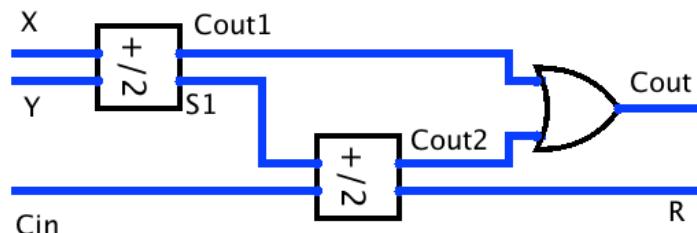
Its implementation requires an XOR-gate and an AND-gate — but that is all:



The new circuit elements here are Level 0/1 terminals (also known as “pins”): the square ones are for input, the round ones are for output. Each pin has a name which helps to identify it when the circuit is abstracted away as a single aggregate. A circuit complete with a full set of pins and having no unattached wires is usually referred to as a *chip*. We have just built a half-adder chip from gates.

### 11.1.2 Full adder

We use two half-adders to construct a full adder. In the diagram below we represent each half-adder as a square box labelled  $+/\sqrt{2}$ . Two half-adders are combined thus:



This is how the full adder works. The left-hand half-adder produces signal  $S1 = (X \oplus Y)$  and  $C_{out1} = X \wedge Y$ . The right-hand half-adder then outputs  $S1 \oplus C_{in} = (X \oplus Y) \oplus C_{in} = R$ . The OR-gate computes

$$(X \wedge Y) \vee [C_{in} \wedge (X \oplus Y)],$$

which is equivalent to

$$(X \wedge Y) \vee (C_{in} \wedge X) \vee (C_{in} \wedge Y) = C_{out}.$$

(Use truth tables to validate the equivalence)

We conclude that the above design is correct.

## 11.2 Wire bunches and gate arrays

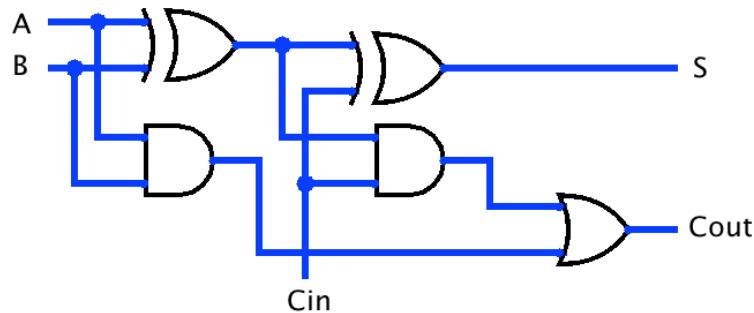
Circuit designs possess a large degree of homogeneity: they are built by replicating some basic fragments the number of times that correspond to the size of the machine word, memory cell size,etc.

Drawing circuit diagrams of this kind is tedious, because they involve a great many symbols that are repeated because the same task is being performed by many circuits in parallel. This also makes them hard to read.

To make it easier both to define and to understand such designs, gate symbols may be interpreted as *gate arrays*, i.e. several identical gates indexed by a nonnegative integer number. Consequently, each incoming and outgoing wire to such a gate becomes a *wire bunch*, i.e. also an array, similarly indexed.

It is important to understand that the replicas are not connected with one another in any way at all. All connections between gates in an array, or wires in a bunch, **must** be made explicit, and are shown by splitting a wire bunch into individual wires and connecting them as appropriate. On the other hand, same-size wire bunches may be thought of as solid wires when they are connected bunch with bunch using the correspondence between wire indices. Such connections are useful when two gate arrays of the same size are brought together.

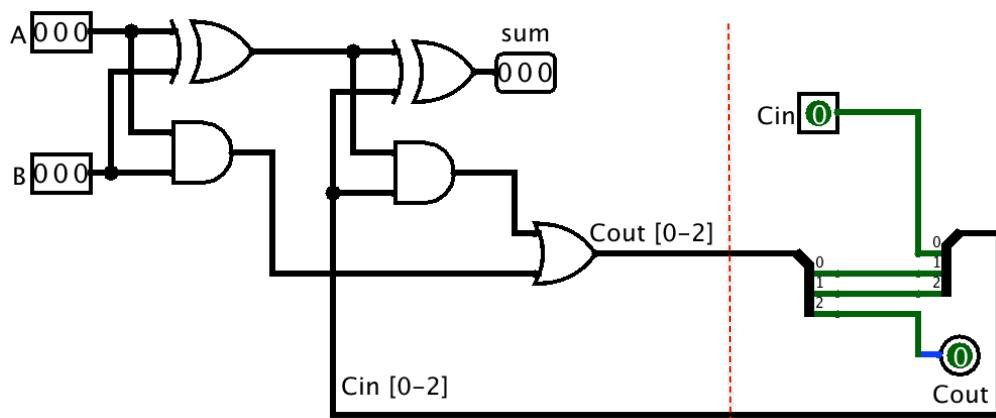
Let us see how arrays and bunches work by building a 3-bit adder from gates. Our first step is to redraw the circuit on page 215 in more detail by expanding the half-adders into gates:



Now let us recall how 3-bit numbers are added:

$$\begin{array}{r}
 & \text{A2} & \text{A1} & \text{A0} \\
 + & & & \\
 & \text{B2} & \text{B1} & \text{B0} \\
 \hline
 \text{C}_{out2} & \text{C}_{out1} & \text{C}_{out0} & \\
 & = \text{C}_{in2} & = \text{C}_{in1} & \\
 \hline
 \text{S2} & \text{S1} & \text{S0} & \text{C}_{in0}
 \end{array}$$

The corresponding circuit is shown in figure 11.1. We require three 1-bit adders. The carry-in for the adder numbered 0 is the carry-in of the whole 3-bit adder, and the carry-out of the adder numbered 2 is the carry-out of the whole 3-bit adder. The internal connections identify the carry-out of one 1-bit adder with the carry-in of the next. Now let us assume that each gate is in fact an array of three gates with the inputs and outputs being wire bunches rather than individual wires.



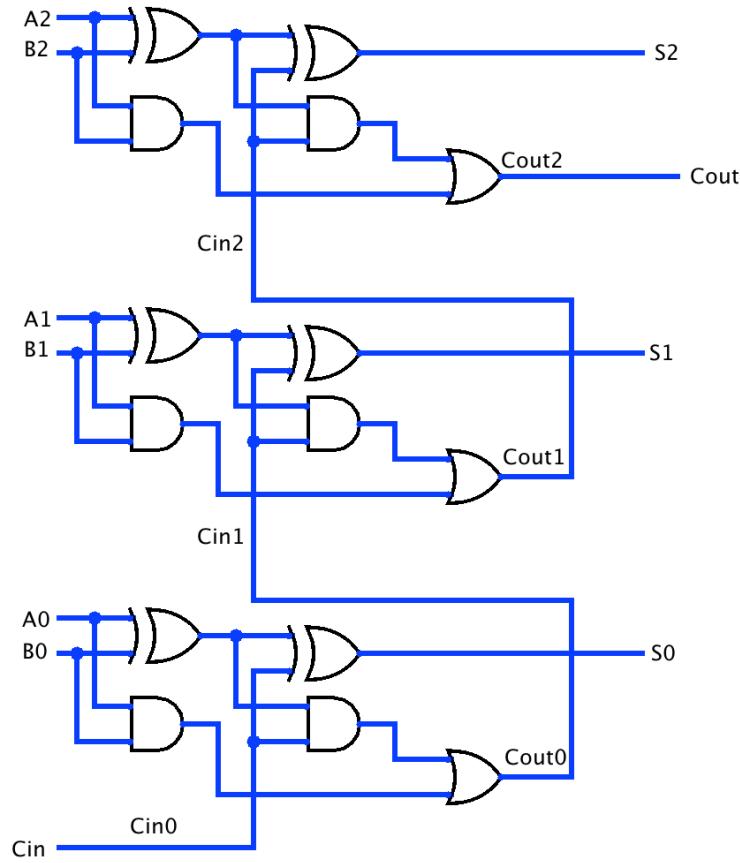


Figure 11.1: 3-bit adder laid out using individual wires

At first glance, this diagram is little different from the single bit adder on page 216, at least until we follow the signals to the right hand side of the diagram. The difference is that the lines connecting the gates are bunches of wires numbered from 0 to 2.

All gate symbols stand for *arrays* of individual gates, with the array size matching the sizes of all wire bunches. Which wire goes to which element of the array is determined by the index. As a result the diagram on the left of the vertical dashed line (which is merely a marker we have placed there and which has no effect on the circuit) represents three independent 1-bit adders, whose carry inputs and outputs are bunched together and brought over to the right hand side. There we encounter a new circuit element: a *splitter*, which is simply an unbundling of the bunch into individual numbered wires:

As the diagram shows, the  $C_{in}$  of the circuit is connected to the 0th adder's  $C_{in}$  input, the  $C_{out}$  of the circuit is identified with the 2nd adder's  $C_{out}$ , and the rest of the  $C_{in}$ 's and  $C_{out}$ 's are connected as per previous diagram.

It is not immediately apparent from the circuit diagram that all lines in it are in fact wire bunches, and that all of them have the same size.

The tool that we use for circuit design and simulation in this book is LogiSim, and it uses a somewhat peculiar circuit notation. Bunches in LogiSim are shown in black, whereas individual wires are shades of green or alternatively blue/red if floated/conflicted.

In paper diagrams colour is not used; when it is necessary to show that a line represents a wire bunch rather than an individual wire, the line is crossed with a short oblique stroke and the number (representing the bunch size) is placed next to it. We will not follow this notation to avoid the confusion between it and the LogiSim interface. You should bear in mind that the LogiSim circuit tool allows you to examine the attributes of a bunch or a gate array with ease, so further annotations would only encumber the view.

## 11.3 Combinational circuits: general design

The question that immediately comes to mind is how did we work out which logic gates to use and how to connect them to one another to produce our design?

Building an adder from half-adders is in fact an optimisation, since we could produce the full adder right away from the functional description on page 214. Indeed the latter contains all the information about the functionality required.

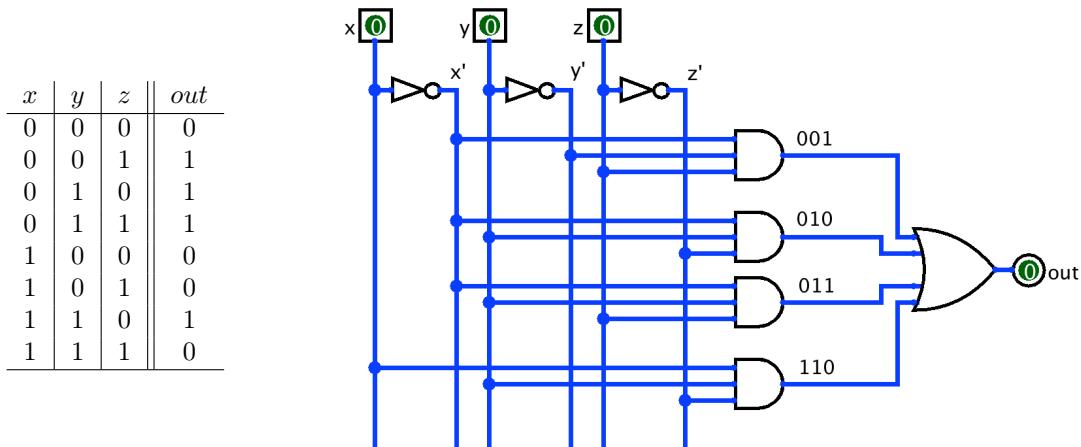
What we are looking for is a *method* for developing a combinational circuit given a formal description of the functional relationship between its inputs and outputs. This is an example of so-called *circuit synthesis*, an area that requires a combination of formal models and methods and an engineering discipline for meeting stringent requirements of modern electronic technologies: power dissipation/heat extraction, time delays, and sometimes more specific microelectronic parameters, such as power limits for CMOS drains, etc. For this Computer Science text, the engineering factors will at best be touched upon briefly, or otherwise completely ignored, but it is important to bear in mind that the material that follows is something of an idealisation. It does, however, faithfully reflect the structural aspects of electronic design.

Circuit design, as any design would, goes from a specification of what is required to the description of a solution. In the case of combinational circuits, all that needs to be specified is the output values given the input values, i.e. the functional dependency of the former on the latter. If the circuit has  $n$  digital inputs, on which 0's and 1's can be asserted in any combination, and  $N$  outputs that depend on them, it is sufficient for a complete specification of the circuit behaviour to provide a table that lists an  $N$ -bit string against every one of the  $2^n$   $n$ -bit strings defining the input values.

A circuit that has more than 7 inputs would require a table more than 100 rows in size which makes it impractical for any unassisted design. Industrial strength tools do not offer such tables to the designer as a specification vehicle. Still, for small circuits (and minimalism is in the spirit of the whole CdM-8 ecosystem) it is possible to use tables as a basic design methodology, which we will do next.

### 11.3.1 Implementation of truth tables

A table that has only one output listed against all combinations of 0's and 1's for the inputs is conventionally called a *truth table*. Indeed it is the truth table known from logic under the standard interpretation: 0=**false** 1=**true**. Each row of the table defines the output of the circuit under a unique combination of its input values. Let us start with an example of a circuit with 3 inputs and its implementation:



Note that for each row of the table on left **that yields a 1** there is an AND gate on the right. Each input of the AND gate is driven either by the design's input signal (one of  $x, y, z$ ) or its inverse ( $x', y'$  or  $z'$ ). When the value in an input column of the truth table is 0, the corresponding input is inverted, otherwise it is taken unchanged. For example, the top AND gate is connected to  $x', y'$  and  $z$ , which corresponds to the row with inputs 001. That combination is indicated in the circuit diagram next to the gate, and similar combinations are displayed by every AND gate in the design. An interesting property of such a connection is that each

AND gate only yields 1 when the values of  $x$ ,  $y$ , and  $z$  are as indicated. Under any other combination, at least one of the gate's inputs will be 0, which will drive the output down as well.

As a consequence, if the combination of inputs is such that it does not match any one of the AND gates' labels, then this results in zeros asserted by all the AND gates and consequently zero coming out of the OR gate. Since the labels of all the AND gates are different no more than one AND gate can produce 1 at the output, and only in a case when the circuit inputs correspond to a row in the truth table where the output is 1. In all those cases the OR will also assert 1 on its output.

The table method does not guarantee optimality. For example, the majority of the rows in the table could have the output 1, in which case there will be quite a few AND gates in the design, whereas one could simply negate the output twice and use the first negation to obtain the truth table that has fewer 1's as output, consequently fewer AND gates. All that would be at the expense of a single NOT-gate (one CMOS pair), representing the second negation, which is a fraction of the cost of an AND gate.

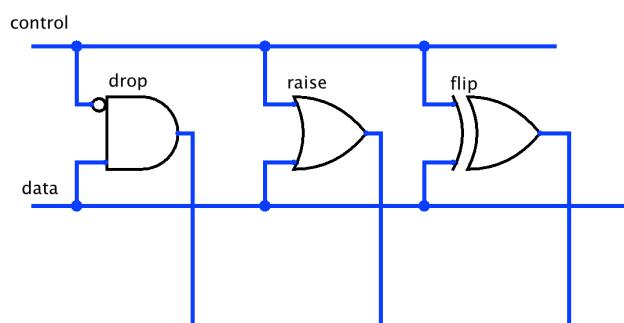
This is an example of so-called *circuit optimisation*, a highly mathematical and advanced area of technology which has produced very powerful tools for digital circuit design. Those issues go beyond the scope of this introductory book; we only note that, more than software, a hardware specification allows for a great many correct solutions with vastly different efficiencies.

The above universal method supports a completely mechanical implementation of a combinational circuit based on mathematical specification. In a way this makes the designer herself redundant. The specification can in principle be written in a "programming" language and compiled down to a circuit, or rather the component list and the connection table, by a software tool. This is exactly what happens in practical designs these days. The so called HDLs, or Hardware Description Languages, of which the two most popular specimen are VHDL and Verilog, are perfect tools for getting circuits produced by a machine. But so are high level programming languages, such as Java. Yet we write assembly language programs and compose circuit diagrams here, and there is a good reason for it: high-level tools obfuscate costs and underlying structures to their user, and it is those costs and structures that we are most interested in.

**Gates as controllers.** To help us see better how signals are manipulated by gates and how many gates of different types are involved, we focus on a very typical yet utterly simple task of manipulating one signal (call it a data signal) using another for control. In other words we are interested in a solution that *conditionally* changes a single-bit signal. There are only four ways in which such a signal can change:

- (a) not at all
- (b) become 1
- (c) become 0
- (d) become its inverse

Now to the control side. When one signal controls the other, this means that in the absence of an active control level (usually 1) the other signal (usually data) should be left as is, but when the control is active, then we require one of the changes (b-d) to occur. Option (b) can be described as *raising* the data signal on a condition, option (c) as *dropping* it, and finally option (d) as *flipping* it. Those control actions are extremely common in hardware design, so there should be, and there is, an efficient solution, which is also straightforward. We show it below for reference purposes.

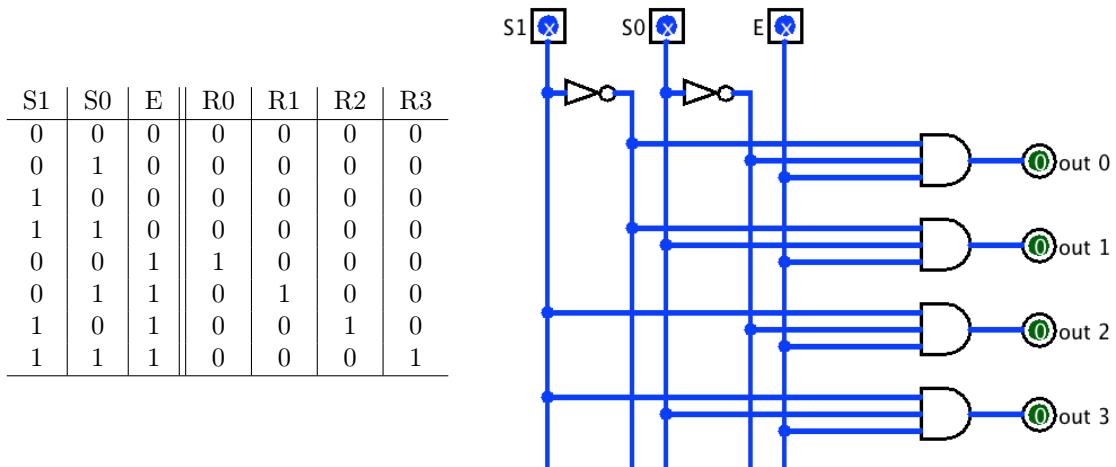


## 11.4 Decoder

This is a standard Level 1 component, which is used for selecting one wire out of a bunch usually for control purposes. The decoder has two inputs: one, the *selector*, is a bunch of  $k$  wires, and the other is a single-wire *enable* input. There is one output, which is also a bunch, and it contains  $2^k$  wires. The selector is interpreted as a  $k$ -bit unsigned number, and that number determines which of the output wires is raised when the enable is up. The rest of the outputs are driven to zero. When the enable is down, **all** of the outputs are driven to zero. A decoder with a  $k$ -bit selector is commonly referred to as the  $k$ -to- $2^k$ -decoder. There exists a version of decoder without the *enable* input; it works as if it had one permanently held up.

A decoder is a combinational circuit with more than one output, but it can also be defined using truth tables. Rather than creating  $2^k$  different truth tables (one for each output wire) we construct a single table, with  $k+1$  inputs to the left of a dividing line (the last one being the *enable* signal), and  $2^k$  columns to the right of it (one for each output wire).

For example, a 2-to-4 decoder is defined as follows:



The right hand side above shows a direct implementation of the truth table. Observe that each output column only has a single 1 in it, and consequently a single 3-input AND-gate is added to the design for each output. There is no need to introduce any OR-gates since the output of each AND-gate implements a whole column by itself.

## 11.5 Multiplexer

A Multiplexer or mux is a very important part of the Universal Platform 1. It has  $2^k$  m-bit inputs and a  $k$ -bit selector indicating which input to pass through to the single m-bit output. An implementation of an 8-bit 4-to-1 mux is shown in figure 11.2.

Here we can see familiar structures once again. The left-hand side of the design contains the now easily recognisable 2-to-4 decoder, which connects to four 8-bit bunches asserting eight 1's on the bunch that corresponds to the selector value given in  $(S1, S0)$ . The new element here is the OR-gate array on the right bringing the four separate wire bunches into one.

If the selector value is such that a given AND-array has an 8-bit zero on its lower input, its output will also be an 8-bit zero and that does not affect the OR outputs, since for any  $x$ ,  $(0 \vee x) = x$ . Consequently only one AND-gate array will determine the output of the OR, and that is the one that receives 1 from the decoder. For example if  $S1=S0=1$ , then the 8 bits of D3 will appear at the output terminals.

Notice that the design of the multiplexer almost breaks down into circuits strongly connected to a single “data” input. There are four such circuits in our example in the figure. There are two vertical wires on the left which are used by the selection AND-gates. If we wanted from 5 to 8 inputs, we would have to use three vertical wires, slightly larger AND-gates (3 input as opposed to 2-) but a drastically larger OR-gate, whose size would be proportional to the number of data inputs, as would be the *number* of the other gates.

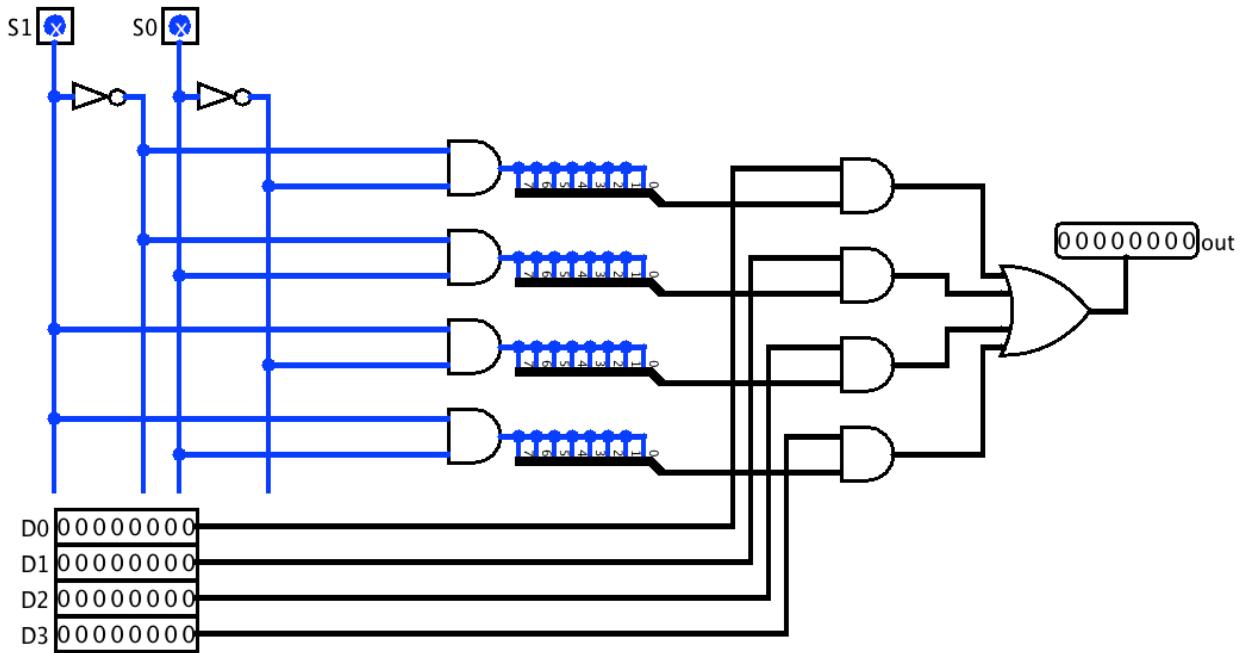


Figure 11.2: 8-bit 4-to-1 mux

The wire structures on the left are typical of platform architecture: a collection of wires that serve *all* components of the system (or a large part of it) with data, a many-to-one or one-to-many connection. They are called *buses* from Latin *omnibus*, meaning “for all”. The set of vertical wires on the left of figure 11.2 is, in fact, a *selector bus*.

## 11.6 Sequential circuits: an RS flip-flop

The circuits we have looked at so far are combinational: the outputs are functionally dependent (i.e. are a mathematical function of) the inputs *at the same time*, ignoring, as we have done so far, any propagation delays.

There is a second category of circuits, which are no less important for building computing platforms: sequential circuits. The difference between combinational and sequential circuitry is very dramatic:

- the outputs of a sequential circuit do *not* follow its inputs at the same time. Instead they follow the circuit’s *internal state*.
- The internal state of the circuit is established when the platform is reset (powered up) and may change after receiving a *command* on any of the inputs. A command is a signal, occurring at a certain time, that tells the circuit to change its state in a certain way.
- A sequential circuit must allow repeated application of the same command to ensure that state can be changed as many times as necessary.
- As long as no new command is received, a sequential circuit must *hold* its internal state.

The smallest sequential circuit is one that holds exactly one bit of internal state. The circuit needs at least one output that indicates which value is held inside: 0 or 1. It also requires two inputs in order to receive two commands: change to 1 (called *set*) and change to 0 (called *reset*).<sup>1</sup>

<sup>1</sup>you may be wondering whether these could be just one command “change” and a data parameter (0/1); the answer is yes, but the difference between the two sets of inputs is functional and can be bridged over by a trivial combinational circuit.

A command can *not* simply be asserted on an input wire as a signal level, 0 or 1, since under such conditions the command will have no reference to time, i.e. it will not be possible to tell *when*, over the duration of that level, the event of receiving the command actually happened. We conclude that a command has to be represented as a signal *transition*, i.e. a *change of level*: from 0 to 1 or from 1 to 0. The point at which this change is detected defines the command's arrival time.

It is easy to see that on a single wire a repeated issue of a command is only possible in two cases:

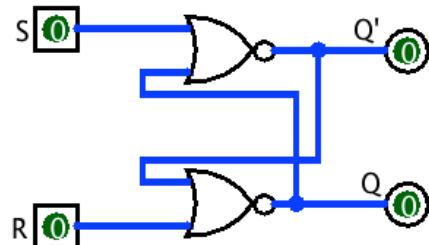
- when a command is represented as a transition either way (0 to 1 or 1 to 0), or
- when a command is represented as two transitions back to back (e.g., 0 to 1 followed by 1 to 0).

Indeed, after a 0-to-1 transition the input is at 1 and a subsequent 0-to-1 transition is impossible. Ditto for 1-to-0.

In the two-transition case the time reference of the command is either the first or the second transition, depending on what the circuit requires. This gives rise to the classification of sequential circuits as triggered by either the *rising edge*, i.e. the transition 0 to 1, or the *falling edge*, i.e. the transition 1 to 0. The single transition case is somewhat more expensive to implement (requires 5 more CMOS pairs to bring it to the second option) and as a result neither faster nor smaller. Consequently, it is not used in the simplest sequential circuits.

To summarise, we wish to build a sequential circuit with two inputs and 1 bit of internal state, which is asserted on an output terminal. Two-transition commands on the first and second input will change the internal state to 1 or 0, respectively. Such a circuit is known as an RS (Reset-Set) flip-flop, also knowns as a SR (Set-Reset) flip-flop. Since it holds on to its state between the commands, it is sometimes referred to as an RS (or SR) latch. Instead of insisting on "correct" terminology selected by arbitrary choice, we permit ourselves to be quite indifferent to it, as long as it is clear from the name precisely what circuit is meant.

This is what a classical RS flip-flop looks like when implemented using two NOR-gates:



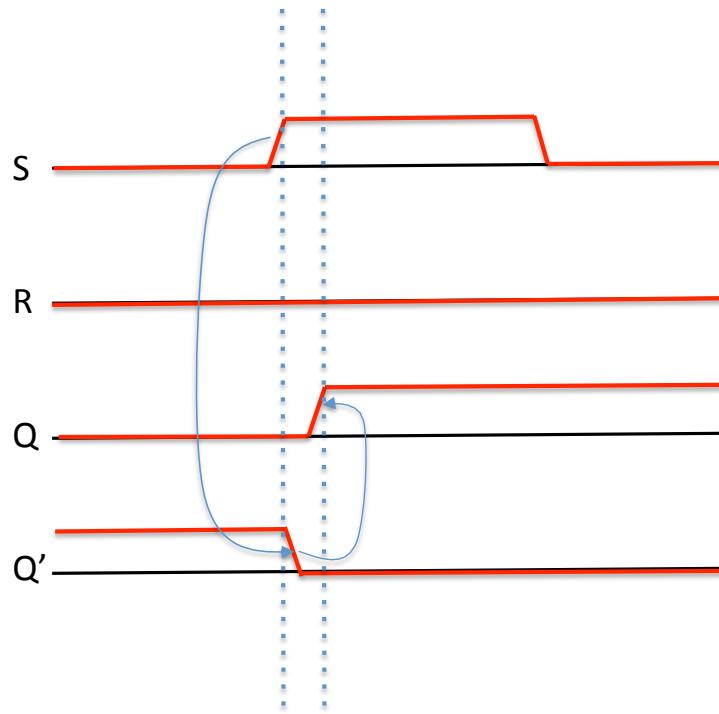
The circuit has two inputs, S[et] and R[eset] and accepts two-transition commands on either. A command on S brings the internal state up to 1, and a command on R brings it down to 0. If the flip-flop is in the state 1, the S command has no effect, and if it is in the state 0, then similarly the R command has no effect. The output Q follows the internal state and the output Q' is at any time the inverse of Q (not Q). The actual change of state occurs already at the rising edge of the command. Consequently the time interval between the rising and the falling edge only affects how soon the next command can be processed.

The novelty of this circuit from the construction point of view is the fact that it has *feedback*, i.e. signal propagation by a wire that connects an output *back* to the circuit input, so it is possible for a signal to travel over a closed circuit. This is something that is *never* possible in combinational circuits. To understand how the flip-flop works, we must write some very simple equations that reflect the logic of the NOR-gate:

$$\begin{aligned}\neg(S \vee Q) &= Q' \\ \neg(R \vee Q') &= Q\end{aligned}$$

Both of these equations must hold for the wires not to be in conflict. Do they hold initially, after the Platform reset? Indeed they do. Assuming that  $S = R = 0$ , we simplify the above to  $\neg Q = Q'$  and  $\neg Q' = Q$ , which agrees with the above-mentioned property that Q' is the inverse of Q. The equations do not constrain the value of Q though, it can be either 0 or 1, depending on which state the flip-flop finds itself in after the reset.

Let us assume that the initial state is  $Q = 0$ . This means that initially  $Q' = 1$ . Now let us raise the S wire to 1, and see what happens using the *timing diagram* below:



As the voltage on  $S$  rises to 1, the NOR-gate's output,  $Q'$ , obviously goes down to 0 irrespective of the other input, which is reflected in the diagram by the faint curved arrow between transitions. This is known as a *causal* arrow, it shows that one transition *causes* the other. Now since  $Q'$  is down to 0 and  $R$  remains 0, the output of the lower NOR-gate must rise to 1. Finally since  $Q$  is at 1 now (the second vertical dashed line marks the time when this happens)  $S$  may come down at any time from now on, and that will not change the output of the upper NOR gate,  $Q'$ , since  $Q$  is up ensuring that the output remains at 0.

We are at a stable point, and will remain there forever or until the next command. Both input signals are now down (the far-right end of the diagram), and the circuit is ready to accept a new command. We could start in the state  $Q = 1$  and receive the command on the  $R$  input; the situation being completely symmetrical, we would discover that the circuit makes the state transition to  $Q = 0$ . Doing this on a timing diagram as we did for the  $S$  line is left as an exercise to the reader.

It can be seen that if the flip-flop is in the state  $Q = 1$ , issuing a command on the  $S$  input will have no effect at all: since  $Q = 1$ , the output of the upper gate is 0 no matter what signal is present on the other input, and so no signal propagation will occur from the  $S$  wire. The same can be said of the command on the  $R$  line when  $Q = 0$ .

Notice that the second transition of the command cannot happen too soon. In order for the flip-flop to work correctly,  $S$ , raised in the state  $Q = 0$ , can only be dropped after  $Q$  has risen to 1. The time delay between the former and the latter is exactly two gate delays. It is in the tens of picosecond range for commercially available CMOS structures, but... it is still a finite amount of time which limits the number of state transitions that the flip-flop can perform per second to a few tens of billions. This is one of the factors that limits the speed of modern computers.

## 11.7 Pulses and edges

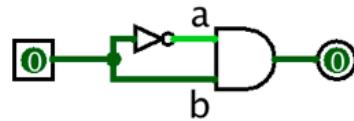
As we will see in the next section, the main purpose of an RS flip-flop is to serve as the core part of a latch that holds *data*. Before we go there, there is, another problem for which an RS flip-flop is a *primary* solution. The problem is to *link* events. In the previous section we saw how input signal edges caused output signal transitions in a flip flop, and drew some timing diagrams. What linked those transitions was a feedback network of two NOR-gates, which realised the behaviour of a flip-flop. In a complex platform which interacts with the outside world, those causative links between events quite often turn out to be more complex as well. They include decision-making, which involves dedicated hardware and even software. A description of

a set of causative links is called a *protocol*, and in order to implement it we need to be able to detect what transitions have happened on which wires. In other words, we need a technique of capturing and holding an edge-event, so that we may use combinational logic in order to work out what transition to put into effect on an output terminal, thus implementing a certain protocol.

An RS flip-flop is perfectly adequate to support protocols, except we need to figure out a method of converting an edge to a short enough pulse. A pulse, because an RS flip-flop is controlled by pulses, i.e. pairs of edges causing the signal to go up and then down. Short enough, because until the pulse is fully articulated, i.e. until the signal has risen and fallen on a wire, a new command for the same flip-flop must not begin.

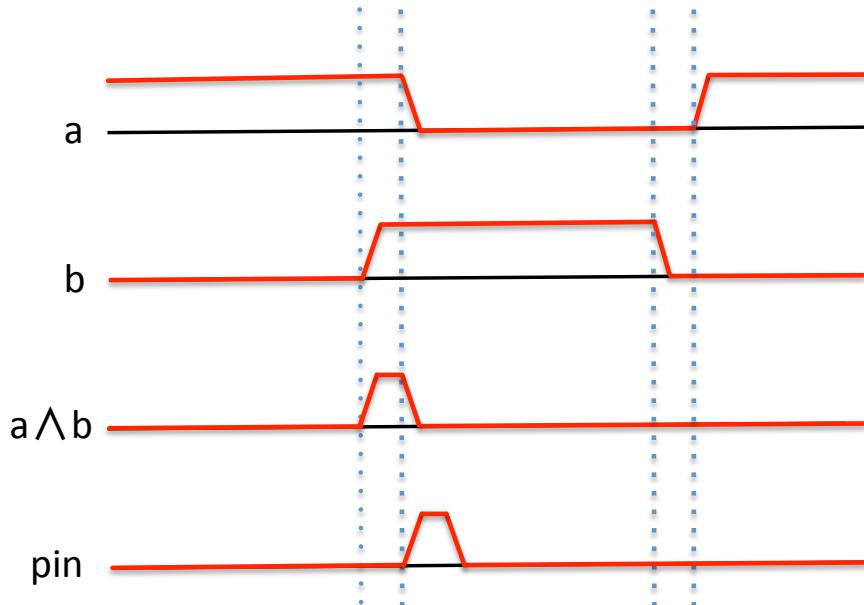
Specifically we require two solutions: one that converts a rising edge into a short pulse, and another that converts a falling edge likewise. We shall start with the former.

Consider the following circuit:



At first glance this circuit is completely nonsensical. Naturally, whatever the value of the input pin, the AND-gate will see both high and low on its input terminals and will respond with a low on its output; consequently the result is 0 no matter what input, and, it seems, the circuit as a whole can be replaced by a ground connection. Indeed, as a *combinational circuit* the above little block can be replaced by the ground wire connected to the output. However, it is *not* a combinational circuit, since what we want to use it for is not its stable output given a stable input, but it is behaviour in time.

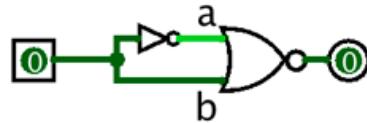
Here is an interesting timing diagram:



Assume that the input pin is at 0 initially, and allow it to rise. Since the terminal **b** of the AND-gate is wired directly to the input pin, it will rise *at the same time* as the pin. However the terminal **a** of the gate is connected via an inverter, it is initially at 1, and it must fall, but *a little later* than the rising edge of the pin. The delay is due to signal propagation across the NOT-gate, which takes some time. Now, the AND-gate sees both inputs at high in the brief interval between the rise of **b** and the fall of **a**, this lasts as long as a NOT gate's delay. Finally, the output pin is depicted in the last timing diagram above, and it is naturally the same shape as  $a \wedge b$ , except it is delayed by the signal propagation time across the AND-gate. If for any reason the duration of the resulting pulse is not long enough, it can be made wider by replacing a single inverter by 3, 5, etc., which will result in the output pulse being longer by the respective factor.

Notice that, although there is a similar separation of edges when  $b$  goes *down*, the AND-gate briefly sees both its inputs as being *low*. For an AND gate, there is no difference between one or both inputs being low; consequently no transitory behaviour is observed here.

To convert a *falling edge* to a pulse, a similar arrangement is made with a NOR-gate:



As before, it is easy to see that as a combinational circuit, this has the same effect as the ground wire connected to the output. As a sequential circuit, it delivers a pulse after the falling edge of the input pin. Demonstrating this on a timing diagram is left as an exercise to the reader.

## 11.8 Latches and the Clock

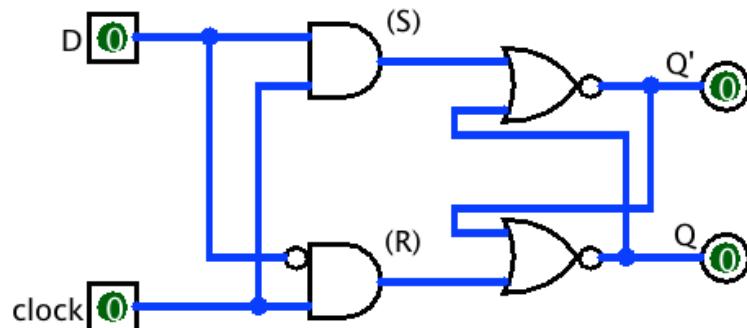
The RS flip-flop is a work horse of computer architecture, it is a stable circuit that has memory, and which can be used as a building block to create various circuits that hold data until it is changed by outside signals.

The advantage of the RS flip-flop is its simplicity and low cost, the disadvantage is that the input signals play a dual role: they define both the timing of the state change and its direction. Indeed, a command on the S input tells the flip flop to make a change *now* and to change to 1 (if it is not at 1 already). For construction purposes it is highly desirable to decouple different functions and to assign them to different signals. This is achieved by a clocked *latch*.

One of the most fundamental features of Platform 2 requiring a Platform 1 implementation is that of a *machine clock*. The clock is a signal that reifies time without mixing it up with data. The clock signal is simply a series of transitions 0 to 1 and 1 to 0 happening at equal time intervals.

The role of the machine clock is to deliver timing to all sequential circuits in the design, to let them “know” that they should make a transition now. This can be signalled by either edge of the clock, rising or falling. The machine clock is, as it were, the conductor of a large “orchestra” of sequential circuits. It ensures that they keep to the rhythm, whereas the combinational circuits define the “melodic lines”, the progression from one state to the next that each musician of the orchestra is playing.

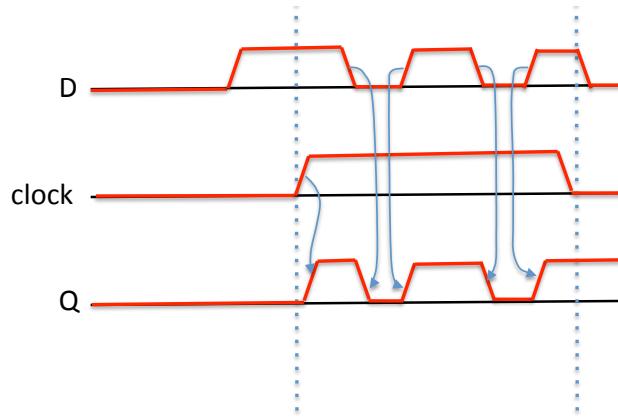
By choosing the clock interval to be long enough, the designer ensures that the time between two transitions of the same kind (rising or falling), which is also known as the *clock cycle*, is sufficient for all the *data* to propagate from the holding place across the combinational logic (such as adders, etc) back to the holding place where the results can be held (“latched”) before, and become available immediately after, the beginning of the next clock cycle. That is exactly what a latch does. It has two inputs, clock and D[ata], which is why it is often called a D-latch. Here is the circuit diagram:



This is how it works. The right-hand side of the circuit is the familiar RS flip-flop. The outputs of the two AND-gates drive the (upper) S and (lower) R inputs to the flip-flop. If D= 1, the next rising edge of the clock will be passed by the upper AND-gate through to the S wire. If D= 0, the lower AND-gate, due to the

inverted input, will pass the rising edge of the clock to the R wire. While the clock is up the RS part of the latch will interpret any changes in D as commands. If D goes down within the clock cycle, this will generate a command on the R line, and the state of the latch will change (assuming that D was up at the clock edge). It takes little thought to realise that the final state of the latch is determined by where D happens to be (0 or 1) just before the *falling* edge of the clock.

We observe here the two different functions of the input signals. D delivers data for holding, at some point before it is needed and the clock acts as a trigger. While the clock is up, the latch is transparent: the output Q follows D with some small delay. The moment the clock falls, the latch ceases to be transparent and latches in the last value of D.

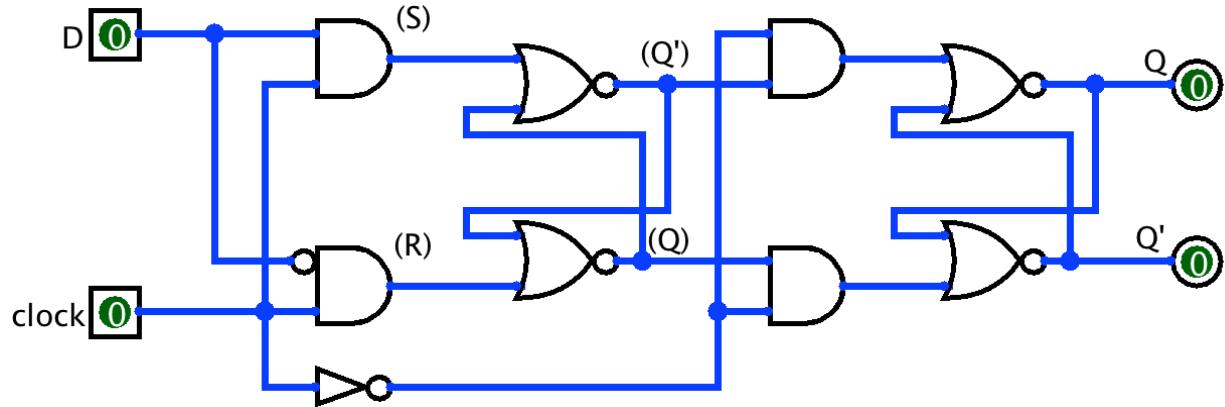


The rising edge thus merely indicates the *beginning* of the latching process. In a circuit devoid of feedback (other than the feedback we saw when we got acquainted with the RS flip-flop) the unstable behaviour of the latches presents no problem. Sooner or later the signals will propagate from their sources across combinational logic to the inputs of the latches and on the falling edge of the clock the new value will be stored. The question is: what signal sources are there that are stable enough to maintain the input signals on the combinational logic until the whole system stabilises and the clock cycle ends?

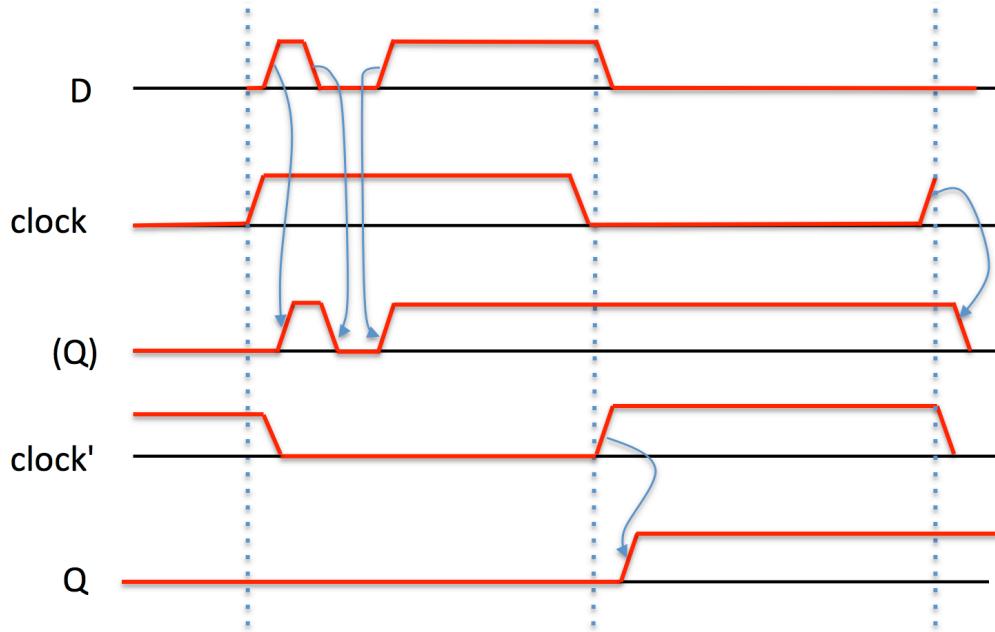
At this point we need to cast our mind back to the earlier chapters of the book. The Universal Platform 1 is there to implement some version of Platform 2. Consequently we can, at least speculatively for the time being, consider how the Platform 2 uses the circuitry to execute machine instructions. Registers are polled at the beginning of the clock cycle and the data is marshalled to the ALU, a combinational circuit, for processing. The output of the ALU goes back to the registers, as would be the case if we looked at, say, the CdM-8 instruction `inc r0`. The register `r0` mentioned in the instruction is the source of data. The data goes to the ALU for incrementation. While the ALU is at work (or rather while the signals are propagating across the ALU) we must not change the content of `r0` since if we did, we would change the inputs of the ALU. That way it would no longer be adding 1 to what *was* in the register, but actually what *is there* at this point. The result of such addition would be completely unpredictable!

The difficulty of this sort arises as a result of the *system-wide feedback* characteristic of a digital computer. The memory elements (latches in particular) are both the source and the destination of data; the data goes backwards and forward between the latches and the ALU (and other combinational circuits) requiring a phased discipline of state update. We shall consider how this is done next.

## 11.9 Master-Slave Latch



A master-slave latch (MSL) is an extension of a D-latch that adds another RS flip-flop at the end. The left-hand side of the circuit is an ordinary D-latch which works as we saw in the last section. The intermediate ( $Q$ ) signal follows the  $D$  input as long as the clock is up. Unlike the proper D-latch, this circuit does not connect the intermediate ( $Q$ ) straight to the output, but rather uses both the ( $Q$ ) and the ( $Q'$ ) after the first RS (called the Master) as a source of commands for the *second* RS (called the Slave). The subtlety is that those two signals are passed through two AND-gates which are fed the inverse of the clock on their other input (see the bottom wire in the above figure).



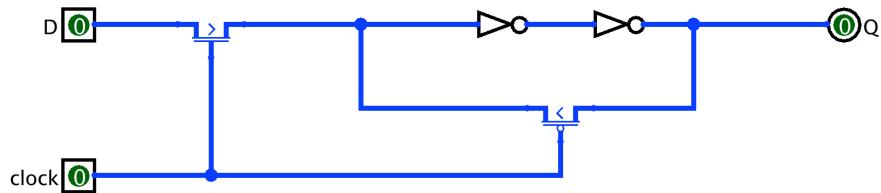
When the clock is up, the inverse of it is down, and the aforementioned AND-gates' outputs are at 0 no matter what. Consequently the Slave is not commanded anything and remains in its current state even as the intermediate signals ( $Q$ ) and ( $Q'$ ) are going up and down to follow the yet unstable input  $D$ , which stabilizes some time before the end of the high-clock period. When the clock falls, the tables are turned: first the Master latches  $D$  and becomes stable as long as the clock is down, and then (after one inverter's delay) the Slave becomes transparent and copies the stable input from the Master to its own output for the down part of the clock cycle. We conclude that the output of the MSL is stable *at all times* except during the falling edge of the clock when it latches the current value of  $D$  for the next clock cycle.

## \*11.10 A CMOS alternative

So far we have been strictly hierarchical in our treatment of the Universal Platform 1: gates are universal, they *may* be realised as transistor aggregates if CMOS is our choice or Platform 0, but could be built from anything else (light-pulse technology, for example). We forgot transistors as soon as we built gates, and only remember about our example Platform 0 when we wish to compare the efficiency of different implementations.

One must not be overzealous with hierarchies. Sometimes peeping through a level delivers very significant benefits. Designers of hardware often build specific transistor circuits that implement important aggregates normally constructed from gates in a much more efficient manner. Such optimisation is technology-specific: if we replace CMOS by, say, ECL (an old-fashioned bipolar transistor technology no longer used), none of our transistor circuits will be effective, but the gate implementation of the RS flip-flop will survive unscathed provided that all gates have suitable ECL implementations.

Let us take a look at the following CMOS circuit:



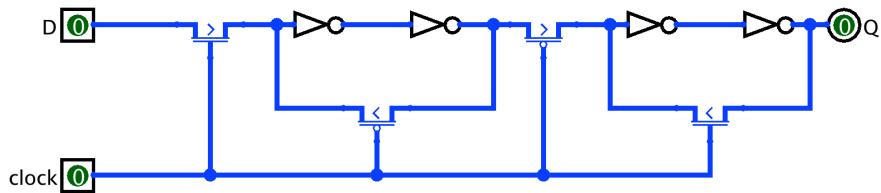
Here the two inverters should not confuse the reader: we mean the standard two-transistor implementation of the NOT-gate that we started Chapter 10 with. Consequently the circuit is an arrangement of only 6 transistors. Let us see how it works.

When the clock is up, the N-transistor connected to the pin D is open and the signal travels through the pair of inverters straight to the Q output without change. The value of Q follows the value of D with some delay. The second transistor does not introduce a conflict because it is a P-transistor and it is closed, isolating the output of the pair of inverters from the input.

Now let us think what will happen when the clock falls. The transistor on the west will close, isolating the pin D from the rest of the circuit. The other transistor will open and will feed the output of the pair of inverters back to its input. The feedback will sustain the last value of Q when the clock was still high until the rising edge of the clock.<sup>2</sup>

We have just described the behaviour of the D-latch from the section 11.8. The circuit there used two NOR-gates (cost 4 transistors each), two AND-gates, and one NOT-gate (cost 2 transistors). Totalling it up, we conclude that the cost of a single D-latch is  $2 \times 4 + 2 \times 6 + 2 = 22$  transistors if implemented following our platform hierarchy — and only 6 transistors as per circuit diagram above, if we peep through Level 1 and see CMOS transistors underneath.

Let us go one step further, and build a Master-Slave version of the D-Latch:

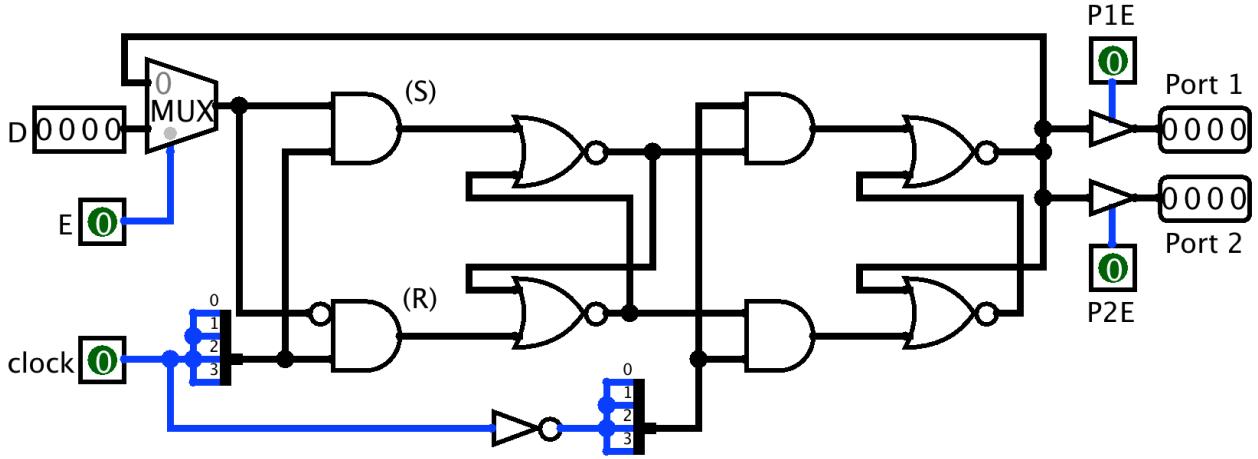


Notice that we avoided a NOT-gate between the master and the slave clock inputs by simply using the opposite types of transistor in the slave (a saving of 2 transistors).

These patterns are known to chip designers, and optimisations of this kind are done automatically by design tools.

<sup>2</sup>Notice that both transistors operate in PTL, rather than pulling up and down as they should, which will definitely result in signal degradation. However, in this ingenious circuit signal restoration is instantaneous: either transistor's output is put through a pair of standard inverters which produce a strong pull at their circuit output Q, making this solution completely safe in an arbitrary context.

## 11.11 Multiported register



We have finally arrived at the fundamental building block of Platform 2: the *register*. We have seen registers before, on a purely software level, and are now about to see how they are implemented in hardware.

The above circuit diagram is almost identical to that of the master-slave latch. There are, however, three important differences.

**Multibit storage.** First of all, the MSL is a single bit memory device. Its state is fully reflected by the single-wire output  $Q$ . As we saw during our study of higher-level platforms, the functionality of a computer requires data to be bunched into machine words, which are typically tens of bits in size (while only 8 bits for our tiny CdM-8 platform). All bits in a bunch are treated as a single item (content of the word). Consequently the timing of all bits in the unit of storage is *synchronized*: they are latched at the same time.

The above diagram displays a 4-bit register. This means that there are 4 MSLs in it working together in the same way under a single set of controls. The input  $D$  is now a 4-bit bunch, not a single wire, and all the gates that make up the two D-Latches are in fact size-4 gate arrays. Each gate in the array belongs to a separate MSL. The two four-way splitters in the bottom of the diagram simply deliver the clock to *all* the master and slave latches. This is how the circuit ensures that the timing of all commands is the same for each bit.

**Controlled latching.** The second new feature is the  $E[\text{nable}]$  signal. The MSL is a perfect device to latch a new value every falling edge but the problem is that there is not necessarily a new value to be latched every time the clock falls. Indeed if this register were to be used in a program, it would only be written into when a machine instruction mentions it as a destination register. That could happen once in many clock cycles. Consequently the register has to have the ability to hold its value for more than a clock cycle and this should be controlled by an external signal. The signal  $E$  is such a signal. When  $E$  is down the multiplexer in the north-west corner copies the current state (which is asserted on the  $Q$  output of the Slave) back to the  $D$  input of the Master. When the clock falls the MSL will simply latch its current value making no state-transition whatsoever. Not so when the signal  $E$  is up. The mux then passes through the data asserted on the  $D$  pin and it is that data that gets latched at the falling edge.

**Never gate the clock!** It is important to understand that not only  $D$  but  $E$  as well can be unstable up to just before the falling edge of the clock. This has no effect on the Master since it only latches data on the falling edge. One might be tempted to avoid the use of the multiplexer by gating the clock signal, i.e. by inserting an AND-gate which combines  $E$  and the clock so that when the latching is prohibited, the Master is simply not triggered. While apparently simple and economical, this would be a potentially disastrous solution. If  $E$  is derived from a complex combinational circuit which “decides” whether or not a particular register must be updated, the value of  $E$  might go up and down several times before it stabilises. This is due to the fact that gates in a combinational circuit combine signals arriving to their *multiple* inputs via different paths and so they may overtake each other due to the unequal delays which results in a repeatedly

changing gate output. If an unstable E is combined with the clock signal by AND, the result is potentially several clock cycles inside a single clock period! Each of these cycles will latch the value of D that happens to be asserted there at that time, i.e. a more or less arbitrary value though the final (stable) value of E may be 0 demanding no change of state. By then the current state may already have been overwritten, and thus lost, by a spurious trigger supplied by an unstable E.

There is another good reason why gating the clock is a decidedly bad idea. A clock edge is supposed to arrive to all sequential devices *at the same time*. The edge arrival signifies the beginning (or end) of a clock cycle, which marks one step of the computation performed by the platform. In reality small delays, such as the *wire* propagation delay on the lines connecting up silicon components are not insignificant and can be dominant at high clock frequencies; what's worse, since components are placed in space at different distances from each other, those delays also *vary* a great deal. For the latching to work correctly, *all data* must be stable by the time the *fastest* clock edge arrives to any component, since it is generally unknown which part of the new data may be latched then. The greater the variation of clock propagation delay the longer clock cycle is necessary to account for those variations. Gating the clock introduces further variation to the edge arrival delay and so should be avoided.

**Multiple tri-state outputs** Finally, the last new feature appears on the far right of the diagram. Notice the two controlled buffers that connect to what was the Q output of the MSL (we do not need the other, Q', output, so it is not shown in the circuit). The need for this merits a brief explanation.

Programming in CdM-8 assembly language already gives sufficient understanding of the kind of thing a register is. One major feature of registers is that not only do they contain multiple bits, they are also used collectively, as a register array indexed by the register number<sup>3</sup>, i.e., r0, r1, etc. This means that data storage or retrieval may potentially require all registers, with the one being used selected by some selector signal. We can easily decode the index of the register array using a standard decoder (1-to-4 for CdM-8) and drive the corresponding E signal high to latch a value off a bus. However if we are to assert a value on a bus we need the ability to control whether or not a given register does the assertion (or else all of them will, creating a conflict). Finally if it is more than one bus that we wish to be able to assert the register content on, we need more than one control signal and more than one controlled buffer.

You may be wondering why not use a multiplexer at the output as well, to select one of the many registers' outputs for the data consumer?

The answer to this question lies in the fundamentals of computer organisation which we will see in the next chapter. The main destination of the register output is to serve as a data source for the ALU. Since the ALU has the ability to perform operations on two operands, a register may be required to provide data for either input of the Unit, or even both (think `add r1, r1`). Instead of having two central multiplexers associated with the two ALU input ports and consequently laying masses of wires (the register width's worth) towards each individual register from them, it is convenient and more efficient to have two buses permanently connected to the ALU ports and to let each register assert its content on either bus (or both) depending on control signals. This architecture will become clear in the next chapter; for now we simply add "output-enable" inputs, P1E and P2E, and two output ports to our design, justifying the term "multi-ported" in the section title.

---

<sup>3</sup>despite its being an array, the standard term is "register file"; it is one of those glorious misnomers of computing, alongside "move", "megabyte", etc.

## 11.12 Memory

Another class of sequential circuits is those Platform 0 components used for implementing computer memory. Logically there is no need to expand the set of basic elements to include silicon structures for supporting memory, since we already have latches that can play this role. In particular nothing is wrong *logically* in assembling billions of MSLs in one place and calling this computer memory. Technologically though, it is infeasible and the reason for it merits some explanation, as do the electronic principles and circuitry of modern memory chips. Those are not straightforward and we advise that you skip the next three sections in the first reading and proceed straight to section 11.12.4.

### 11.12.1 Platform 0 revisited. Capacitors as memory: the story of DRAM

Gone are the days of the early computing era when memory was fast and processors were slow and when speed was the only factor to consider. In the 1950s and 1960s it was common for random access memory (RAM) to be implemented using small ferrite rings (shaped like tiny doughnuts), each storing exactly one bit of data in the form of magnetisation. A ferrite ring (commonly known as a ‘core’ because it served as a core to a solenoid wound around it in other applications) kept its magnetic state indefinitely and needed no electric power to maintain it. So a computer’s memory maintained its contents even when the computer was switched off. A computer’s main memory (also referred to collectively as its *core*) was made up of a large number of ferrite rings ‘strung’ (by hand) onto a grid of criss-crossed wires. All that remains from that happy era is the word “core” in the unhappy phrase “core dump” meaning the dump (i.e. the snapshot image) of the whole system memory made after an unrecoverable crash and before a system re-start. Back in the early days, ALUs and registers were implemented using gates, which were macroscopic and used kilowatts of electricity at system level.

Modern systems (especially those in portable devices) are limited by energy consumption, by the amount of heat they generate, i.e. consumed power, and by size. The huge number of transistors required if we used Master-Slave Latches to implement a memory of typical size (4 to 16 GiB) would take up a great deal of space and would have a prohibitive power consumption. However, this would be the fastest implementation of all: MSLs are at least as fast as combinational logic, as they consist of the same gates/transistors and have an even shorter path between inputs and outputs.

A common design of a memory chip is presented<sup>4</sup> in figure 11.3 and it is a solution that sacrifices speed for large reductions in power consumption and size, and large increases in information capacity.

This example circuit is a tiny memory chip that contains only 16 1-bit memory cells, but it illustrates the principle well. Each cell contains a transistor connected to a grounded *capacitor*. A capacitor is a device that consists of two tiny metal blobs with a gap between them. It has the ability to accumulate electric charge, just like a mobile phone battery, but unlike the latter the charge is also tiny and due to leakages across the gap it lasts only tens of milliseconds without recharging even if not consumed by the rest of the circuit. Nevertheless, tens of milliseconds are an eternity in the computer world where the scale of time is measured in nanoseconds, which are millionth parts of a millisecond, so a capacitor will hold its charge for a considerable number of clock cycles. The transistor of the memory cell acts as a switch that connects the capacitor to the vertical “bit-wire”.

The memory chip operates on the per-row basis, with an operation in progress affecting a single row, each cell in it in the same way. Note that the transistor shown in the diagram is a real MOSFET transistor, not a LogiSim simulated variety. All MOSFETs are symmetrical: when the gate is open, the other two terminals act as if they were connected to a wire, and current may pass in either direction.

### 11.12.2 DRAM operation

**Reading from memory.** The row to read is selected using the signals *a0* and *a1*. A whole row is selected at a time and the gates of all transistors in that row are energised to open the transistors. The capacitors are connected to their corresponding vertical lines and then the MUX (4P2T) is enabled and switched to position 0. The bit-wires are now connected to the inputs of the *sense* amplifiers. The sense amplifiers have

<sup>4</sup> [https://en.wikipedia.org/wiki/Dynamic\\_random-access\\_memory#/media/File:Square\\_array\\_of\\_mosfet\\_cells\\_write.png](https://en.wikipedia.org/wiki/Dynamic_random-access_memory#/media/File:Square_array_of_mosfet_cells_write.png)

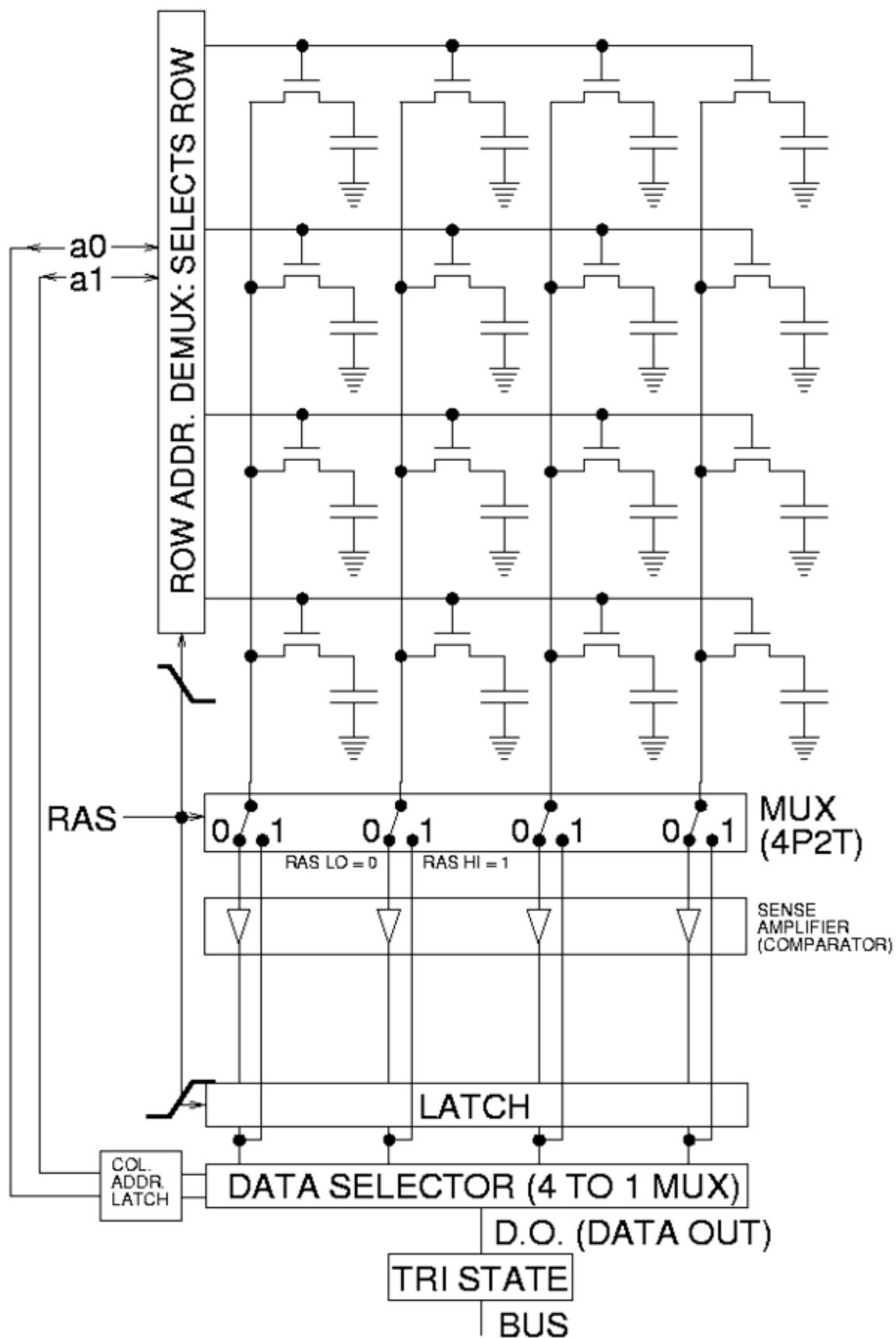


Figure 11.3: Square array of mosfet cells (by Glogger at English Wikipedia)

the ability to produce the digital signal 0 or 1 corresponding to the state of the capacitor in each column for the selected row; the capacitor itself does not have enough power to drive digital logic directly.

The output of the sense amplifier is latched in the horizontal LATCH (using a local timing trigger) and then the MUX (4P2T) is switched to position 1, so the latch output drives the capacitors in the row with the full voltage corresponding to 0 or 1.

If the capacitor's charge was degraded (1 not quite reaching to the power supply or 0 not quite down to the ground), it is now reinforced and the cell is fully charged or fully discharged. At which point the row is said to be *open*. The enable signal for the MUX(4P2T) is called RAS (Row Address Strobe). The row is held open as long as RAS is active. Then the MUX disconnects the vertical lines and the row "closes" and starts to slowly self-discharge. At which point another read cycle may start on a different row.

Which bit to select out of the freshly read row is decided by another MUX, the DATA SELECTOR based on the Column Address Latch (which is really a full address latch as it holds both the row and column addresses), and then the familiar controlled buffer arrangement is employed to connect the data output of the selector to the bus.

**Writing to memory** is very similar to reading. The only essential difference is that when the circuit needs to write into a single bit in an active row, the value to be written forces the corresponding bit's sense amplifier to ignore what is currently in the capacitor and assert the value that the input data demands. This can easily be achieved by combinational logic and will not be described here. Note: the value to be written comes from the bus but it is not shown in the diagram for the sake of readability.

It is important to understand that both reading and writing to memory causes a *complete recharge (to correct 0 or 1 levels as appropriate) of the whole row* that has been opened, no matter which bit of the row is used in the operation.

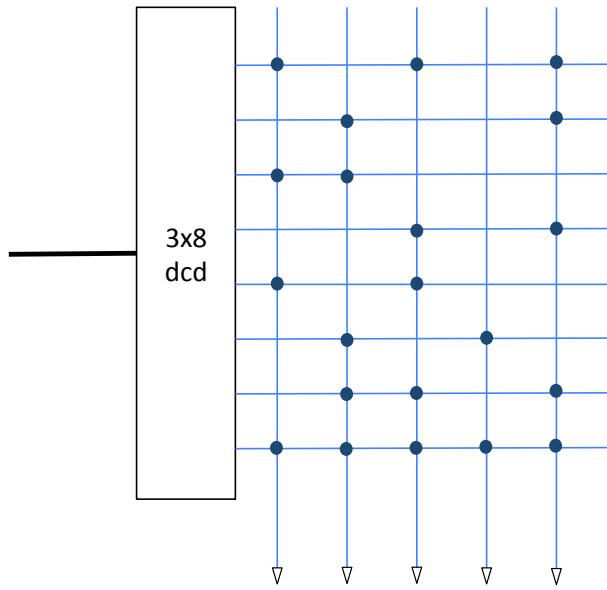
We have described, in a simplified manner but keeping to the essential details, the functioning of the so called Dynamic Random-Access Memory or DRAM, which you will find in most computers in the form of plug-in boards featuring a few DRAM chips. An instructive feature of the DRAM is the trade-off between the loss of speed and the tremendous increase in circuit density and information capacity and an equally impressive reduction in power consumption. Charging and discharging capacitors at the end of (relatively) long "bit-wires" is hundreds of times slower than logic gate operation, but the array of transistors and capacitors is far smaller, and capacitors take next to no electric charge to change state compared to a feedback-driven pair of logic gates.

However one major headache has been introduced besides the speed degradation, and that is the fact that a DRAM chip, if left alone, will lose all its data in a rather short space of wall-clock time (current industry-standard is about 64 milliseconds). So even if your computer is doing nothing, your memory has to read all of its cells roughly 16 times every second just to be "memory", i.e. to keep the data safely latched in.

This is typically achieved by using another clock (inside the memory chip), and some registers that keep the address of the row to be refreshed next. Bear in mind that refreshing a row is a read operation and that the chip, and the circuit shown in fig 11.3 can only do one read at a time. So if the chip is busy refreshing, the system cannot progress any of the programs that are being run at the time until the refresh cycle is over. This can be remedied to a certain extent by "smart" refresh, i.e. logic inside the DRAM chip that learns the access pattern to memory by the currently running program and refreshes those rows that are more likely not to be accessed soon.

### 11.12.3 Read-Only Memory (ROM)

Read-Only-Memory serves the same purpose as RAM except the content there is permanently latched in, requires no refresh and cannot be changed afterwards. Such memory chips usually hold programs and permanent data in embedded systems, such as controllers inside a microwave oven or a washing machine, or the so-called EFI memory in a PC, whose function is to hold permanently the software needed for booting the OS and for running diagnostics. Notice that a ROM is not a sequential circuit: it has no state and consequently requires no clock for timing state transitions.



Physically the ROM is much simpler than any form of RAM. The main part of it is a grid of wires, horizontal and vertical ones on the above diagram, where each intersection is connected with a *diodic* fuse. The fuse only lets the signal through in one direction: from the horizontal to the vertical wire, but not the other way around. Thus no two vertical wires are connected to *each other*, since such a connection would involve a signal travelling from a vertical to a horizontal wire and back to a vertical. Each fuse can be blown once in the process of placing content in the ROM, or left alone. The data words run horizontally and are selected by a decoder on the left.

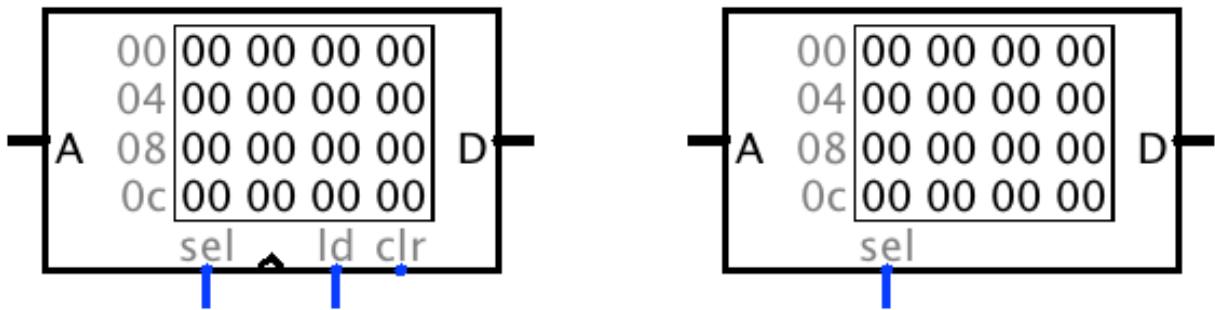
In the above figure the top row represents the content 0b10101, the next one down 0b01001, etc. The blank ROM chips are manufactured with all fuses intact, so each row looks like the bottom one in the diagram. The blowing of the fuses is done either in the process of manufacturing the device (the corresponding intersections are simply isolated inside) or by a ROM user (*a system* as opposed to component manufacturer) in which case the type of ROM used includes a special decoder with powerful output transistors that are able to sink sufficient current into the horizontal lines to destroy the connection. The user only needs to ground those vertical outputs they wish to blow the fuse for and the row of data is written in.

After the ROM has been filled with data (the term is “programmed”) it works as follows. When a row is selected the blown fuses stop the signal on the horizontal line from reaching the vertical one. The fuses left intact serve as ordinary electrical connections. As a result, when the decoder input is asserted with a memory address, the corresponding row is energised and the vertical lines have a binary code that corresponds to the stored content, 1 is read off as 1, and 0 as Z (floating). The latter can easily be converted to a proper 0 by connecting a pull-down resistor to each output, which is done inside the ROM chip.

#### 11.12.4 Memory as a Platform 1 building block

Fortunately, there is very little that we need to know about memory in order to use it in our simulated platforms without leaving anything important out. After all, as computer scientists we are interested mainly in the functionality of platform building blocks and only then in any engineering constraints that can potentially affect the logical view of the whole system. The former is essential, the latter is less so, and one needs to master the basics before taking a plunge in the technological quagmire of modern systems taking all the constraints, cost considerations and limitations on board.

For our purposes a memory is an array of sequential circuits. Those behave more or less like a register each, so there is very little difference between an array of registers and a memory chip, at least as far as their function is concerned in simple simulated systems. The figure below depicts LogiSim circuit elements representing two main kinds of memory: the RAM (we ignore the dynamic aspect of DRAMs, assuming that memory refresh logic, if required, is so efficient that the refresh process is completely invisible), and the ROM. The RAM is shown on the left.



It has two bunched terminals, A[address] and D[data]. Control signals include the clock (conventionally shown on chips in circuit diagrams as a tick symbol  $\wedge$ , the **sel** signal normally referred to as “chip select” which is a simple “enable” for the chip operation (and which allows several memory chips to be used on common address and data buses with a decoder choosing the active chip for a given operation), the **ld** signal which is quite misleadingly called “load” but means “memory read” when it is up and “memory write” when it is down and finally an unimportant **clr** terminal which can be used to reset the whole memory content to 0 at any time irrespective of the clock.

In order to read from memory all that is required is to bring the **ld** up and assert an address on the A terminal. The D terminal will be asserted by the content of the memory cell with that address as long as the latter is maintained on A.<sup>5</sup> The signal **sel** must be up for any memory operation, otherwise the D terminal is disconnected.

In simulation, it is only possible to write to memory at the rising edge of the clock. The **ld** signal must be down at that point in time telling the memory chip to start the write operation.

If we are to ensure the same clock discipline in memory as we have in MSLs, this behaviour is not suitable, since the new state must be locked in at the falling, rather than rising, edge of the clock. This is easily achieved by inverting the clock signal before feeding it to the clock input of the memory chip.

The working of the ROM chip is much simpler. The A terminal is sensed continuously as soon as **sel** is up and the D terminal is asserted with the content of the memory cell pointed to by A. For our purposes the only important thing to understand is that if the address is asserted at the beginning of the clock cycle, by the end of the cycle the ROM will definitely stabilise and we can latch the content of the memory cell in a register by using D as data on the falling edge of the clock.

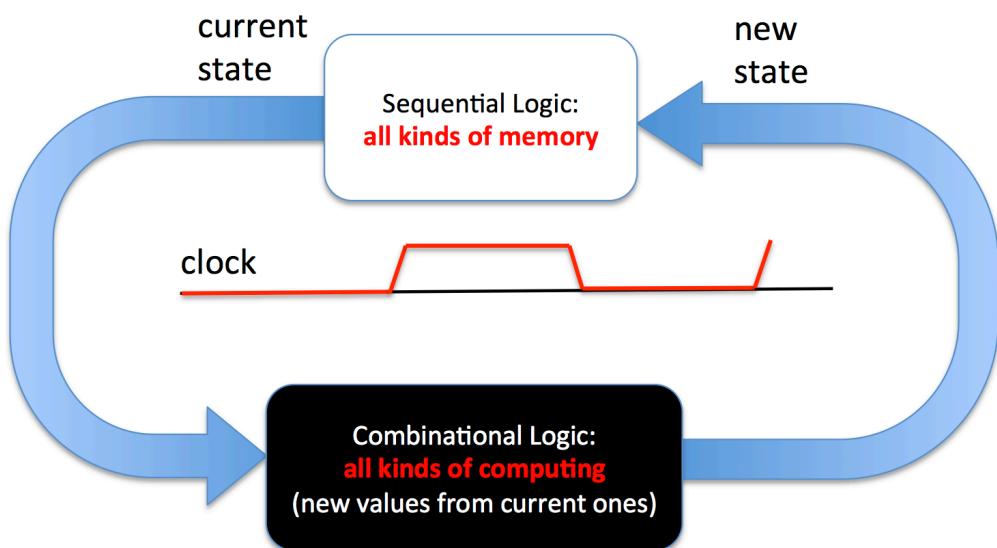
---

<sup>5</sup>LogiSim, which we use in this book, is not particularly accurate with the timing of large library circuits; in particular, it makes the memory delay unmeasurable, unlike the gate delay. However, since all of our designs are clock-synchronous, we do not care if some circuits are faster than others as long as the signals stabilise by the end of the clock cycle, which they do in LogiSim.

# Architecture of Platform 2

We are now fully prepared to explore platform *architecture*. By this we mean a specific arrangement of sequential and combinational logic units that make a circuit a Platform 2, i.e. a machine that runs bit-string programs. Although our ultimate goal is to build a CdM-8 Platform 2, this chapter begins with a general overview of platform architecture (which is not CdM-8-specific), and works its way down to a simple working machine (simpler than CdM-8). Only when the principles of the machine become clear do we proceed to a specific CdM-8 solution, which we need to provide a detailed illustration of the general principles we begin with.

## 12.1 The (high-altitude) eagle view



The above is a very abstract view of the main part of any computer. As we already mentioned in section 11.9, the central feature of a digital computer is a system-wide feedback: the data goes backwards and forward between the sequential and combinational parts of the design. Specifically, the sequential logic part is essentially memory which holds the current state of the computation. In order to advance it, some of the current state needs to be fed to the combinational logic part, which contains all sorts of computing devices, such as ALU, multiplexers, decoders, etc. The whole purpose of the combinational part is to compute a function of the current state and thus determine the next state. That next state needs to be latched in the sequential logic part, overwriting where necessary some of the current state. Once this is done, the machine is ready for the next step.

Notice that in the above we do not focus on what the current state consists of. Nor are we talking about how exactly the function is defined, but it is clear that its definition is contained in the current state itself — there is simply nothing else on the diagram that can be used to serve as a basis for a state transition<sup>1</sup>. Let us first consider the timing of those important events.

<sup>1</sup>for the time being, we are ignoring input/output, which is the interaction between the machine and its environment; we will dwell on it later

**The clock cycle.** At the heart of any computing platform is its principle rhythm driver, a single signal that ultimately defines when things happen around the circuitry. This signal is, naturally, the clock, which we have seen already. The platform requires two triggers in order to perform a single step. At a certain point in time, the step begins, and that needs to be marked by a signal transition. The signal transition is in other words an edge of the clock, we will assume it is the rising edge for certainty. Similar to every other physical mapping that possesses duality, it does not matter which edge the start of a clock period is marked with, rising or falling, but it has to be decided and adhered to. A selection of current-state data is then made, based on itself: depending on the data held in sequential logic devices, a certain part of the state (the content of a certain register, for example) is asserted on the connecting wires, which are shown in the diagram as the curved arrow on the left. There are a certain number of those wires, and the number determines how much data can be processed in the combinational logic (the black box at the bottom of the diagram) at once. Naturally, selecting the right part of the current state already requires a small amount of combinational logic (a multiplexor), which is typically co-located with the sequential devices. The black box at the bottom only sees the relevant part of the state.

As soon as the clock goes up, the connecting wires deliver a valid and stable data to the black box. Those data items contain the information about what function to perform on some part of the lot in order to produce a useful result. The output of the black box is functionally dependent on the input, except there is some delay in following it. So initially the values on the output wires of the black box will not change, but after a while (and it is important that this *always* happens before the *falling* edge of the clock) the output will stabilise and represent the valid result. The white box at the top must continue to maintain its output throughout the process. At the falling edge of the clock the handover takes place: the white box latches (updates) part of its state with the new values, while *still* maintaining the current output until the latching is finished. Which part of the current state gets updated depends, as before, on the current state itself, and there will be combinational logic (decoder or equivalent) embedded in the white box to make that selection. Now the clock is down and the new state is fully latched in. The white box must start asserting the new values on its output and it will make them valid and stable by the time the clock rises up again.

In this arrangement the white and black boxes take turns governed by successive clock transitions: computation, state-update, computation again, state-update again, etc. Notice that there is no logical reason why it should take the same amount of time for either box to do its job. So the low-clock period could, for instance, be made shorter than the high-clock one or vice versa. What do those times depend on? For the black box, the time depends on the complexity of the ALU. There are two kinds of complexity: the number of transistors, which affects the power consumption and the thermal regime of the circuit, and the number of layers through which the signals must propagate, and it is the latter that interests us. To get the idea of what is involved, imagine a multiplier unit that does multiplication by summing up partial products, as we discussed in section 3.8. All the partial products can be produced at the same time by circuits operating in parallel. The totalling up of the partial products breaks down into stages: for 8 products we need to add up pairs of them resulting in 4 sums, then those are added together pairwise to produce two results and finally the two results are added together. Accordingly, the signals propagate through several adders before they reach the output. Each adder introduces a delay, and the delays build up to contribute to the overall delay of the black box and consequently the minimum duration of the high clock period.

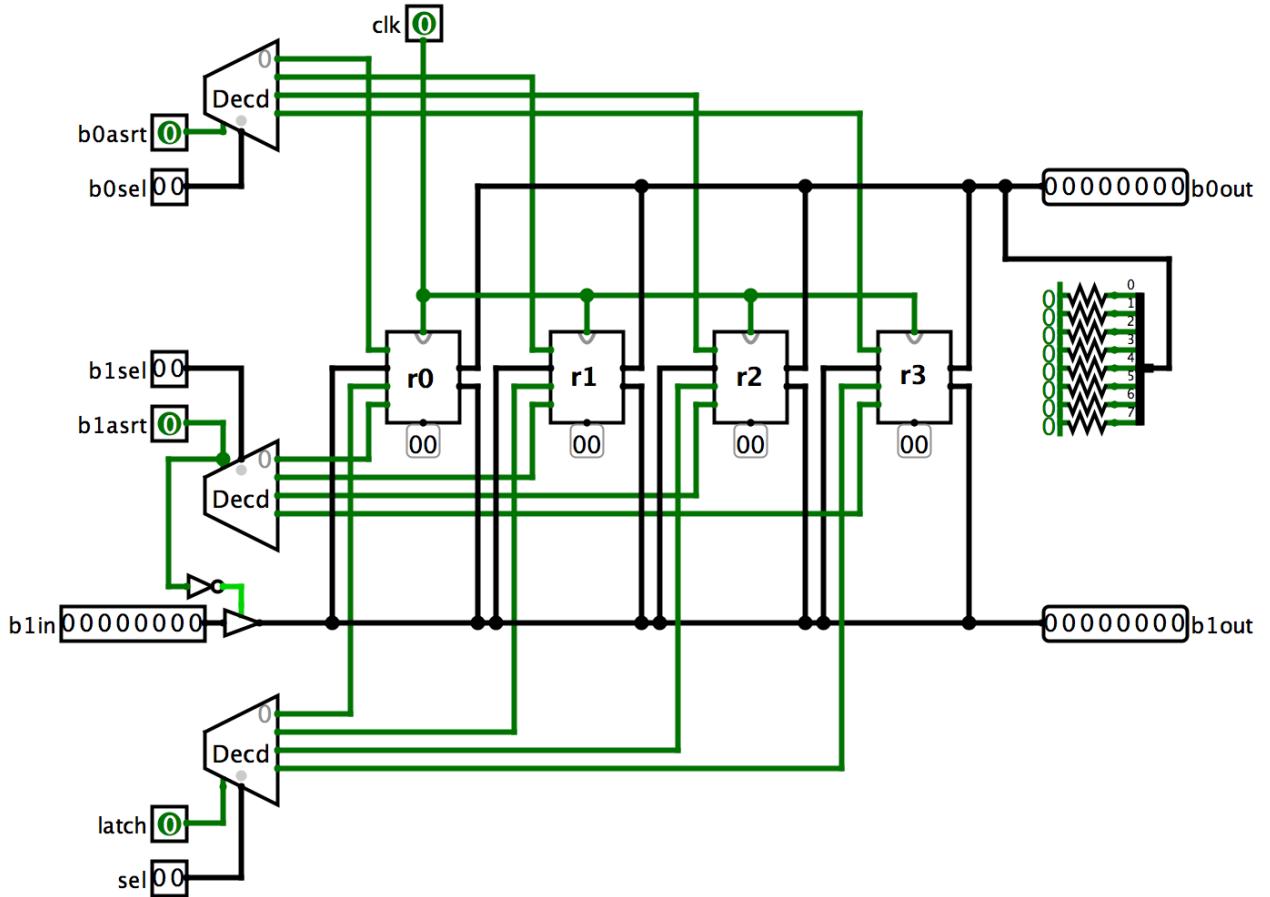
The low clock period must last long enough to allow the latching of the result to be completed and the fetching of (a part of) the new state to stabilise *before* the clock rises. In fact the latter requirement is not strict, since if the high clock period is sufficiently long, even if some data is not stable at the beginning of it, it may become stable later leaving enough time for the ALU, etc., to finish signal propagation. So in fact the sum of the store-load time and ALU propagation time should be less than the duration of the full clock cycle.

If all sequential logic were registers and other forms of latches, the store-load delay would be extremely small compared to the “heavy” ALU units: a couple of transistor delay’s worth or thereabout. Unfortunately, an important part of any hardware platform is proper memory: perhaps a dynamic RAM, which is large enough to support Level 2 programming and which cannot realistically be implemented using latches, see section 11.12. The physical processes in the RAM (charging or discharging capacitors in order to latch new data) are typically much slower than the process of charging up a transistor gate in a latch, because the capacitors’s charge must last tens of milliseconds, i.e. millions of clock cycles. Due to these considerations, the clock tends to be non-square, either the high or the low clock period is longer than the other.

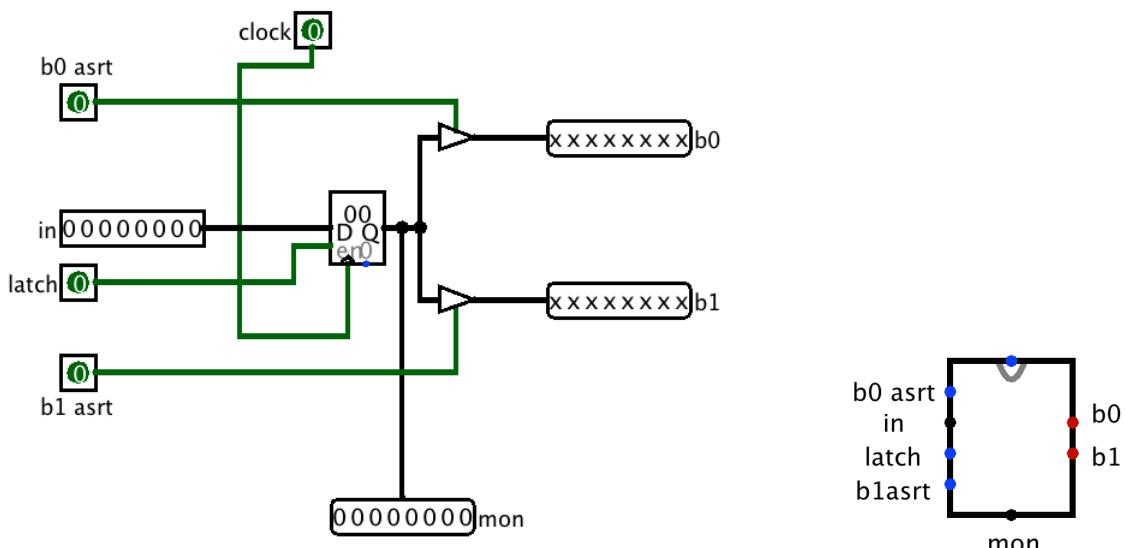
However, in this book we concern ourselves with the physical side of platforms as little as possible; all we care about is the sequence of events and how any data gets computed. Accordingly, our next question is about the structure of the white box. We explore it in the following section.

## 12.2 The register file

The main part of the white box, which is used in nearly every clock cycle is called the *register file*. As mentioned earlier, it is an array (rather than a file) of registers all multiplexed onto at least two buses and indexed by a common set of external inputs. Here is a realistic enough example.



Let us take a closer look. The core unit here is an individual register, of which the file needs four:



The above circuit diagram uses a standard component “register” available from Logisim. It has the data “D” input, which is used for supplying data to the internal MSL on the falling edge of the clock, the enable signal

“en” input which is required to be up for the latching to take place, and the “Q” output on which the internal MSL array asserts its content at all times. The circuit contains two multi-bit controlled buffers, which are controlled by the signals **b0asrt** and **b1asrt** and which pass the Q value to either output **b0** or **b1**. Finally the value of Q is also asserted on the monitoring output of the circuit so that we may observe what data is stored inside by connecting a simulator probe to it. The pinout diagram on the right helps to read the pins of the register chip “in context”, when looking at the register file. Notice that the clock input is marked with a little triangle: this is the standard mark of a clock input in electronics.

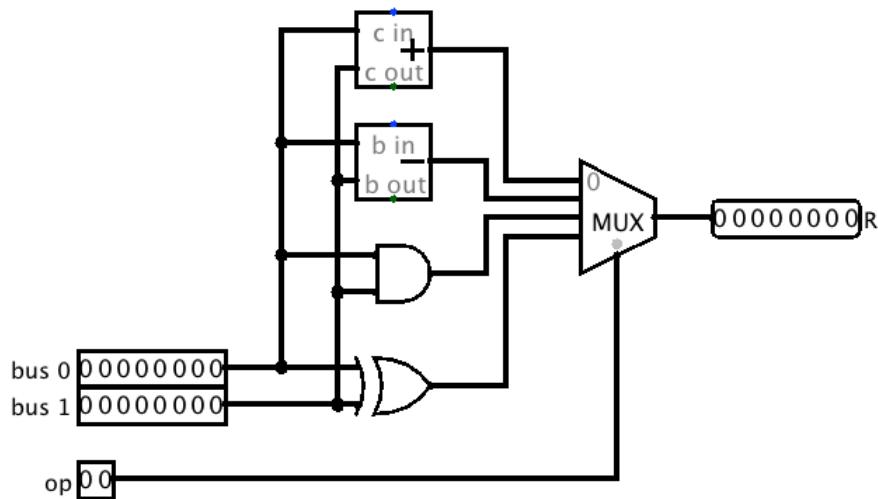
The register file is built around two buses, which we call **b0** and **b1**. The upper bus is asserted on by the register whose **b0asrt** input is high. No more than one such register will be found at any given time, thanks to the upper decoder, which supplies the enable signals to all four registers. If the decoder is not enabled, i.e. if none of the registers asserts data on the upper bus, the resistors pull it down to zero<sup>2</sup>. Which register is asserted on the upper bus is determined by the decoder’s main input: **b0sel**.

Each register can also be asserted on the lower bus, **b1**, which is controlled by the pair **b1asrt** and **b1sel** in exactly the same manner. As well as reading from, the register file supports writing into its registers, i.e. latching input data in them. This is achieved by using the pair of input pins on the register, **in** and **latch**. The former is wired to the lower bus, which means that **b1** is the only source of data for latching in a general-purpose register. Which of the registers does the latching (and there can be at most one) is controlled by the bottom decoder in the manner that we are already familiar with. Notice that the register file provides an external input to the lower bus, but in order for that not to conflict with the bus assertion coming from a register, the “in” input of the file is connected to the lower bus via a controlled buffer coordinated with the middle decoder by an inverter.

To summarise, a register file is a unit that has an array of registers inside and some ancillary circuitry required to utilise it. The file is able to latch input data asserted by an external driver on the lower bus, and/or to assert data on either or both buses.

The register file is the main sequential circuit of a processor. Let us now proceed to its largest combinational circuit: the ALU.

### 12.3 The ALU



The ALU is a direct extension of a single arithmetic device such as the adder. For our preliminary excursion into platform architecture we choose an ALU composed exclusively from ready-made parts available from our logic simulator: an adder, subtractor, an AND array and a XOR array. For now we ignore the condition flags, such as CVZN. Also ignored are shifts, bit sliced operations and other bit-wise logic. In fact this is not an ALU in any real sense, it is a placeholder for one, which can only do a few very basic things. The important *structural* aspect here is that this circuit has all four units working at the same time based on the same pair of 8-bit inputs. The names of the input pins, **bus0** and **bus1** betray the intention to connect this

<sup>2</sup>Just to be consistent, we continue with the wire-OR technique here, but in this case a multiplexer would suit us even better.

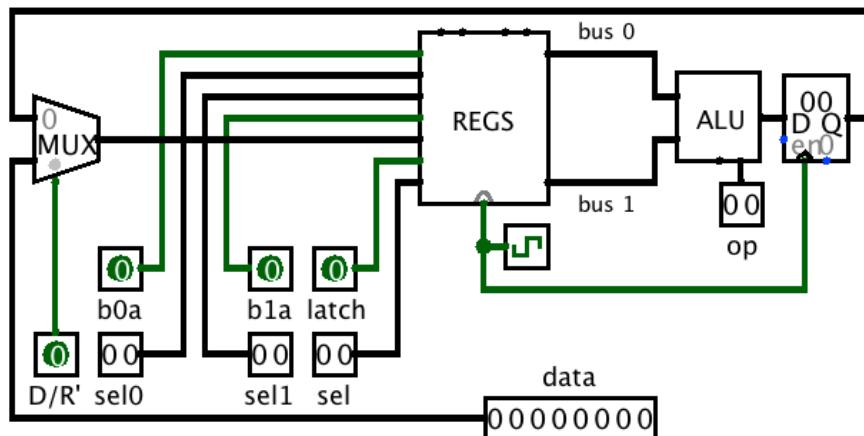
unit to the register file. Indeed the latter is the only source of data in the sequential part of the processor so far. There is also a 2-bit “op” input in our example, which encodes the operation that the ALU is supposed to perform. The result is taken off the correct unit by the mux on the east side; the rest of the units’ results are ignored.

This is the simplest possible architecture: it is incomplete and it does not compute condition flags. It is oblivious to the power issues as well: those combinational units (at least three in any clock cycle) that are not involved in the current operation still see signal transitions on their inputs, which they translate into transitions on the (ignored) outputs; such transitions waste power. In a real design, large arithmetic units’ inputs would be inhibited when not used.

However, we need an ALU to demonstrate the workings of the system, and an ALU it is. Here is the interpretation of the “op” input following directly from the circuit:

code	interpretation
0b00	add
0b01	subtract
0b10	AND
0b11	XOR

## 12.4 Example Data Path and Register Transfer Language



### 12.4.1 Data path

The above is a complete compute engine, i.e. a circuit that can perform a series of computations, presented in a single circuit diagram; the conventional term for such a circuit is the *data path*. This data path is able to perform arbitrary (within the capability of our ALU) computations on the four registers contained in the register file. Because it is only an engine, its controls must be driven by an external agent, for example we could do it by asserting values on input pins by hand<sup>3</sup>. Consequently, one will not be able to load and run a *program* here either. Even the clock, shown as a box with a waveform inside it in the centre of the diagram, needs to be advanced by hand (by clicking on it in the simulator). Nevertheless, we shall stay here a little longer to discuss what signals the data path needs and what effect those have on various kinds of data.

First of all, let us take a closer look at the design. The two buses of the register file are connected with the ALU inputs, and the ALU output is fed to the result register triggered also by the falling edge of the clock. The output “Q” of the register thus contains the ALU result from the previous cycle. The ALU does produce the result in *every* clock cycle, whether it is needed or not, since it is a combinational circuit. On the west side, the result register output comes to the 0th input of the mux, whose first input comes from an 8-bit constant “data”. The purpose of the mux is to deliver either the ALU result or the *immediate* data to the input of the register file for storing it in one of the registers.

<sup>3</sup>here and below we do not distinguish between real electronics and Logisim simulations; it is acceptable for our purposes to assume that those are the same thing.

The whole arrangement requires up to 9 input signals in every clock cycle. The actual number of signals required varies because some of them are ignored when others carry certain values. For example, here is how the CdM-8 Platform 3 instruction `ldi r3,7` can be executed on the data path:

clock cycle	D/R'	data	b0a	sel0	b1a	sel1	latch	sel	op
1	1	7			0		1	3	

The above table shows the value of input signals of the datapath in the first clock cycle. Those signals that do not matter (since they do not affect the computation) are not shown. The multiplexer control input is up, hence the bits from the arrayed pin `data` (i.e., the bit string representing 7) are asserted on the lower bus. The `latch` signal being up as well, the selector `sel` chooses the third register `sel=3`. The whole ALU part of the engine clearly does not matter: which register if any is asserted on the upper bus and which operation is selected via the “`op`” input of the ALU — none of it makes any difference, since the result of the ALU is ignored by the mux.

Let us do a bit more: let us drive the engine to compute the following.

```
ldi r3,7
ldi r2,4
sub r3,r2
```

And here is a solution in 4 clock cycles:

clock cycle	D/R'	data	b0a	sel0	b1a	sel1	latch	sel	op
1	1	7					1	3	
2	1	4					1	2	
3			1	3	1	2	0		1
4	0						1	2	

One can easily imagine a larger computation put into effect by signalling on the above 9 inputs over a number of cycles. Two observations are of note here. First of all, something has been missing in the picture so far: a unit that produces the control signals (which we have produced by hand) based on a *program* stored in memory, i.e. a *control unit*. We will see how this can be built a bit later. Secondly, even for the tiny example shown above it is already clear that the table content is rather sparse, which makes a table an inconvenient representation of the actions taken by the engine. This brings us to our next point,

### 12.4.2 Register Transfer Language

First a few more observations. The meaning of the signals displayed above can be summarised as being one of the following two: a selector, controlling a mux somewhere in the circuitry, or a trigger, causing values to be latched in a register or asserted on a bus. Some selectors are combined with a trigger, for example `sel1` only matters when `latch` is high, and some are not, e.g. `op`. Those that do are usually indices of some file; the aforementioned `sel` is in fact the index that selects one register out of four. Let us call the former category *functional*. In this design `D/R'`, `op` and `data` are functional selectors, while `sel,sel0,sel1` are indices.

Consider the following *symbolic* representation of a row of the signal table. For every functional selector, introduce symbolic names that represent the meaning of each of its possible values, except when the value is an abstract number (example: `data`), in which case we simply write that number. For the `op` signal we use the mnemonics summarised in the table on page 240: `add`, `subtract`, `AND` and `XOR`. For the `D/R'` signal, we use `result` and `immediate` for 0 and 1, respectively.

Each of the triggers causes a register transfer (hence the name of the language), i.e. the transfer of data between two registers or a register and a bus. Instead of specifying which trigger is raised, we write this using the arrow  $\rightarrow$  constructor as follows:  $x \rightarrow y$ , where  $x$  and  $y$  are the (possibly indexed) source and destination of the data. Since there is only one trigger in the data path that causes the transfer between any valid source and destination, the arrow notation contains all the necessary information, but it is much more readable than the original row of numbers. For example, the instruction `move r2,r3` can be written in Register Transfer Language (RTL for short) as follows:

$r[2] \rightarrow bus_1$ ,  $bus_1 \rightarrow r[3]$

and interpreted as the shorthand for

clock cycle	D/R'	data	b0a	sel0	b1a	sel1	latch	sel	op
1						1	2	1	3

Indeed for the first transfer to take place  $b1a$  has to be up, and  $sel1$  should correspond to  $r2$ . For the  $r2$  value, asserted on  $bus_1$ , to be latched in  $r3$ , the  $latch$  signal has to be up and the selector  $sel$  must point to  $r3$ .

The reader will have noticed that the two actions are written on the same line of RTL separated by a comma and may be wondering whether the mutual ordering of the arrows is important as well as the fact that they share one line. The answer to this lies in the architecture of the data path. The engine is triggered by the clock; all register transfers in it during a clock cycle take place at either edge of the clock (asserting after the rising edge and latching at the falling one). Therefore it does not matter which of the transfers we write first. It does matter, however, in which clock cycle they happen. That is why the convention in RTL is that each line represents a separate clock cycle, the cycles go top down, just like assembler code is executed top down, so the first line of RTL is cycle 1, then cycle 2, etc.

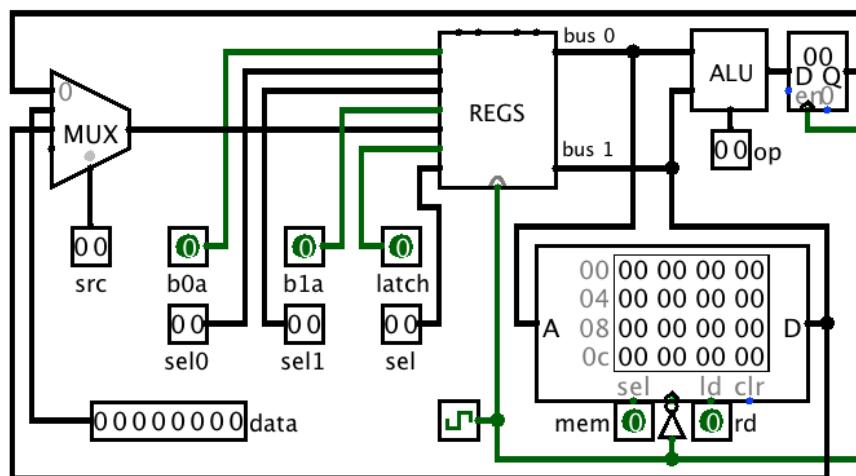
Here is the example from page 241 written in RTL:

```
immediate, 7, bus1-> r[3]          # ldi r3,7
immediate, 4, bus1-> r[2]          # ldi r2,4
r[3]->bus0, r[2]->bus1, subtract  # sub r3,r2      1st cycle
result, bus1->r2                  # "           2nd cycle
```

This already looks much like a program, albeit a very low level one!

### 12.4.3 Adding memory

Let us expand our example data path by adding a memory unit to it.



For simplicity, this is a single-cycle, static memory chip. In simulation it is clocked by the rising edge (no option to change that) hence the inverter in the clock path. This is how we connect the memory unit to our current design. The set of control signals has grown to 11 in total as we now have the trigger  $mem$ , which indicates that the memory is enabled in the current clock cycle, and a 1-bit selector  $rd$  which, if up, indicates that memory is supplying data in the current cycle (i.e., memory is being **read**), otherwise that it is recording the data supplied on its  $D$  terminal by the data path. Just as the ALU, the memory unit is connected to the two buses via its two terminals. Unlike the ALU,  $D$ -terminal is genuinely input/output, so it can be used in both directions, which is why it is connected to both the input and the output of the register file (on  $bus_1$ ). Since the value of the  $rd$  selector is only material when the trigger  $mem$  is up, all we

need to add to our RTL is that selector, giving it two symbolic values, `load` and `store`. Notice that the D/R' selector is now replaced by a 2-bit selector `src` with the values 0 and 1 denoted as before, and the new value 2 named `memory`. In doing so, we have committed to a *design constraint*, namely that whenever `load` is in progress, the value of `src` must be `memory`, since otherwise there would be a conflict between the input and the output of the register file on bus 1. In reality this means we have given up the opportunity to feed memory data to the ALU directly while using the register file at the same time for copying or latching some other values.

Let us illustrate on this simple arrangement of units how memory instructions may be executed. We wish to store the value 7 at memory location 3:

```
immediate, 7, bus1-> r[3]          # ldi r3,7
immediate, 4, bus1-> r[2]          # ldi r2,4
r[2]->bus0, r[3]->bus1, store    # st r2,r3
```

This corresponds to the signal table below:

clock cycle	src	data	b0a	sel0	b1a	sel1	latch	sel	op	mem	rd
1	1	7					1	3			
2	1	4					1	2			
3			1	2	1	3	0		1	1	0

A slightly more complex example: increment the content of the memory cell with the address 0xa3:

```
immediate, 0xa3, bus1-> r[0]          # ldi r0,0xa3
memory,   r[0]->bus0, bus1-> r[1], load  # ld r0,r1
immediate, 1, bus1->r[2]              # ldi r2,1
r[1]->bus0, r[2]->bus1, add           # add r1,r2      1st cycle
result,   bus1->r[2]                  #                   2ns cycle
r[0]->bus0, r[2]->bus1, store         # st r0,r2
```

This results in the following signalling

clock cycle	src	data	b0a	sel0	b1a	sel1	latch	sel	op	mem	rd
1	1	0xa3					1	0		0	
2	2		1	0			1	1		1	1
3	1	1					1	2		0	
4			1	1	1	2	0		0	0	

An interesting observation: even without the (8-bit) `data` signal, the number of wires (and bits in a bit string that defines the signals that they carry) is pretty large: 15 bits, which includes five 2-bit selectors and five triggers. Despite that, the amount of *information* that the 15-bit string contains is much less: not all combinations are meaningful, which is evidenced by the sparsity of the above signal tables. For example, the selector bits are only important if the selection is triggered by the corresponding trigger; when both buses are used for asserting registers on them, the latch logic is not used nor, accordingly, any latch and mux selectors there, etc.

**Instruction Decoder.** The redundancy of the signal bit-string is hardly surprising: we have seen earlier that the whole variety of CdM-8 Platform 2 instructions<sup>4</sup> only requires 8 bits — and here we have more than twice the number. It is also clear that the signal bit-string that corresponds to such an instruction is *functionally* dependent on it. The process of expanding a bit string into a bigger one containing the same information is called decoding. We have seen some primitive decoders before, those are *components* of the Universal Platform 1 toolkit. What is needed here is an *instruction* decoder, i.e. a combinational *circuit*, which turns an instruction bit-string into a control signal bit-string, functionally dependent on it and on the clock cycle number.

There are a couple of problems we need to solve before we can build an instruction decoder. A CdM-8 Platform 2 instruction generally requires more than one clock cycle, for example the instruction `add`, which

<sup>4</sup>with the exception of the ones that require the immediate operand, but then we did not include the `data` in the total of signal wires either.

is implemented in the same manner as the one above. Accordingly, the decoder should be able to produce more than one control bit-string out of a single instruction. Secondly, we require a circuit that will actively fetch machine instructions from memory, decode them and control the data path based on them in one or more clock cycles, then fetch the next instruction. We do in fact need a whole “instruction machine” inside a processor, just like a living cell needs its power plant, the mitochondrion, to drive the cell’s chemistry. That instruction machine has four distinct parts: a register set comprising three dedicated registers: the Instruction Register (IR), the Program Counter (PC), and the Processor Status (PS); the Sequencer; the Primary Decoder, including the Branch Logic; and the Secondary Decoder.

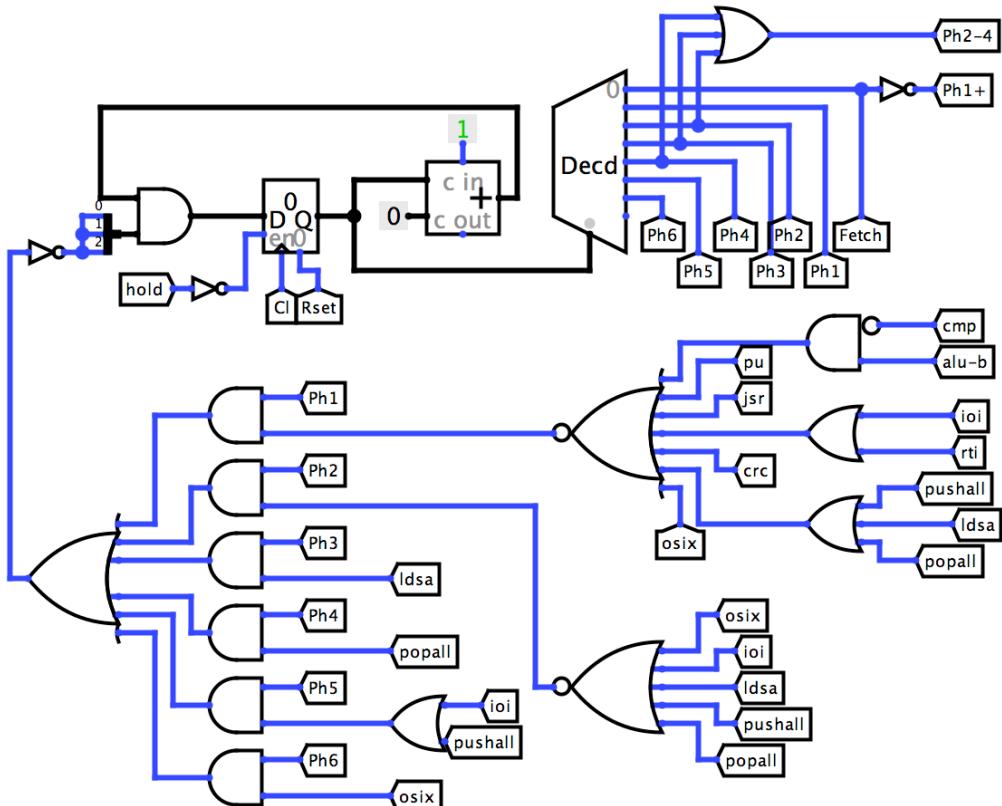
## 12.5 Instruction machine

### 12.5.1 The Sequencer

is the driver of the instruction machine. It is a circuit which produces the machine’s primary triggers in a cyclical manner, one trigger per clock period:

fetch, phase1, phase2, ... phasen, fetch, phase1,...

where  $n$  is the maximum number of cycles a Level-2 machine instruction requires to execute. The rest of the instruction machine may request the sequencer to reset its cycle, i.e. to start with **fetch** in the next clock period; this allows the instruction set to have instructions of different execution times. Here is the CdM-8 Platform 2 Sequencer:



Let us see how it works. The sequencer is based on a three-bit register triggered by the falling edge of the clock. This means that the east side of the register holds its value throughout the clock cycle allowing the feedback network (which includes a 3-bit incrementer realised as an adder with a wired carry-in of 1) to deliver the new value to the register input, which is latched in at the end of the high-clock period. The effect of this is that the register counts up modulo 8: 0,1,2...7,0,1... unless the input of the westmost NOT-gate is high. That input comes from the counter-reset network laid out below. The reset signal fed to the NOT-gate is raised according to the following:

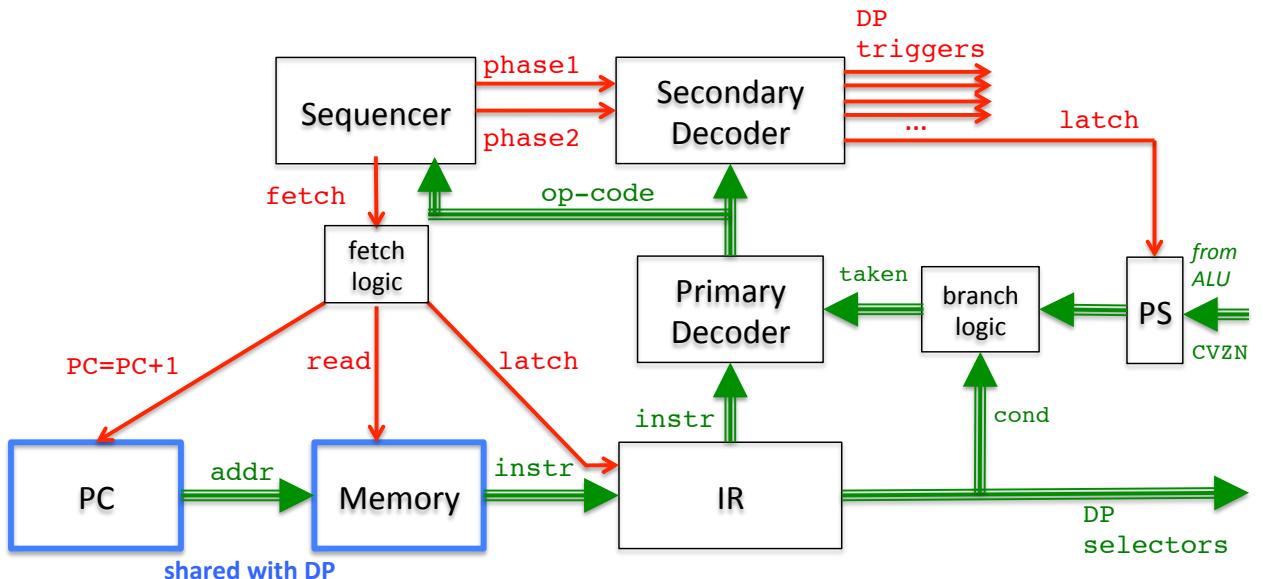


Figure 12.1: Instruction Machine. Red thin arrows stand for triggers, thick green ones are selectors and other data. Thick blue boxes are shared with units outside the instruction machine.

- All instructions are single-phase, and consequently reset the Sequencer at the end of Phase 1, except: any alu-b (but cmp is still single phase), any of ioi, rti and osix, any of jsr, push and crc (notice that pop is still single phase), and any of pushall, popall and lds. Those are the only exceptions.
- The rest of the instructions are two-phase, and consequently they reset the Sequencer at the end of Phase 2, except: osix, ioi, pushall, popall and lds
- only the above five instructions require more than two phases. Specifically lds needs three (cf. lda, which requires only one), popall four, pushall and ioi five, and finally osix six. At the end of Phase six the Sequencer will definitely have been reset no matter what instruction.

The output of the Sequencer is a collection of phase signals (Ph1, Ph2, ..., Ph6) tunneled to the Secondary decoder and the Fetch signal for the instruction-fetch logic<sup>5</sup>. The input of the Sequencer is the op-code signals tunneled to it from the Primary Decoder. The Secondary Decoder produces all the triggers needed for the datapath in any given non-Fetch clock cycle.

The structural view of the instruction machine is given in figure 12.1.

**Registers and memory.** As shown in the figure, three registers are used in the instruction machine outside the decoder. The IR has no purpose outside the instruction machine, there is no direct access to it in the program and the data path does not need its content for data processing purposes. However, the instruction held in it contains various selectors needed by the data path, such as register indices, ALU operation etc. Those are tunneled to the corresponding selector inputs of the data path. The PC is not limited to the instruction machine despite the fact that it belongs there. The register can be used for fetching immediate data that follow a machine instruction, it can be pushed onto the stack for subroutine calls, or an instruction may need to write into it (branches and other control instructions). The memory used by the instruction machine can be the same memory that the data path uses to read and write data; this is called a von Neumann architecture, which we mentioned before. Alternatively, instruction memory could be separate from data memory; the former does not even need to be a RAM, since a ROM can hold a program, and it does so in many small systems. Such an architecture is called *Harvard* and will be discussed later. Finally there is the PS register which is used to hold the flags (as well as for other purposes, which are not important at this point). The instruction machine only reads it, the data path only writes into it.

<sup>5</sup>The instruction-fetch triggers are produced by a part of the Secondary Decoder from the Fetch signal in the same manner as other triggers are produced from the Primary Decoder output, but their purpose has nothing to do with the Data Path, which is why the fetch logic is shown in the diagram as a separate unit. Also note the hold signal which is normally grounded allowing the sequencer to run through the sequence, but which can be raised by the wait/halt logic discussed later on.

**PC.** When the Sequencer produces the **Fetch** trigger, the fetch logic causes a memory read operation using the current content of the PC as the address. At the end of the high clock period, the PC (which is a falling edge triggered register like the rest of them) will latch the value of PC+1, as it should, thus ensuring the standard instruction fetch discipline already familiar from Platform 3. That incrementation is done by the Program Counter itself without using the data path with its ALU (hence the name “counter”). The **Fetch** trigger also triggers the IR to latch the new instruction at the end of the fetch cycle. The instruction sits in the IR until the next fetch cycle and is used as source data for Primary and Secondary Decoders, the Sequencer and the data path itself.

**The Primary Decoder** is a mere combinational circuit! All further triggers will be derived from the Phase triggers coming to the Sequencer, but that is the function of the Secondary Decoder. The Primary one has a narrow purpose of identifying the instruction op-code, or part of the opcode corresponding to the instruction class, in which all instructions have the same behaviour with respect to triggers. Such is, for example, the two-operand ALU class, where the op-code contains one bit that identifies the class unambiguously (bit 7 in this case), and the rest of the op-code (bits 6-4 in this case) encode the ALU operation thus forming a functional selector available directly from the IR.

The Primary Decoder identifies each instruction or instruction class and the corresponding output signal is raised. There are, consequently, as many output signals leaving the Primary decoder as there are different instructions/instruction classes. Some of these signals are used by the Sequencer to cut its sequence short, but all of them without exception go to the Secondary Decoder.

There is one special case of primary decoding that falls half-way between instruction and data parts of the platform. The instruction machine has its own logical engine, the *branch logic*, which is used in combination with the Primary Decoder. The primary decoder recognises a branch instruction op-code by its prefix (like every other machine instruction). The branch op-code contains a 4-bit field, which we call the *condition code* and denote as *cccc*. It encodes the condition on which the branch is taken. The branch logic is a combinational circuit, which reads this field from the Instruction Register as one of its inputs, and the CVZN flags from the PS register as another. It then determines whether or not the combination of the condition code and the flags is such that the branch is taken. This one-bit decision is wired to the Primary Decoder.

If the current instruction is indeed a branch (which the PD determines immediately), then the output of the branch logic further classifies that instruction as a branch that is taken, or a branch that is not taken, and the corresponding output of the PD is raised. The Secondary Decoder then deals with the specific class. Notice that there are 16 different branches in CdM-8 Platform 2 instruction set, but that they are decoded/recognised *in the context of the current value of the flags* as being the above two types. Indeed, as far as the effect of the branch is concerned (the new value of the PC), taken or not is all that matters.

**The Secondary Decoder** receives two phase triggers from the Sequencer and the decoded op-code. It is also a purely combinational circuit, but now the intention is to work out which triggers need to be raised for a given op-code *and phase*. The Secondary decoder is simply a set of decoding sections, with each section being a collection of buffers driven by the AND of a phase trigger and a decoded op-code. For example, the AND of the *pu* signal (which is the op-code of the CdM-8 Platform 3 push instruction decoded by the Primary Decoder) and the *Ph2* trigger causes the rise of three triggers: *RdAssrt*, which tells a GP register to assert its content on bus 1, *SP2b0*, which tells the stack pointer SP to assert its content on bus 0, and *mem*, which triggers a memory write cycle. Notice that which register gets asserted on bus 1 is defined by a selector contained in a fixed place of the Platform 2 instruction and wired directly to the register file.

We will discuss the implementation of both decoders next.

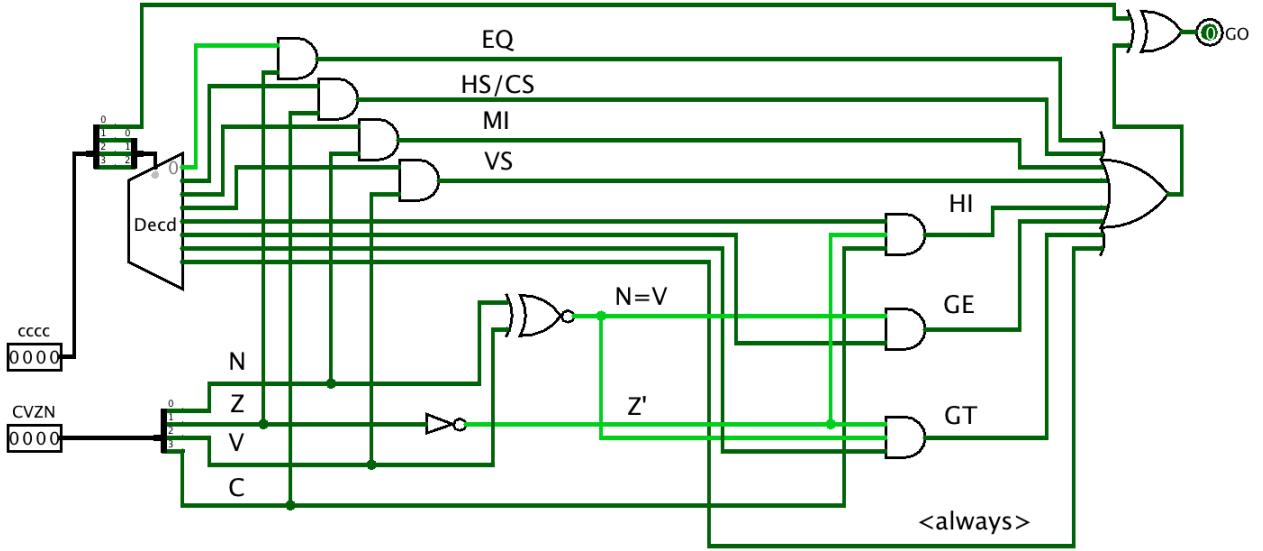
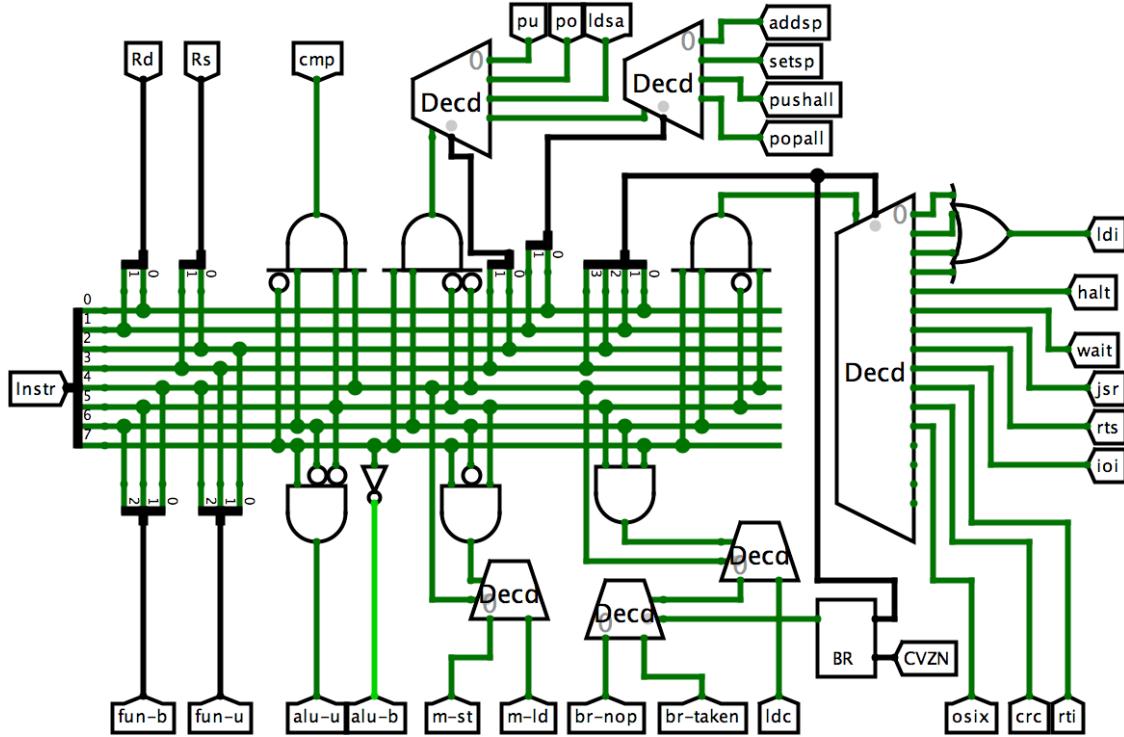


Figure 12.2: CdM-8 branch logic

### 12.5.2 Primary decoder (PD)



The specific task of the PD is to decode the op-code of the instruction held in the IR. The word “decode” here is taken in its standard sense, i.e. to raise a wire associated with a specific instruction op-code or instruction class whenever the IR contain that type of instruction. This is similar to the Universal Platform 1 decoder unit raising the wire that corresponds to the binary number contained in the multibit input, only in the case of a processor’s primary decoder the decoding is a little trickier, since the instruction identification is based on more than one condition: in CdM-8 Platform 2 the op-code does not have a fixed place in the 8-bit instruction bit-string. (We refer the reader to section 8.2 for details.)

The circuit is fed the full content of the IR via a tunnel named **Instr** on the west side. The data is asserted onto the PD’s central bus, which drives all recogniser logic around it. Additionally, tapped off the bus are the CdM-8 data path’s two pairs of principal selectors (one index and one functional), which are :

Tunnel	Selector Description
Rd	index of the destination register, e.g. 3 for add r1,r3
Rs	index of the source register, e.g. 1 for add r1,r3
fun-b	ALU op for a 2-operand instruction (e.g. add)
fun-u	ALU op for a 1-operand instruction (e.g. inc)

Now let us briefly go through the decoding part. Each instruction type is decoded using the same type of structure. Signals off the bus meet an AND-gate with suitable input inverters which recognises the (prefix of the) op-code<sup>6</sup>. Then either this is the answer straight away, or it is a signal that enables a Platform 1 decoder component, which decodes some further bits into the actual instruction op-codes represented as signal wires.

For example, a low bit-7 is immediately identified as `alu-b`, which is a two-operand ALU operation. Whether the instruction is an `add`, `sub`, `and`, etc., is immediately defined by the selector `fun-b` which is tunnelled to the ALU, while `alu-b` carries on to the Secondary Decoder to be combined with the phase signals to produce some triggers for the data path. The selectors `Rs` and `Rd` are tunnelled to the register file to ensure the correct registers are used to assert operands on the ALU inputs and to latch the ALU result.

As mentioned earlier, the PD utilises branch logic, which is included in the diagram as a chip marked BR. Its implementation is shown in figure 12.2. The inputs of the branch logic are the branch condition code which is tapped off the instruction bus (see north-east corner) and the CVZN flags tunnelled to it from the PS register. There is a one-bit output which is decoded as either `br-taken` or `br-nop`, when the logic determines that the branch should be taken or ignored, respectively. Naturally, the output is only meaningful when the current instruction is a branch, which is determined by the PD independently.

Perhaps the most complex decoding is done for a class of instructions that include `ldi` and a collection of 0-operand ones<sup>7</sup>, such as `halt`, `jsr`, `rts` etc. The whole class requires a prefix 0b1101 in the most significant bits, which is recognised by an AND-gate. The remaining four bits represent either an `ldi` with its single register operand (codes 0b0000 to 0b0011, four patterns in total, since there are four choices of the register operand), or a 0-operand instruction, of which there are up to 12. Not all 12 are present in the current version of the processor, only 8 have been implemented. They are identified and associated with a tunnel that bears the corresponding instruction's name. In the case of `ldi`, the position of the register index in the bit string coincides with the `Rd` field, so the register file has no trouble latching the immediate operand in the correct register. The implementation of the whole class in the PD is obtained immediately by using a Platform 1 16-way decoder for the relevant four bits, and then OR-ing the first four outputs to produce the `ldi` signal.

A similar case of mixed decoding is observed in the group of stack-orientated instructions with a common prefix 0b1100. Here the first three instructions, `push`, `pop` and `lds` have the same format (one destination register), and the remaining four another (zero-operand instructions). Furthermore, the instructions `lds`, `addsp` and `sets` all have an immediate operand, while the rest of them do not. Unlike the previous group, we opted for two decoders one driving another, as this provides for a more compact solution.

The reader is encouraged to revise section 8.2 and check that there exists circuitry in the PD that derives the op-code signal and any necessary selectors for each instruction in the instruction-set diagram presented there.

### 12.5.3 CdM-8 Data Path

In section 12.4 we discussed an example data path which was complete enough to even execute some CdM-8 Platform 2 instructions. Of course neither the register file nor the ALU was up to the job of executing *all* such instructions. This was not important for illustration purposes, and we were easily able to see how the registers and the ALU were coordinated by a small set of triggers and selectors to achieve realistic functionality. Our conclusion was that an instruction machine is necessary to produce triggers and selectors of this kind. We noted that the set, although small, was highly redundant, easily twice to three times the length of the 8-bit CdM-8 Platform 2 instruction. Since instructions can take several cycles, the actual redundancy is even worse. We concluded that we needed a Secondary Decoder (SD) to produce the required

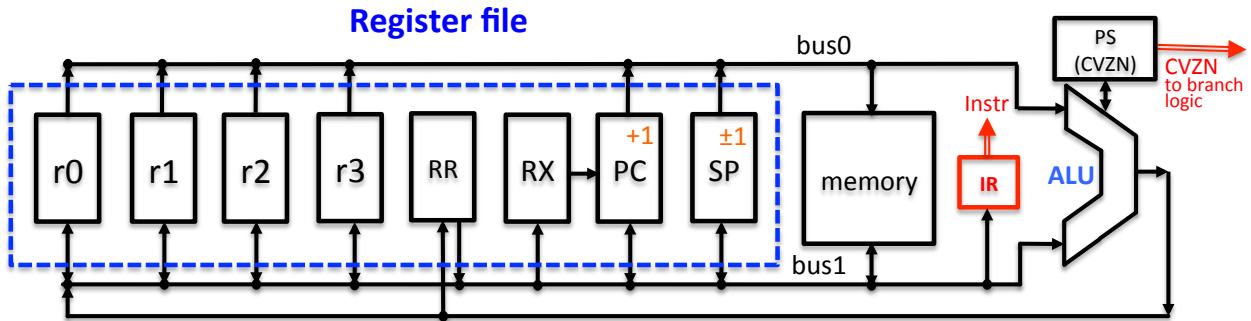
<sup>6</sup>Exceptionally, when the prefix is a single bit the AND-gate is not required for recognising its value.

<sup>7</sup>`jsr` has an operand but it is not a register and thus is not part of the instruction byte, hence the instruction is still considered a 0-operand one. Ditto for `addsp` and `sets`.

triggers/selectors automatically from just a few bits of data that contain all the information required for the purpose.

Before we are ready to discuss the SD, we must take a look at the complete data path to understand what signals (triggers and selectors) it requires from the SD to offer full platform functionality<sup>8</sup>.

**The CdM-8 register file** has a total of eight registers, four of which are the familiar GP registers and the other four are the PC, SP, RR and RX. While the first two are as familiar to the reader as the GP registers themselves, the intention of RR and RX is not as clear. Here is how they work. When the ALU performs a single-operand operation, such as `inc`, it uses as its data source only the upper bus, and can assert the result on the lower bus in the same clock cycle. However, as we saw in section 12.4, in order to execute an ALU operation on a 2-bus architecture such as ours, we require a result register in which the ALU result is latched when the Unit is using the two buses as data sources.



The DP solves this by offering the result register as well as the direct return path to the lower bus for the ALU result. The result register is called RR, it latches the result coming from the ALU every clock cycle, whether ALU was engaged or not, and asserts its content on the lower bus if triggered to do so. That is all the RR can do. The connection of the ALU output to the bus 1 input of the register file is governed by a controlled buffer, which opens only when the ALU performs Phase 1 of a single-operand using the trigger `alu-u1` from the SD.

The other register, RX, is used with `jsr` logic. When a `jsr` is executed, the current PC value must be pushed on to the stack before it can be overwritten by the immediate operand of the instruction. However, a stack push requires two cycles: one for decrementing the SP, and one for a memory write. Without an extra register, the `jsr` instruction would have to be 3-cycle.

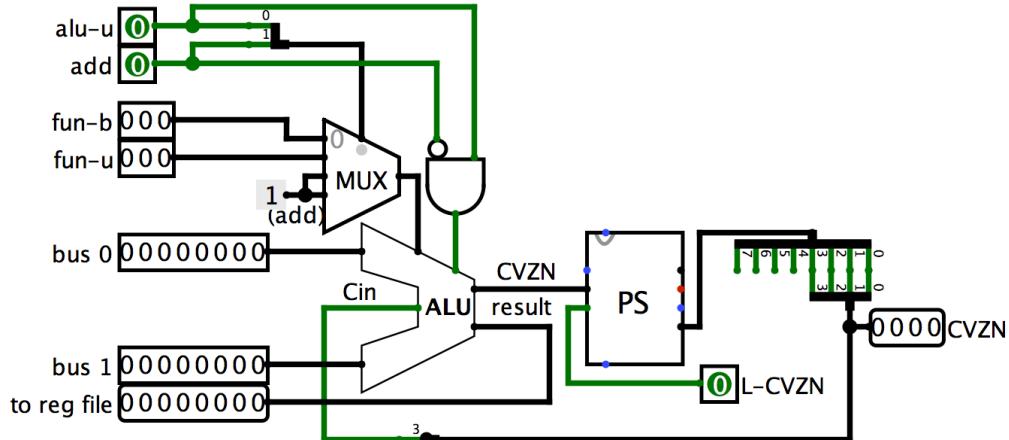
Enter the *exchange register*, RX, which is used in the first clock cycle: while SP is being decremented, the SD executes memory read and latches the immediate operand in RX at the end of the cycle. In the following cycle RX asserts its content on a *third* input of the PC while the upper bus is used to supply the SP content as a memory address and the lower bus to assert the *current* content of the PC as memory data for the write operation. Since the PC is a register, and any register is read first then updated, this works, while improving the speed of the `jsr` by a third!

Of course we did not need to worry much about optimising CdM-8 Platform 2 implementation for *speed* (while optimising such a small machine for memory space utilisation always makes sense). However, the RX tradeoff, namely an extra register for a 33% speed improvement, is instructive by itself and was retained in the processor design as the only example of hardware optimisation.

The PC and SP have their own arithmetic units, with the PC being able to increment its content on an external trigger and the SP to increment, decrement, or increase it (by an arbitrary 8-bit integer; this obviously works in 2's complement as well). The data path also has triggers and selectors for latching and asserting the GP registers. Some of the selectors are hardwired to the IR as the content comes from fixed sections of the machine instruction, others are provided by the SD.

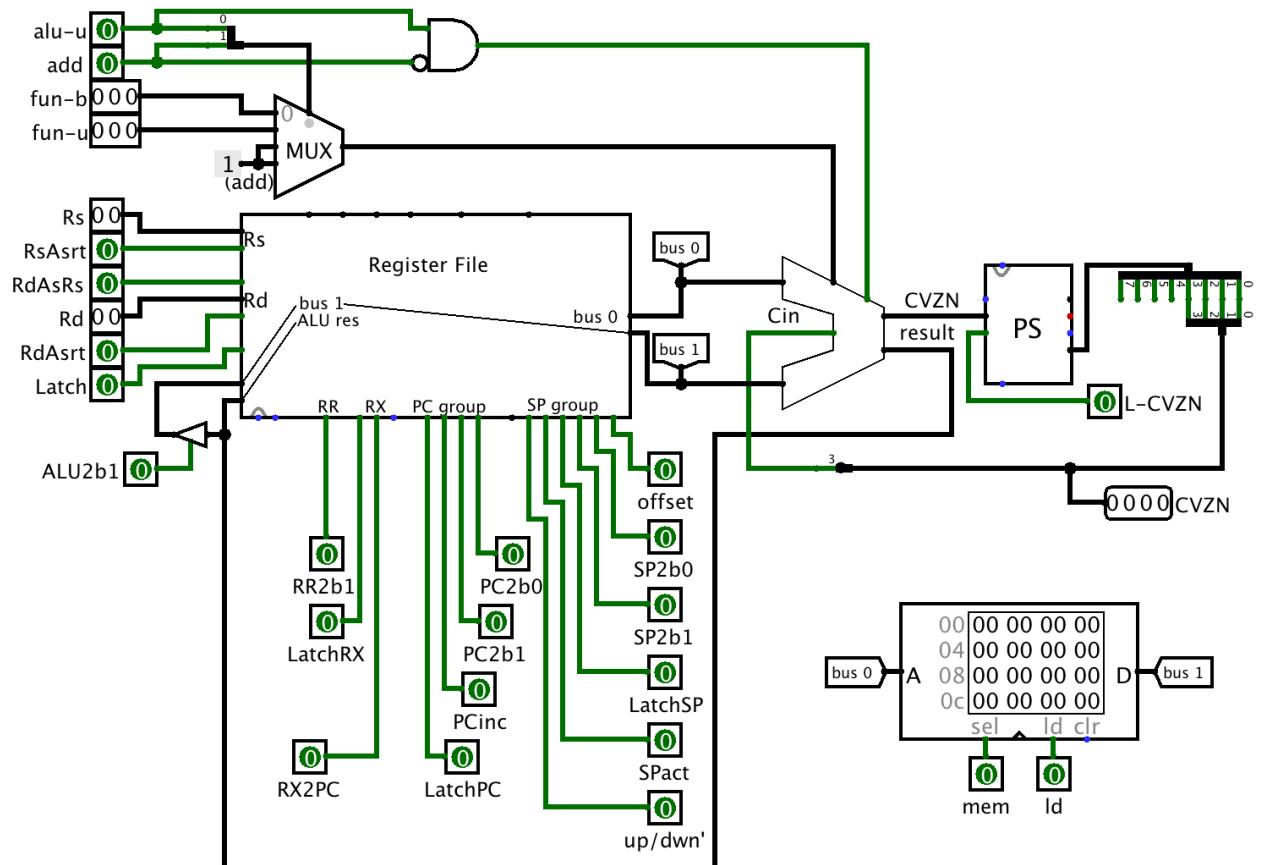
**ALU and Processor Status register.** The CdM-8 Platform 2 ALU implementation is straightforward, needing no detailed explanation. It is consistent with the view on subtraction presented in this text, namely that subtraction is identical to adding the 2's complemented second operand. The output value of CVZN is consistent with that view as well.

<sup>8</sup>Remember that the data path lies outside the instruction machine and is the principal client of the latter.



The 3-bit ALU operation code is multiplexed to one of the side inputs, and the other side input is the binary selector telling the ALU that the operation is a one-op one (such as `inc`). When the selector is down the ALU assumes it is computing a two-op operation. Thus the 4-bit ALU op-code is split 3+1. Apart from **bus 0** and **bus 1** data inputs, the ALU reads the C flag from the PS register, which it uses with some arithmetic and shift operations, and outputs the result and all four flags, the latter wired directly to the PS input. The PS only latches those in when it receives the trigger (**L-CVZN**) from the SD. The current value of the flags is wired to the PD to be used in the branch chip. Finally, there is an SD-produced selector **add**, which overrides both the ALU op-code and the **alu-u** signal for using the ALU as an adder when the instruction latched in the IR is not an ALU instruction.

**Implementaton.** An implementation of the CdM-8 Platform 2 data path is shown in a circuit diagram below. Notice that it has more features than we had a chance to discuss in our exposition of CdM-8 Platform 3. In particular the PS is 8-bit, not 4- as we have seen previously, and the remaining 4 bits are used to control multipage memory and interrupts, to be discussed later.



Signal name	Category	Description
ALU2b1	trigger	Assert the ALU result on bus 1
CVZN	4-bit data	Flags
L-CVZN	trigger	Latch CVZN in PS
Latch	trigger	Latch the data asserted on bus 1 in Rd
LatchPC	trigger	Latch PC from bus 1
LatchRX	trigger	Latch the data on bus 1 in RX
LatchSP	trigger	Latch SP from bus 1
PC2b0	trigger	Assert PC on bus 0
PC2b1	trigger	Assert PC on bus 1
PCinc	trigger	Increment PC at the end of the cycle
RR2b1	trigger	Assert register RR on bus 1
RX2PC	trigger	Latch PC from RX
Rd	2-bit data	Index of the <i>destination</i> GP reg to assert and/or latch on bus 1
RdAsRs	1-bit data	Use index Rd to select the source GP reg to assert on bus 0
RdAssrt	trigger	Assert register Rd on bus 1
Rs	2-bit data	Index of the <i>source</i> GP reg to assert on bus 0
RsAssrt	trigger	Assert GP register Rs on bus 0
SP2b0	trigger	Assert SP on bus 0
SP2b1	trigger	Assert SP on bus 1
SPact	trigger	Decrement or increment SP
add	1-bit data	If up, ALU is forced to add (op-code 0b001) instead of <b>fun-u/fun-b</b>
alu-u	trigger	if up, ALU does a single-op operation (e.g. inc), else 2-op (e.g. add)
ld	1-bit data	if <b>mem</b> is up, memory data is loaded on bus 1 at ld=1 or stored at ld=0
mem	trigger	if up, memory is active in the current cycle
fun-b	3-bit data	ALU op for binary operations
fun-u	3-bit data	ALU op for unary operations
offset	1-bit data	If offset $\wedge$ LatchSP, the SP <i>adds</i> its input to current content
up/dwn'	1-bit data	Direction of SP move: if up then inc SP, else dec SP

Figure 12.3: Signals of the datapath (including memory) in alphabetic order. Interrupt- and **pushall-** and **popall-**related signals not included.

The above circuit diagram shows a complete data pass with memory attached to it<sup>9</sup>. Its triggers and selectors are summarised in figure 12.3. The data path has a total of ten selectors, of which four are provided by the Primary Decoder: **fun-u**, **fun-b**, **Rs** and **Rd**, as is the trigger **alu-u**, used here as a selector for the correct ALU op-code and in order to feed the ALU result to either lower bus or the RR. The Secondary Decoder provides the selector **up/dwn'** that defines the direction of SP movement.

There are many more triggers around the data path than selectors and it is convenient to group them according to their intended function.

**GPR Triggers** operate GP registers. They are pinned out on the west side of the register file. Signal **RsAsrt** tells the register file to assert on bus 0 the content of the register indexed by either

- the **Rs** selector extracted directly from the IR, if **RdAsRs** is down, or
- the **Rd** selector extracted directly from the IR, otherwise.

The former option is in place when the instruction is a 2-operand ALU (or a memory) instruction and both operands are used as sources of data and hence need to be asserted on the corresponding bus each. The latter option is taken when the instruction is a single-op ALU instruction, so the destination register needs to be asserted on the lower bus rather than the upper bus, since it is the *first* operand of the ALU.

This looks like rather complicated arrangements but the intention is simple enough: the design of the ALU is easier to understand if 1-op operations receive their operand on the first input of the device.

<sup>9</sup>In reality the memory chip is outside not only the DP but the processor as well. The control signals **ld** and **mem** are provided as pins on the processor chip, and so are the two busses.

**RR Trigger** is a group of one. Whenever `RR2b1` trigger is up the Result Register asserts its content on `b1`. This trigger is supplied by the SD in phase 1 of a 2-op ALU instruction in the process of latching the ALU result (currently held in RR) in the destination register.

**The RX group** has a single driven trigger, `LatchRX`, which latches the exchange register from the lower bus. For completeness, and also for a possible instruction-set extension, the signal `RX2b1` is provided (but not used currently) making it possible to both read and write into RX supporting its use as a general-purpose auxiliary register.

**The PC group** has five (!) triggers, which is unusual for a register. The common three: `PC2b0`, `PC2b1` and `LatchPC` assert the Program Counter on the buses and latch it from the lower bus, respectively. The fourth trigger, `RX2PC` latches the current RX content in the PC and finally `PCinc` causes the PC to increment its current value using its own dedicated incrementer.

**The SP group** is fairly analogous to the PC group, the difference being that instead of an increment trigger, the SP group has a generic action trigger `SPact` and the selector `up/down`, determines whether the SP is incremented or decremented when the action trigger is up. For the purposes of the `addsp` instruction there is a selector `offset`, which indicates whether bus 0 supplies a new content (`offset` is down) or an offset to the existing value when the trigger `LatchSP` is up.

**The PS group** is the final group (of one). It contains a single trigger `L-CVZN`, which, when up, causes the Processor Status register to latch the ALU CVZN output at the end of the clock cycle.

**The memory device (RAM)** has two signals that the SD needs to take care of: the trigger `mem` and the selector `1d` whose meaning is obvious from the table description and section [11.12](#).

Interestingly we are very close to the destination of our architectural journey. We have now specified the computing engine completely. We have also built the core of the instruction machine. What is missing is the last link in the chain: a circuit that inputs the phase signals from the Sequencer and the decoded op-code triggers from the PD, and produces any necessary triggers to operate the engine in successive phase cycles. That is the Secondary Decoder and we are entering the last section of this chapter now.

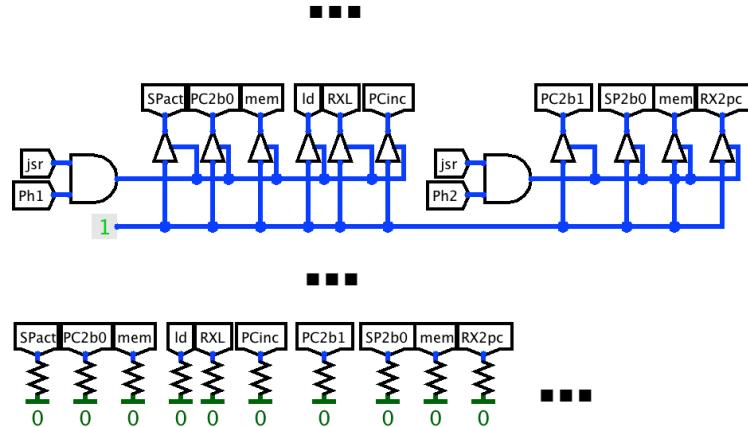
#### 12.5.4 Secondary Decoder

**Wire-OR.** Most triggers can be raised by several machine instructions/instruction groups. For example, `Latch` is used in ALU instructions to latch the ALU result. Also the trigger is raised in an `1d` instruction to latch the content read from memory in a register. The same is true of `1di`. The standard technique of dealing with such situation is to have a large OR-gate in the SD for each trigger, and to get the decoding circuit to feed the corresponding OR-gates when triggers need to be raised. Doing this would create a complicated network of wires making the circuit hard to read. Our approach will be less efficient but a great deal better structured; we use the technique known as Wire-OR, as follows.

The SD is organised as a set of disjoint handlers, each handler producing triggers for its corresponding type of machine instruction (or instruction class). When a handler is active, i.e. its input PD signal is high, it asserts 1 on some triggers in each of the phases. However, if the handler is not active, then the same triggers are *not* driven to 0, but simply left floating. This means that another handler, which *is* active, may assert 1 on it in the same clock cycle without causing a conflict. In order to drive the trigger down when it is not asserted by *any* of the handlers in a given clock cycle, and hence needs to be down at 0, we use the pull-down resistor. Any 1-bit (*data*) signals, such as `up/dwn`, or `RdAsRs` can be treated the same way for convenience; they are driven by the SD, so can be pulled down by resistors and pulled up by SD handlers only when the value of 1 is required in a given clock cycle. For data signals the meaning of the value 0 is not “no action” as it would be for a trigger, but merely a useful default which need not be asserted explicitly.

Consequently all the triggers produced by the SD (which are listed in figure [12.3](#)) are implemented as one-bit buses (and are tunnelled between the engine and the SD using tunnels named the same as the corresponding triggers), with each bus being equipped with a pull-down resistor. That way we can localise all SD circuitry relating to a single machine instruction/instruction class in one place.

**An example handler.** Here is the implementation of the `jsr` handler of the CdM-8 processor, chosen here as a substantive example of a 2-phase instruction<sup>10</sup>.



Let us discuss the operation of the `jsr` in some detail.

In the first cycle (Ph1) the Stack Pointer is decremented thanks to the `SPact` trigger and the fact that the selector `up/dwn'` is at its default 0, signifying decrement. The trigger `PC2b0` asserts the content of the PC on the upper bus, the selector `1d` is up and the memory trigger `mem` is active. We conclude that we are reading the memory cell pointed to by the PC currently (`PC2b0`), i.e. the byte following the instruction (since the instruction fetch always increments the PC). Now we see the signal `RXL` in the first phase, so we are reading the content of that byte into the exchange register. So `RX` will hold the immediate operand of the `jsr` instruction at the end of the clock cycle. Finally we see the `PCinc` trigger in the first phase and conclude that at the end of the clock cycle the PC will point to the byte after the immediate operand, i.e. *the next instruction*. That is clearly the return address of the `jsr`. To summarise, at the end of the first phase:

- $SP = SP - 1$
- PC points to the next instruction, i.e. *the return address*
- RX holds the immediate operand of the `jsr`, i.e. *the address of the subroutine*

On to the second phase, and we see that `SP2b0` and `PC2b1` and `mem` are active, and we conclude that the content of the PC, i.e. the return address, is stored (since `1d` is not asserted) in memory at the address given by the current value of the Stack Pointer, i.e., pushed onto the stack. Also `RX2PC` causes the PC to latch the value stored previously in the exchange register, which, we recall, was the address of the subroutine. To summarise, at the end of the second phase (which is the end of instruction execution as well):

- The return address is pushed onto the stack
- The PC holds the address of the subroutine that was provided as the immediate operand.

We conclude that the `jsr` handler has driven the data path and memory through two instruction phases to achieve precisely the effect of a `jsr` instruction. Each CdM-8 Platform 2 instruction can be implemented in a similar manner, most in a single phase some in two phases and some, including ones that we have not discussed yet, in up to six phases.

One could contemplate a handler chip in simulation which houses the AND gate and a certain number of controlled buffers connected to it, to make circuit diagrams less cluttered with trivial structures. However, once we know what an individual SD handler looks like, all we need is a table listing the output triggers against the input op-phase pairs.

The complete, and rather compact, definition of the CdM-8 Platform 2 Secondary Decoder<sup>11</sup> is found in fig 12.4:

<sup>10</sup>The attentive reader will immediately see that since the controlled buffers are fed with the constant 1 from below, a CMOS implementation (as opposed to an abstract one based on the Universal Platform 1) can replace the controlled buffer with a single P-type transistor, not the full transmission gate, without any signal degradation at all, which would also require the substitution of a NAND-gate for the AND-gate in the diagram. This reduces the cost of the SD roughly by a factor of four.

<sup>11</sup>the table does not include interrupt instructions covered later

Instruction	PD signal	Phase 1 triggers	Phase 2/ <u>Phase 3</u> triggers
N/A	Fetch*	PC2b0 ld mem PCinc	
2-op ALU	alu-b	RsAssrt RdAssrt L-CVZN	RRA Latch
1-op ALU	alu-b	RsAssrt RdAsRs ALU2b1 Latch L-CVZN	
ld	m-ld	RsAssrt Latch mem ld data	
st	m-st	RsAssrt RdAssrt mem data	
ldc	m-ld	RsAssrt Latch mem ld	
ldi	ldi	PC2b0 mem ld Latch PCInc	
push	pu	SPact	SP2b0 RdAssrt mem data
pop	po	SP2b0 mem ld Latch SPact up/dwn' data	
ldsa**	ldsa	PC2b0 mem ld Latch PCinc	RdA add SP2b0 <u>RRA Latch</u>
	addsp	PC2b0 mem ld SPL offset PCinc	
br taken	br-nop	PCInc	
br taken	br-taken	PC2b0 mem ld LatchPC	
jsr	jsr	SPact PC2b0 mem ld LatchRX PCinc	PC2b1 SP2b0 mem RX2PC data
rts	rts	SP2b0 SPact up/dwn' mem ld latchPC data	
crc	crc	SP2b0 mem ld LatchRX data	SP2b0 mem PC2b1 RX2PC data

\* The Fetch signal comes straight from the Sequencer and does not require primary decoding.

\*\* `ldsa` executes in three phases. Phase 3 signals are underlined.

Figure 12.4: CdM-8 secondary decoder (main part)

Notice that the Secondary decoder produces triggers and selectors for fetching the machine instruction from memory into the IR and incrementing the PC. The IR is triggered to latch by the `Fetch` signal directly, but it is convenient to produce the rest of the signals as if `Fetch` were a Platform 2 instruction decoded by the PD. This makes the SD nice and uniform and a universal enough tool for driving the engine.

Also notice the signal `data`, which we never mentioned before. It is raised whenever there is a memory operation in the current phase, which reads or writes data as opposed to instruction or its immediate operand. Instructions `ld`, `st`, `push`, `pop`, `rts` and `crc` raise it when they access data placed in memory or on the stack, while `jsr` raises it only in phase 2 when it saves the return address on the stack but keeps it low in phase 1, when it is reading the immediate operand.

The signal `data` has no useful function in a von Neumann system, i.e. such that data and instructions share the same RAM unit. However, the CdM-8 processor can work with a more complex memory organisation, which is what we are going to see next.

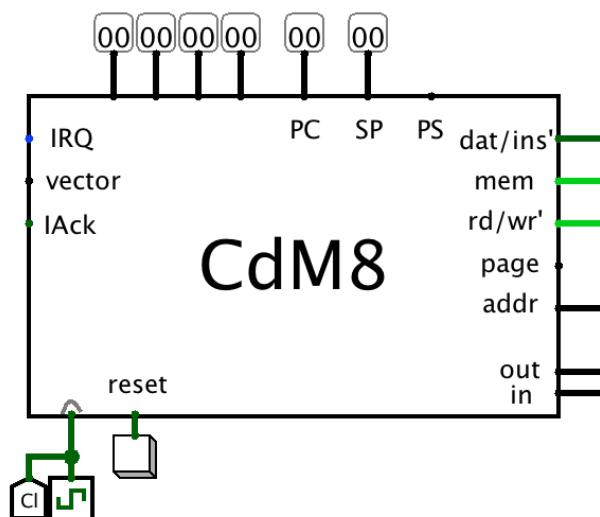
Interestingly, the PD gives a kind of syntax of the CdM-8 instruction set, defining which data fields and markers an instruction must have, while the SD provides its semantics. The latter is phrased in terms of a physical, rather than virtual, machine, the CdM-8 data path. However, the electronic circuit that defines the data path is formal enough to say that the engine has a rigorously defined behaviour, perhaps as much so, if not more, than the virtual machines (Level 4 platforms) of conventional programming languages.

This observation explains the value of detailed understanding of the bottom levels of the Tannenbaum hierarchy by a computer scientist. By contrast with electronic engineering, computer science uses those levels more as a metaphor of computing, and draws its inspiration from their immanent simplicity and elegance.

# Level 2 Systems Architecture and Input/Output

This chapter starts with an eagle-eye view again, only now the eagle must soar over the engineering, rather than conceptual landscape, which includes the world outside the platform and the marches populated by electronic and programmatic interfaces.

## 13.1 Processor



Having discussed the innards of the instruction machine and the data path, we feel justified to call abstraction in and present ourselves with a nice and compact box which has all the Platform 2 electronics inside, except memory. Such a circuit is also known as a *processor*<sup>1</sup>, and we have just built one. To be precise, the above box, which is available as a Logisim simulation from the book web site, has *more* inside than we had the space to discuss; some of the pins, such as *page* and the whole west side have not been encountered before. That is because we focused on the core functionality in the previous chapter, it had to be understood first before we were prepared to widen the scope and invite the outside world into our freshly built Platform 2. The box's terminals that are not connected to wires are supported inside the box with some circuitry built precisely for that purpose, and we will discuss some of these features in this chapter and the next. For now, notice a familiar pin: The terminal *dat/ins'* is nothing more than the *data* signal produced by the SD, which we saw at the end of the last chapter. We will see how to use it to build interesting systems a little later.

The processor chip has only a few terminals<sup>2</sup>. There are two pins: *in* and *out* connected to the lower bus, i.e. *bus 1*. Normally an in/out pin (or rather pin array) would be used for such purposes, but we separated

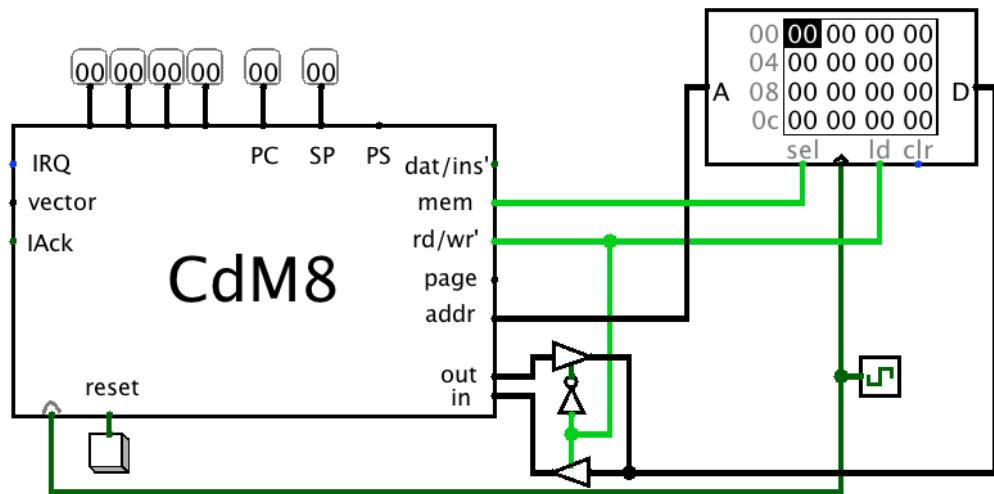
<sup>1</sup>Notice that we keep re-defining the term at ever decreasing levels of abstraction. All our definitions of processor are consistent with each other.

<sup>2</sup>Even though we often say that we do not distinguish real-life electronics from its crude computer simulation in this book, we must also remind the reader that wire bunches are terminated by arrays of pins, each connected to a single wire, rather than a multi-bit pin our chosen simulator prefers to show for reasons of compactness. This means that even a processor as simple as CdM-8 would have as many as thirty physical pins to support the terminals on the east side of the circuit symbol alone. However simulated copper is as cheap and as forgiving as simulated silicon, so let electronic engineers protest all they want.

out inputs and outputs because our chosen simulation technology is not sophisticated enough to manage in/out pins; this should not be taken as a general case, either in electronics or its simulation. The pin `addr` is connected to the top bus, `bus 0`, and works as output. The reason why it is so named is that the only purpose of reading `bus 1` outside the processor is to receive an address to use in conjunction with data asserted on, or latched from the pins `in` and `out`, respectively. The outside of the processor shares the clock signal with it (marked by a triangular shape around the south-west terminal), so the question of *when* the reading and writing of the data takes place does not arise. However, the question whether or not the processor is exchanging data in the current clock cycle via its memory interface (and this is exactly what the east side is, a memory interface) still remains; consequently the pin `mem` is provided, which is raised by the processor when it encounters one of its memory instructions or when it needs to fetch the next instruction from memory. Finally, the direction of the exchange, whether the processor is reading (loading) data or writing (storing) it is indicated on the output pin `rd/wr'`.

At the top are a few pins used for monitoring the internal state of the processor: those are (west to east): `R0-R3`, `PC`, `SP` and `PS`. The west side can safely be ignored throughout this chapter.

## 13.2 System View: von Neumann Architecture



Here we have a complete CdM-8 Platform 2 in one small diagram. The reset button triggers the `reset` pin on the processor, which is wired to all sequential logic inside. The processor needs to be reset before it can execute a Platform 2 program.

The `dat/ins'` pin is ignored since this system uses a single memory for instructions and data, which is, as we mentioned earlier, the essence of the von Neumann architecture. The memory shares the clock with the processor. No part of the outside world is present in the design, consequently all the platform can do is execute a program to a halt in order to obtain the result in memory (and registers).

In order to present a program to the platform, it needs to be loaded into memory. This cannot be done under the platform's own steam, since it has no connection to the outside world. In simulation we can simply load the memory-image file of the compiled program into memory and press the reset button to see it run.

Except for giving us the satisfaction of seeing one's program successfully run to completion computing the correct result at the end, the above solution does not seem to be particularly useful. Let us first of all deal with the volatility of the program loaded into RAM. We can load it into ROM and even run it from there, but we would also need a RAM unit to accommodate the data which is not only read from but also written into memory. It is possible to achieve that with the current processor by following what is known as the Harvard architecture.

## 13.3 System View: Harvard Architecture

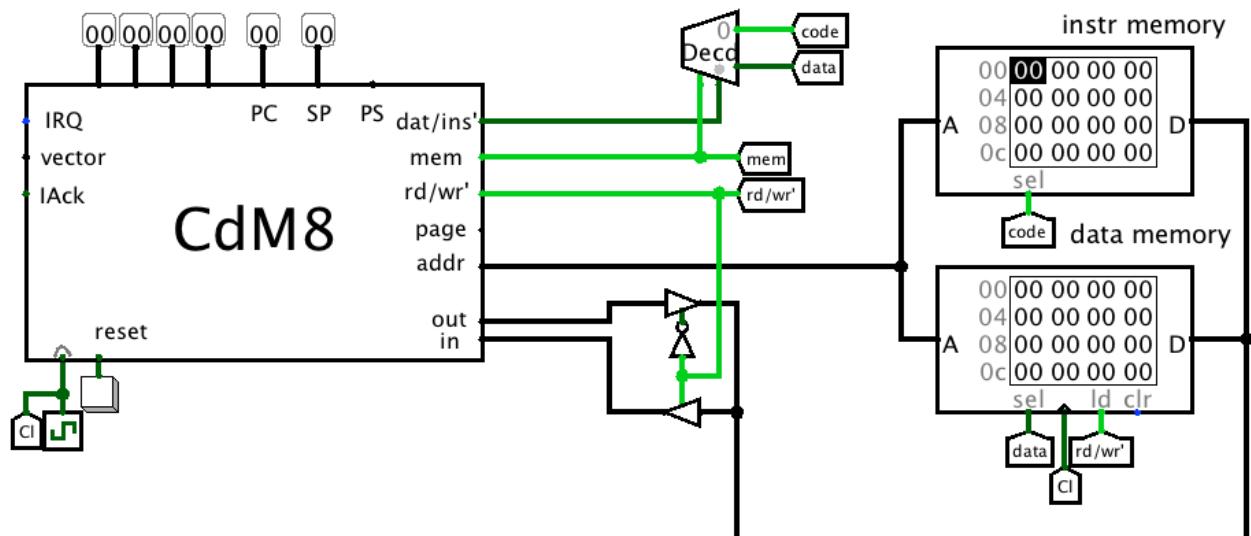
A Harvard-architecture Platform 2 uses two separate memories, one for program code and one for data, each of which has its own separate address space. The main advantage of Harvard over von Neumann lies in

the possibility of using both memories simultaneously (giving greater speed). However, as processor speed is not our concern in this book, we are not going down this route with CdM-8, because it would make the processor circuitry much more complicated. However, by adopting a Harvard architecture we may use 512 bytes of memory for larger system designs, one half for code and the other half for data. Programs need to be written a little differently to take advantage of such an architecture, but the benefits it brings are significant enough — and the differences small enough — to make this worthwhile.

### 13.3.1 Implementing Harvard architecture

From a programmer's perspective, the main differences lie in the use of load and store instructions, because we now need to specify which memory we are addressing. In our work with platforms we will treat code memory as read-only, because overwriting code at run-time can be disastrous. A Harvard architecture requires the CPU to indicate which memory (code or data) it needs to use in each clock cycle. Take for example the `push rn` instruction (which pushes the contents of register  $n$  on the stack). The processor requires code memory during the fetch clock cycle to decode the instruction. In phase 1 neither memory is addressed since the machine is busy decrementing the stack pointer SP and latching the new value in it (this will be used as a data memory address). Finally, in phase 2, the address (in SP) and data (in  $rn$ ) are asserted on the buses and data memory is engaged in the write operation. Another example: a `jsr` needs two cycles of code memory (to fetch the instruction and its immediate operand) and then one cycle of data memory (to operate the stack). The CdM-8 processor supports a Harvard architecture by asserting the signal `data`, which we met with in the last chapter, on a processor pin on the east side of the chip to enable the system to switch between code memory and data memory. But now we need a memory subsystem that contains more than just a single RAM chip.

Here is how it could be put together:



When the signal `mem` is down, the decoder is disabled and both `code` and `data` triggers are down, too. Nothing is asserted on the `in` pin of the processor. If the `mem` trigger is up, then either memory is engaged: if `dat/ins'` is down, it is the instruction ROM, otherwise it is the data RAM. If it is the RAM that gets selected, the signal `rd/wr'` is tunneled to the `1d` input of the memory chip; if the value is 1 then the memory stores data from the `out` pin, otherwise it loads from memory and asserts the value on the memory cell on the `in` pin. On the east of the processor chip the pair of controlled buffers triggered by the `rd/wr'` signal makes sure the correct processor pin is used in the memory operation.

### 13.3.2 Programming for split memory: the `ldc` instruction

How do we write programs if data and code are in different address spaces? Fortunately, there is very little difference in programming style. We only lose the compiler's ability to automatically plan memory for us. We can still use the `dc` pseudo-instruction in our programs, but any constants so defined are placed in code

memory. Also, whereas before we could use ‘ds’ pseudo-instructions to reserve space for storing results, a `ds` placed anywhere in the code is meaningless (because code memory is read-only). So, if we wrote this:

```
ldi r0, result
st r0, r1
...
result: ds 1
```

the compiler would substitute the address of a location in code memory for the label `result` but the `st` operation refers to the data memory with its separate address space. There, in data memory, the programmer can use the stack memory technique mentioned in section 6.4.2 to avoid the static planning altogether. The `lda` instruction would calculate addresses of data objects defined in an appropriate `tplate` section and the programmer would not be disadvantaged by the Harvard architecture in any way. The only thing lost with stack memory compared to static one is compile-time initialisation. However, it is a standard assumption for a Harvard platform that since the program is permanently placed in ROM, the RAM part of memory is not loaded with anything in particular at reset, so static initialisation is impossible.

Despite data memory being supported by a RAM that has no useful content upon reset, we may still keep constants using `dc` outside the program code, knowing that they will be safe in code memory. To make such constants usable, the CdM-8 processor includes an `1dc` instruction, see section 8.2. This works exactly like `1d`, except the `dat/ins`’ pin is held low by the processor; so the value is fetched from the code ROM instead of the data memory. Consequently, the following snippet is fine:

```
ldi r0, chars
1dc r0, r1
...
chars:
dc "a character string",0
```

The code above loads the first character of the string `chars` into register `r1`. Reading the sequence of characters in the string may be achieved in the usual manner. This technique may be used when we wish to include constants of any kind — and any complexity — in our code. Examples include arrays containing number tables, text messages and table-defined functions. Of course we cannot update the contents of the memory cells containing these values.

## 13.4 Memory mapped I/O

So far our platforms have been things in themselves. They were unable to interact with the rest of the world and served the sole purpose of taking the initial data into the final result. Our platform view was heavily biased towards perhaps not too important purpose of computing: crunching data. These days platforms are used for a much broader range of purposes, in which communication and control play the central role. Naturally, we had to start with the engine and its instruction machine, both on the software and the hardware sides, but now we are at a point in our journey when we are fully equipped to connect a platform to the world. In other words, we now concern ourselves with the machine’s *input and output* (I/O)

As far as the CdM-8 CPU is concerned, the data that we load from “memory” is just data from outside the core. It may come from any device to the `in` terminal of the processor. So any outside agent that takes an address from the CPU and asserts data on the data bus at the right time is good enough to make an `1d` instruction work. The same is true for data that we store into “memory”. Any system that takes an address from the CPU and accepts data that the CPU has asserted on the data bus is good enough for an `st` instruction, regardless of whether the data is actually written anywhere or not. The reason is that the processor memory interface is just that, an interface, a convention for interacting with memory. This would have been different if the CdM-8 Platform 2 language relied in any way on memory’s correct operation, something that is commonly referred to as ‘semantics’ in the world of higher platforms. For example if any of the Platform 2 instructions stored some intermediate values in memory for subsequent retrieval. In such a case the interface itself would not be sufficient, it would have to be supported by the actual ability to correctly store and retrieve information. But that is not the case in any mainstream Level 2 language, i.e. machine instruction set. Instructions are self-contained, their effect is local, and since modern processors

are orders of magnitude faster than memory, it is hard to imagine any of the machine instructions referring to memory both ways and relying on its consistent retrieval of previously stored data. This gives rise to a rather elegant solution to the problem of platform interaction with the outside world.

We can treat I/O as if it were the same as loading from and storing to memory. In order to do this we can give up access to a few bytes of memory and use the addresses they occupy for input from, and output to, peripheral devices. This is a technique known as memory-mapped I/O, and is commonly used by small systems. Each peripheral device is allocated a number of (data memory) addresses, some of which may be used to send data to devices, some of which may be used to receive data from devices, and some of which may be used both for sending and receiving data. Such devices can be very simple, e.g. individual LED lights that can be lit up by applying a high level on a connecting wire, or more complex, such as a keyboard asserting 7-bit ASCII codes on its output terminal when a key is pressed (or in our case a *simulated* keyboard, which can be typed on using the real keyboard when the simulator is focused by a mouse click on the corresponding screen control). Logisim allows us to simulate even crude video displays, joysticks etc. In real life, the choice of peripheral devices include such immensely complex machinery as graphic cards (which are themselves multiprocessors with thousands of processing units each of which is much more complex than the CdM-8 processor), network adapters, etc. We are not going to get anywhere near that level of complexity in this introductory text.

The simulated CdM-8 processor can be connected with a broad range of (also simulated) I/O devices, in fact all those supported by the last version of Logisim. When adequate interface circuitry is built outside the processor, it can treat any such device as one or more memory cells which have addresses and which can be read, and/or written into, just like ordinary memory.

### 13.4.1 Memory-mapped I/O organisation

Let us discuss the main issues pertaining to the practical organisation of input/output. Several question arise in conjunction with processor-I/O interface which we answer below.

**Which addresses should be used?** Normally a contiguous *range* of addresses is allocated for I/O, called the I/O segment (or, sometimes, the I/O page). It makes sense to slice off a section of the address space for I/O so that programs can use a contiguous segment for data storage as well. Typically the I/O segment is placed at the top of the address space, so that data segment can start at address 0. For a CdM-8-based system with its inherent limitation of the address space to 256 bytes, it makes sense to allocate no more than 16 addresses to I/O, placing them in the address segment 0xf0–0xff. This may accommodate 16 devices or more, since two devices may share the address by utilising different data bits at it. On the other hand some devices may need more than one address if 8-bit data is transferred, since control bits would also have to be accommodated.

**How do we deal with the differences in speed between processor and I/O devices?** Peripheral devices are typically much slower than the processor, and they may not be controlled by the same clock, or even operate in real time, for example the keyboard has data to be sent to the processor when the human operator presses a key — and that has no relationship with any clock whatsoever. Direct communication is therefore problematic, so we must provide one or more “holding areas” for data, so that the device and the processor may “drop off” and “pick up” data in their own time.

A device interface circuit must be constructed to support that. The circuit will contain a number of I/O registers, each of which is used to hold data on a temporary basis. Some of the I/O registers will be visible to the processor and others will be internal to the interface and the I/O device it supports. As far as the processor is concerned, each *visible* I/O register looks like a memory cell (in data memory), with an address in the I/O segment of the address space (an I/O address). The peripheral device can also access the register in order to read processor-supplied data from it or store device-related information there.

**At what time can the I/O registers be read and written to by the processor?** A device will typically provide facilities for a processor to control its operation, and to determine its status at any point in time. Without these facilities the device or the processor may attempt to update an I/O register before the other has read its current value, or may attempt to read the next data value from an I/O register before

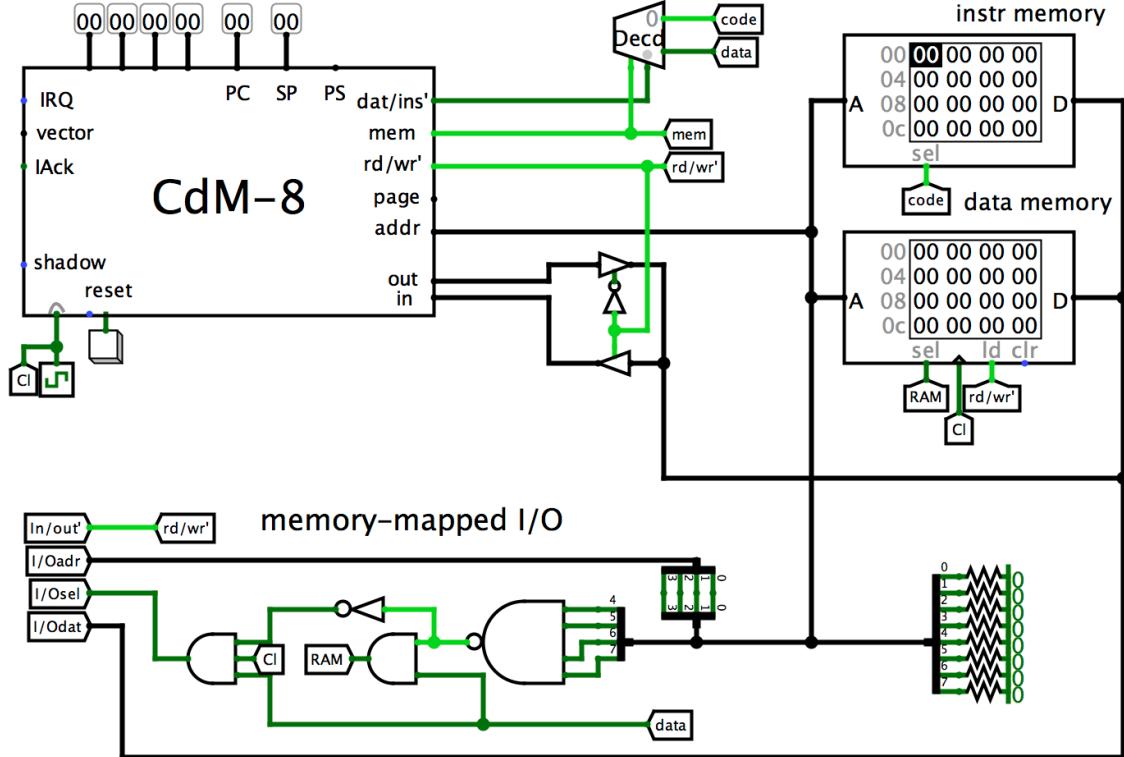


Figure 13.1: Memory-mapped I/O

it has been updated by the other. What we need here is so called *synchronisation facilities*, i.e. the ability to make sure that both sides of a transaction are ready to engage, and consequently that the data to be transferred is valid.

### 13.4.2 CdM-8 I/O bus

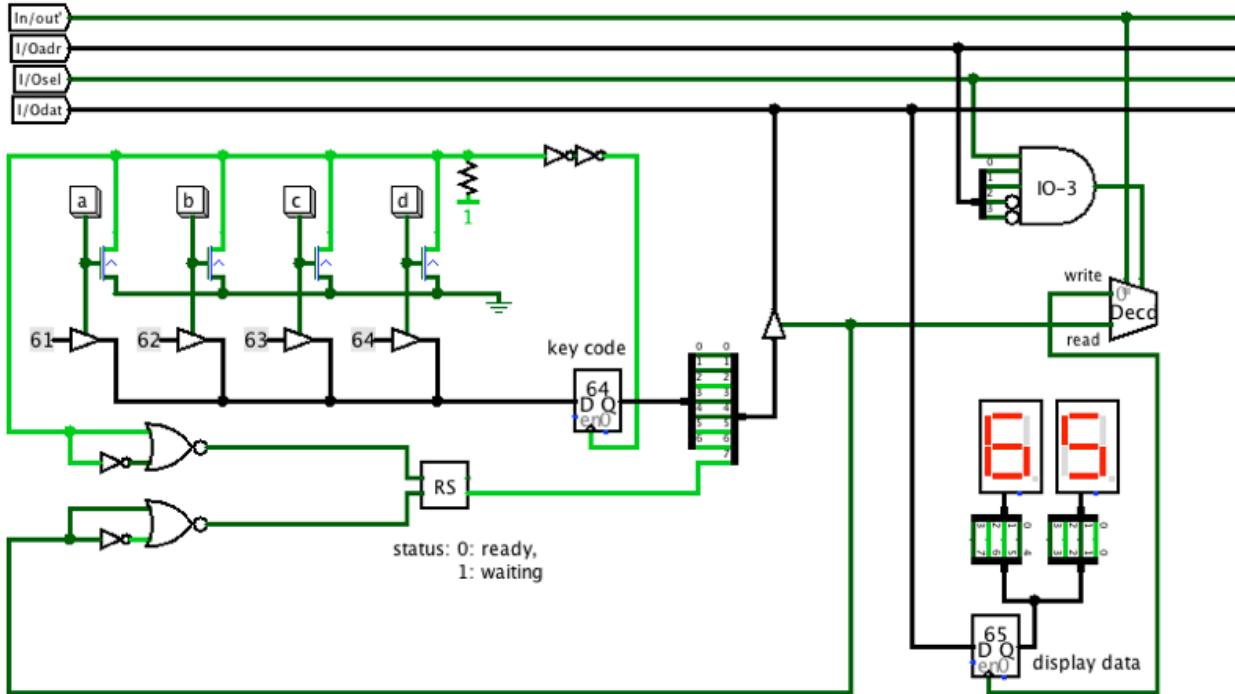
Figure 13.1 shows how memory-mapped I/O is handled by CdM-8. When the `dat/ins'` and `mem` signals are both high, the processor is addressing data memory. If all of the 4 most significant bits of the address are 1 (the address is in the range from 0b11110000 to 0b11111111) the `I/Osel` signal is high and the `RAM` signal is low. Alternatively the `I/Osel` signal is low and the `RAM` signal is high. The `I/Osel` signal is correlated with the clock by the west AND-gate, so it plays the role of clock for the I/O registers. The `I/Oadr` signal is a 4-bit bunch carrying the 4 lower-order bits of the address, identifying one of the I/O registers. Finally, `In/out'` plays the role of `rd/wr'`, and `I/Odat` serves the same purpose as the data terminal for the I/O registers. The four tunnels in the southwest corner collectively represent the I/O bus, which can be tunneled to a different part of the system circuit for the purposes of interfacing the I/O registers on the project circuits with the CdM-8 platform.

Notice that the I/O bus logic is implemented using ordinary gates and so must not have its inputs floating. On the other hand, the processor only drives its `addr` pins when `mem` is high. The reason for it is that the processor expects potential other users of the address bus to be there when it does not need to talk to memory<sup>3</sup>. To prevent chaotic behaviour of the I/O bus when the address is not being driven, its address part is pulled down by grounded resistors as usual.

### 13.4.3 Peripheral interface: example

Let us consider a complete peripheral device interfacing with CdM-8.

<sup>3</sup>It is the case when Direct Memory Access (DMA) discussed in the next chapter is used.



**A simple input device.** The above circuit contains four buttons labelled ‘a’, ‘b’, ‘c’ and ‘d’. South of each button is a 7-bit buffer asserting the ASCII code of the letter corresponding to the button that is pressed. What we have here is a little keyboard.

**Sending signals from keyboard to processor.** When a button is pressed its corresponding ASCII code is fed into the 7-bit key code register. The transistors connected to the buttons are set up in an arrangement known as “Wire OR”. They collectively implement an OR-gate (in fact, a NOR gate, but we know that CMOS transistors always invert signals). As long as no button is pressed, all transistors (N-type) are closed and the pull-up resistor ensures 1 is asserted on the horizontal wire to the north of the four buttons. At a press of a button, one of the transistors grounds this wire. This is a standard method of deriving a trigger from an array of buttons. The trigger causes the key code register to latch the constant value associated with the button.

In order to ensure that the latching occurs at the right time we have used two inverters connected in series (to the east of the pull-down resistor) to implement a signal delay circuit. So the trigger is delivered to the register after the 7-bit ASCII code has been asserted by the button press. We need to arrange signals in time order locally, because the timing of button presses is not governed by a clock. This is known as self-timing: both the data and its validity trigger are derived from the same local event: a button press. A peripheral device is often driven by a human, or some other external agent: this is the real trigger, so self-timing (and its inherent timing pitfalls) is an unfortunate necessity.

The value held in the key code register provides the 7 low-order bits of the bunch that is connected to the processor’s I/O bus. The most significant bit, bit 7, is derived from a separate RS flip-flop, and has special significance. As long as no button is pressed, the latch providing bit 7 is held at 1 signifying that the keyboard is waiting for a button press. The moment any button is pressed, the latch is changed to 0, indicating that the register contains new data. A read signal from the processor re-sets the latch to 1, and bit-7 now remains at this level until the next button press occurs.

**Reading data from the simple keyboard.** The program running on the CPU will contain a loop that repeatedly reads the 8-bit value from the I/O register and tests bit 7 (using a `tst` instruction to set/clear the N flag). Each time bit 7 is 1 (the N flag is set), the program will go back and read another value from the device’s address. As soon as it reads a data value in which bit 7 is 0 (the N flag is clear), the program will know a button has been pressed and that the 7 low order bits form the ASCII code associated with that button.

**Synchronisation with the keyboard.** The synchronisation is implemented using an RS flip-flop. Unfortunately Logisim only has a clocked variety, so we use the standard (unclocked) one from our own CdM-8 circuit library. The key technique here is one of converting edges to pulses. There are two circuits, each made up of an inverter and a NOR gate, connected to the east of the RS flip-flop. These are falling-edge-to-pulse converters, which we have met before, see section 11.7.

When a button is pressed, the wire OR will produce a falling edge, and the northern falling-edge-to-pulse converter will in response send a very short pulse (an inverter-delay short) to the RS flip-flop. This will set it, signalling that the data is ready. The reason why it is important that the pulse is short is that we do not wish the RS flip-flop to be reset by the processor while the flip-flop is responding to the set pulse. Fortunately, there is no way a processor can respond in less than one full clock cycle, which is many times the duration of an inverter delay. So the RS cannot be commanded to set and reset at the same time, which would result in unpredictable behaviour. This solution is safe and valid.

The processor will read the value of the key code next time it executes an `1d` from the address of this device. When that instruction is executed, the decoder on the east side will decode a read. This will produce a pulse for the duration of the clock cycle, since the read signal is derived from I/Osel and that in its turn is based on the clock. On the falling edge of I/Osel the southern falling-edge-to-pulse converter will produce a short pulse (as short as that from the other converter) for the reset input of the RS flip-flop. That will happen after the data from the key code register has been read by the `1d` instruction. The flip-flop will reset, and since the splitter is connected to the Q' output (the lower one) of the flip-flop, bit 7 will rise again to signify that the circuit is waiting for the next button press (the data has already been read).

**A simple output device.** Now let us turn our attention to the east side of our example peripheral interface. Here we find a circuit that drives two hex indicators. So we have a simple two-digit hexadecimal display.

**Sending signals from processor to display.** This is an easier circuit to implement, and to understand. An 8-bit display data register is provided to hold the byte whose value will be shown on the hex display elements, each of which requires a 4-bit word, encoding a single hexadecimal digit (0..F). The westernmost of the two display elements receives its data from bits 7..4 of the register, and the other display element receives its data from bits 3..0. Thus, if we store the bit-string 0b11101001 in the data display register we will cause the display to show the hex value E9.

The trigger for the data display register comes from the decoder (more about this later). The only really interesting aspect of this is that the data is taken off I/Odat on the *rising* edge of the trigger (this does not show on the circuit diagram, but the register attributes are as mentioned). This goes against what we have seen before. Surely we need a falling edge, since data from the data sources has to go through combinational logic (ALU, MUXs etc), which takes time, so it is only guaranteed to be valid (i.e. to have arrived) at the falling edge. Or is it?

The reason why the register is triggered by the rising edge of a clock pulse is that the trigger is a distant derivation from the clock. By the time it has traversed the memory-mapped I/O logic, and also undergone the selection in the interface, the CPU clock signal may have risen and be falling again, so data asserted by the data path and delivered directly to the I/O device may be in the process of changing, and cannot be relied upon to be valid.

The main reason for triggering the data path with the falling edge is to give the ALU and other combinational circuitry a chance to complete its work before results are latched. This does not apply here: the data for an I/O device always comes from an `st` instruction, which asserts the data and the address (from which our trigger is derived) at the same time. By the time the trigger has propagated through the memory-mapped selection logic and is about to rise at the I/O device, the data, which is read directly from the processor bus (relabelled as I/Odat) has already been valid for some time, so it is fine to latch it on the rising edge, thus neutralising the skew.

**Writing data to the display.** There is no need to read the status of the display as it is in fact just a visualised register, and the only thing that can be done is to send it bytes. To drive this device we load its address into one register, the byte to be displayed into another, and then use a store instruction to send the data to the display.

#### 13.4.4 Address sharing.

Because the processor does not need to write data to the keyboard, and it does not need to read data from the display, they can share the same I/O address. Of course this requires care when programming, because an **ld** from that address means reading data from one device (the keyboard) and an **st** to the same address means writing data to a different device (the display).

The two devices in the example are located at the same address. One or the other of them can be enabled by applying the appropriate input signal to the gate labelled **I0-3** here. This gate's output goes high when its inputs (the 4 low-order bits of the address) are **0b0011**. Thus the address of the two devices is **0xf3**. The decoder is used to select which of the two devices is enabled: the keyboard when **In/out'** is high or the display when **In/out'** is low.

In our example things are even more complicated, because the input device and the output device employ different interpretations of the same bit-string. In this case, when the 'a' button is pressed on the keyboard the bit-string read by the program is **0b01100001**. If a program sends this bit-string to the display it will show the hex value **61**. Here is a program that reads key codes (in ASCII) in the above example and displays their hex equivalents on the display:

```
    asect 0xf3
IOReg: # Gives the address 0xf3 the symbolic name IOReg
    asect 0xf0
stack: # Gives the address 0xf0 the symbolic name stack
    asect 0x00
start:
    setsp stack      # sets the initial value of SP to 0xf0

    ldi r0, IOReg # load the address of the keyboard and display in r0

readkbd:
    do      # begin the keyboard read loop
        ld r0,r1      # load r1 from data memory, which includes
                        # the I/O address space
                        # now bits 0..6 of r1 encode the last char typed,
                        # and bit 7 indicates the keyboard status
        tst r1      # flag N is taken from bit 7 of the register
    until pl      # drop out of the loop when the N flag is 0
                    # and now r1 contains the ASCII code of the last char
        st r0,r1      # display the hex of the ASCII code

    br readkbd      # go back to the start of the keyboard read loop
end
```

### 13.5 Interrupts

The above example gave a taster of synchronisation issues occurring in a platform's interactions with the outside world. Let us cast our mind back on what we have learned.

The processor can signal its intentions by writing to an I/O register, and since the data is synchronous with the act of writing, it is guaranteed to be valid. The device (interface) will simply watch for the memory cycle initiated by the processor. If the device is not ready to accept the data (for example, it is a mechanical typewriter and it is busy printing the current letter, which takes macroscopic time in the platform world), the data can be buffered off (i.e. stored temporarily). But even then the overzealous program can eventually overwhelm the buffering capacity of the interface, so some protocol based on signalling both ways via I/O registers is necessary in the general case.

The situation is markedly worse with input. The data is no longer synchronous with the act of reading; some data is always present in the I/O register but it may or may not be valid at any given time. Consequently synchronisation involves the device signalling to the processor that the data *is* ready/valid, and such signalling is possible in two ways.

- As we saw above, the device could raise a flag (a bit in an I/O register) that signifies readiness. However, the processor has no way of knowing that the flag has been raised, so the program has to check *periodically* whether it is or it is not yet. Besides being a nuisance in program design (the flow of computations has to be periodically interrupted to check the flag) this also presents a difficult timing problem since if the flag is not checked often enough, the data could be lost due to the device's lack of buffering capacity, and if the flag is checked too often, the program will have little time left to do something useful.
- As a new thought, the device could also actively interrupt the processor running a program. That has all the advantages: promptness, low cost, separation of concerns, etc. — but is technically quite complex and requires processor modifications. We will dwell on this in the current section.

How do we enable the device interface to interrupt the program when it is ready to communicate? The key requirement for such interruptions, technically termed *interrupts*, to be useable and useful are as follows:

- The program should be able to decide whether it may be interrupted or not. There can conceivably be work more important than processing a specific device's interrupts.
- If the program allows interrupts, such events must be *serviced*: there should be a special program that gets the control when an interrupt happens. The program, usually referred to as the Interrupt Service Routine or ISR, can be specific for either a device, or a class of devices, e.g. a group of sensors, or indeed a single ISR could serve the whole set of peripheral devices connected to the platform. The purpose of the ISR is to send/receive some data, store it in memory and possibly attract the main program's attention when the data transfer, which may involve a series of interrupts, has been completed. Upon the completion of a single interrupt, an ISR must transfer the control back precisely to the point in the program where the interrupt occurred, so the main program may continue as if the interrupt never happened.
- Since an interrupt can occur at *any* point in the program as long as the interrupts are enabled, the main program is completely defenceless against disruptive resource sharing with an ISR. The registers can be shared safely by following the protocol whereby the ISR saves and restores any registers it intends to use. However, an interrupt can perfectly legitimately occur between a CVZN-setting instruction and a branch that follows it. This means that the flags as well must be saved and restored, or rather the whole PS Register, since, as we shall see soon, the Processor Status includes more than just the flags.

An interrupt must not disrupt the execution of a machine instruction, since the sequencer's state (phase number) is contained inside the sequencer and so cannot be saved or restored. This means that when a device signals an interrupt request, the request can be acted upon *only* at the beginning of the Fetch phase, i.e. when one Platform 2 instruction has been executed and the next one has not been fetched yet.

To implement interrupts the following **six extensions** of the platform architecture we have studied so far are necessary:

**Interrupt Master (IM)** , which senses interrupt requests, is able to identify the device from which a request has been received, grant it if the processor status permits it (i.e. notify the device that its interrupt has been granted, which means that an ISR is effectively in progress), and tell the sequencer that the normal sequence must be interrupted at the next Fetch. The IM holds the device ID until the start of an ISR, so that the latter can be chosen based on the former.

**Interruptable Sequencer** is a sequencer to which IM can signal the success of an interrupt request so that the latching of the Virtual Interrupt Instruction (VII) may proceed instead of a normal Fetch based on the PC. The Sequencer produces the necessary signalling for the Instruction Register to latch the VII.

**Extension of the Secondary Decoder** to accommodate the execution of the VII, which also needs the Sequencer to sequence more than the usual number of phases (due to a large amount of execution context that needs to be saved and later restored).

**Extension of the Instruction Set with the `rti` instruction** The instruction `rti` (return from interrupt) is the last instruction that an ISR must execute. This instruction restores the processor to the state that it was in when the interrupt occurred.

**Interrupt Arbiter** is needed outside the processor to be able to correctly manage interrupt requests and acknowledgements in a situation when several devices may request interrupts at unpredictable times without waiting for one another.

**Process Controller** is a *software* component that helps the ISR to signal to the main program that a series of interrupts associated with an I/O exchange has been completed and that the main program is able to proceed with data processing. For example, in inputting a line from the keyboard, when each key press causes an interrupt, the Enter key additionally signifies the completion of the line. Only the completion is significant to a main program that expects a line of text.

The process controller also relies on the facility that enables the processor to stop the clock in anticipation of an interrupt when *all* the remaining work depends on the I/O data. The facility is useful in order to save power, and it is usually a `wait` instruction, which has the aforementioned effect. The CdM-8 instruction set also includes a 0-operand `wait` instruction of this exact nature, see figure 8.1 in section 8.2.

The last bullet in fact merits a separate tome as it is the key topic of the subject known as “Operating Systems”. We will provide a brief introduction to it in the next chapter.

### 13.5.1 Interrupts: a system-level view

Figure 13.2 presents a system-level view of an interrupt-capable platform. The bottom part of the diagram is almost identical to the instruction machine view we have seen earlier (figure 12.1). There are some new elements though, as follows:

**New trigger `int`** which makes the instruction register latch the aforementioned Virtual Interrupt Instruction in a given clock cycle instead of receiving an instruction from memory.

**Changes to the Processor Status Register (PS)** which is now shared with the data path just like the PC, and which can consequently be read and written into using the DP buses. Its new role in the instruction machine is to hold the *interrupts enabled* flag in addition to the familiar CVZN. In CdM-8 Platform 2 both halves of the PS register are controlled by independent triggers allowing CVZN and the other four bits to be latched independently, but also the register as a whole can be latched and asserted.

**New selector `int enabled`** which is produced from bit-7 of the PS register and which is used by the Interrupt Master (IM).

Let us now move to the top of the diagram. The Interrupt Master sits on top of the Sequencer. It receives the interrupt request `IRQ`, and when interrupts are enabled, it acknowledges the request (`IAck`) to let the device know that interrupt is currently taking place, and at the same time it signals to the Sequencer to initiate execution of the VII. In doing so it latches the identification of the device that has caused the interrupt for use with the instruction. The mechanics of that is quite complex, it involves the so-called 4-phase signalling protocol, which will be discussed later.

Finally the east side of the IM is connected to the world outside the platform. It is not our first excursion to the wider world, we have already seen how the engine can be extended using the memory-mapped I/O technique to include device interfaces for data exchange. Here we see that the instruction machine, hitherto completely ensconced in the processor has broken through its shell and acquired its own bus, an *interrupt chain*, which runs around all the peripheral devices that need to be served by interrupt. So both fundamental parts of the platform, the engine and the instruction machine are open to the outside, which makes any statements about platform usage extremely application-dependent. In this book we will try to emphasise common features, broadly used in practice. The reader is nevertheless advised to not over-generalise any conclusions: our purpose is to show a complete enough *example* of an open platform, and in doing so we have neither the scope nor the ambition to be comprehensive.

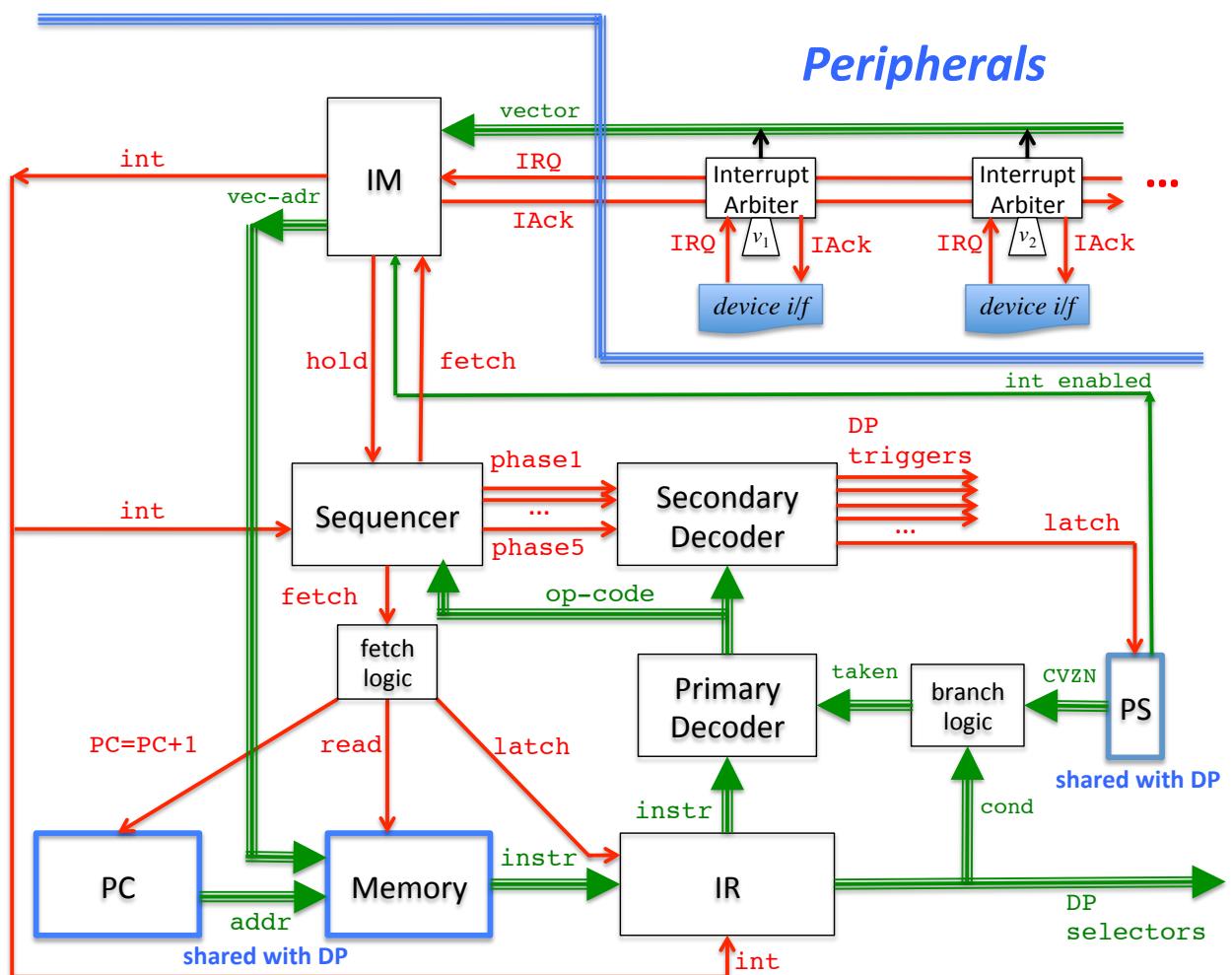


Figure 13.2: Instruction machine with interrupt capabilities

The peripheral architecture in figure 13.2 is based on a common concept of *priority chain*. In order to understand its principles let us consider a peripheral environment consisting of several devices each of which requires interrupt service. The processor has a *single IRQ* signal, to which it responds with a single acknowledgement. If more than one device requires interrupt services at the same time, which is quite a common case, only one of them should be allowed to receive the acknowledgement and the rest must wait with the interrupt request holding up. Naturally, if a device wishes to exchange data and the processor is not ready to do so because, for example, it is busy exchanging data with another device via its memory-mapped I/O circuitry, then, as we explained earlier, data might be lost. So it makes sense to arrange the interrupt network as shown in the figure, with higher priority devices with real-time<sup>4</sup> constraints or low buffering capacity sitting nearer the processor (or rather its IM) than low-priority devices that are either free of such constraints or at least whose buffering capacity is sufficiently large not to worry about them.

The Interrupt Arbiter (see the figure) has two client interfaces (south and east) and one server interface (west). The client interface consists of the signals **IRQ** (input) and **IAck** (output). The server interface is vice versa: **IRQ** (output) and **IAck** (input).

The Arbiter works as follows: when the south is quiescent, it acts like a pair of wires for the east client. The **IRQ** from the east is passed on to the west, and the **IAck** from the west is passed back to the east unimpeded. The moment that the south client raises its **IRQ** the arbiter waits for the east to finish its transaction, i.e. it waits for the east **IAck** to go down if it is not down already, and then temporarily blocks the east: its **IRQ** becomes disconnected and **IAck** grounded. At the same time it starts acting like a pair of wires for the south, until the latter is finished, at which time the Arbiter resumes its action as a pair of wires for the east. Consequently all requests from the devices/arbiters further east of a given arbiter must wait for the south client to finish its transaction. The latter only waits for at most one transaction on its east before it can attempt to propagate its request towards the processor. It is easy to see that the chain acts as a priority mechanism with the priority level rising from east to west.

Now let us examine the software side of interrupt processing before taking a closer look at the implementation of the six extensions that form the interrupt processing subsystem.

### 13.5.2 Interrupts from a program's point of view

In this section we describe a specific solution, the CdM-8 interrupt subsystem. The solution may differ in details from an off-the-shelf microprocessor, but the principles are well established and tend to be common to most practical platforms.

Every interrupt-capable device interface connected to a CdM-8 processor is identified by a unique number  $R$  in the range from 1 to 7. With each number is associated an *interrupt vector*, which is a pair of memory cells with consecutive addresses  $v_R = (0xf0 + 2R, 0xf1 + 2R)$ . For example for  $R = 6$  the addresses are **0xfc** and **0xfd**. The interrupt vector can be referred to by the starting address, i.e. vector **0xfc** or the  $R$ -number of the interface, i.e. vector 6. This causes no ambiguity.

The pair of cells in a vector have a clear function: the first cell holds the pointer to the Interrupt Service Routine associated with the device, and the second cell contains the Processor Status content to be set in the PS register *before* passing the control to the ISR.

**Placement of vectors.** Which memory, instruction or data, is the vector expected to be deployed in? In a von Neumann system the question has no meaning, since there is only one memory for the processor to work with. Notice that in the von Neumann case the interrupt vectors share the top 16 addresses with our chosen location of the I/O segment, so the memory-mapped I/O architecture would need to “know” which category each of the 14 cells belongs to: an interrupt vector or an I/O register. This rather limits the use of a von Neumann CdM-8 system with interrupts.

In a Harvard solution the vector is fetched from the *instruction* memory, whose address space is not shared with I/O and consequently the top 14 addresses are not special. They can all be used as vectors, leaving the whole 16 byte I/O segment available for I/O registers.

Finally, in a *multipage* Harvard architecture, which we will see in the next chapter, the interrupt vectors reside in page 0.

---

<sup>4</sup>in Computer Science *real-time*, or sometimes wall-clock time, means the time expressed in physical units, such as milliseconds, as opposed to *logical time* which is a partial order of events. A real-time constraint is the requirement that two events occur with the time interval between them not exceeding a certain physical limit, e.g. one video frame period.

**Interrupt sequence.** When an interrupt is granted to a device  $R$  by the processor, the latter executes the VII, which is part of the CdM-8 instruction set: the instruction mnemonic is `ioi`. It is virtual because it is not contained in the program, and consequently is not fetched from (instruction) memory, but is instead latched by the Instruction Register all by itself, under the `int` trigger from the Interrupt Master. Which clock cycle it is executed in is determined entirely by the IM based on requests from peripheral devices and the interrupt enable flag (bit 0x80) of the PS register.

The instruction `ioi` is nevertheless a true CdM-8 Platform 3 instruction and can be used in an assembler program for CdM-8 Platform 3. If used that way, its timing is defined by the control flow in the program: the instruction will be executed as soon as it is pointed to by the PC at the Fetch phase of the processor. Because the IM is not involved in such execution, no vector will be supplied to the `ioi` instruction and the processor will assume the originator of the interrupt to be a nonexistent device  $R = 0$ . It will fetch vector 0xf0 accordingly. Programmatic execution of an `ioi` may be useful, among other things, for debugging an Interrupt Service Routine.

The `ioi` instruction induces the following sequence on the data path<sup>5</sup>:

**Phase 1** decrement SP for stack push

**Phase 2** store PC on stack; decrement SP for stack push

**Phase 3** store PS on stack

**Phase 4** fetch new PC value from vector's first cell ( $0xf0 + 2R$ )

**Phase 5** fetch new PS value from vector's second cell ( $0xf1 + 2R$ )

The above resembles a subroutine call but with a major difference: the “subroutine” address is fetched from a location determined by a peripheral device rather than the computer program. Another difference is that the full PS is saved on the stack as well as the return address, and the “subroutine” (in reality, the ISR) has not only a start address, but also an initial PS content. Consequently an ISR can be *completely* transparent to the main program, whereas a subroutine will generally alter the CVZN flags<sup>6</sup>. Also the ISR is free to either enable or disable interrupts for the duration of its execution irrespective of the regime established in the main program. This latter circumstance is important when managing several interrupt-requesting devices: it is often necessary for an ISR to have exclusive access to certain resources in an initial stage of its execution. It would be impossible to guarantee it if the routine can be interrupted even before the first of its instructions has a chance to be executed, which would be the case when running with interrupts enabled in the PS register. On the other hand, a rather long ISR which does not share critical resources with another may choose to run with interrupts enabled, in order to avoid causing long delays in servicing more urgent interrupts.

The above 5-phase execution is easily achieved in the Secondary Decoder by the following extension of the table in fig 12.4:

Instruction	PD signal	Ph1	Ph2	Ph3	Ph4	Ph5
<code>ioi</code>	<code>ioi</code>	<code>SPact</code>	<code>SP2b0 data mem PC2b1 SPact</code>	<code>SP2b0 mem data PSR2b1</code>	<code>vec latchPC mem ld</code>	<code>vec odd ld mem LatchPSR</code>

There are some new signals there, which require an explanation.

**PSR2b1 and latchPSR** Both are triggers and they do what the mnemonics suggest: assert the content of the PS Register on bus 1 and latches the PS content (from bus 1 as well, nothing ever is latched from bus 0), respectively. Previously, the only role of the PS was to hold the flags. Those were accessed by the branch logic of the PD directly, without using the data path. Now that we require the PS register to be saved on the stack and restored afterwards, the easiest way to achieve this is by giving it the pair of assert/latch triggers that any other register in the Register File is equipped with. Yet the PS register still stays outside the data path since it belongs almost exclusively to the instruction machine.

<sup>5</sup>For the extended processor discussed in the next chapter, the sequence is even longer.

<sup>6</sup>In principle it is possible to use branches at the start of a subroutine and ALU instructions at the end of it to analyse and restore the flags, but it would be extremely inefficient to do so.

**vec** and **odd** Those are two selectors for the memory subsystem. Normally memory requires the address to be asserted on bus 0, but when **vec** is high then the address is produced by the memory system itself based on the **ver-adr** supplied by the IM (which latches it in its register from the **vector** bus in the process of the interrupt acknowledgement). If the signal **odd** is up, then the address produced is odd, i.e. that of the second byte of the vector pair.

Naturally we require something similar to **rts** except it should restore not only the PC content to what it was prior to the interrupt, but also the PS:

**Phase 1** read the top of the stack into the PSR; increment SP for stack pop

**Phase 2** read the top of the stack into the PC; increment SP for stack pop

We already reserved a place for a 0-operand instruction **rti** (ReTurn from Interrupt) in the Primary Decoder (section 12.5.2) to achieve that, and here is another extension of the Secondary Decoder that realises the above:

Instruction	PD signal	Ph1	Ph2
rti	rti	SP2b0 data mem ld latchPSR SPact up/dwn'	SP2b0 data mem ld latchPC SPact up/dwn'

Finally, the **Fetch** sequence must only take place when the Interrupt Master has not raised the **int** signal at the beginning of the clock cycle. Indeed if the signal is up, the IR is latching the virtual instruction internally and the PC should be left as is.

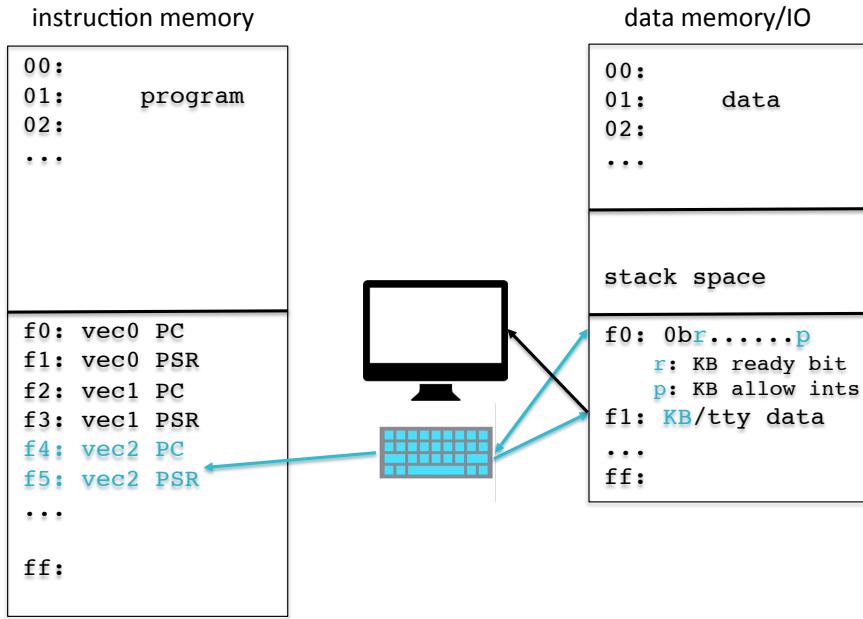
The **int** signal is produced by an RS flip-flop, which means that its negation is freely available, and we will label it **not-int**. We replace the **Fetch** line of the SD table in fig 12.4 by the following:

Instruction	PD signal	...	not-int
N/A	Fetch*	...	PC2b0 ld mem PCinc

### 13.5.3 A program example

Let us now take a look at how the interrupt aspect of a program is represented in the code. Below we present an application that works with two peripheral devices sharing a pair of I/O registers. The *KB status* register 0xf0 is used by the keyboard to indicate to the processor that the keyboard holds the ASCII code of a character that has been pressed by the user. This is indicated by the *ready flag*, bit 7, which is the sign bit. The program may choose to continuously test this flag until it is raised at which point the character code can be read from the *joint data* register 0xf1. Alternatively if the program writes to the status register raising bit 0, the interface is permitted to cause interrupts until the bit is reset by another write.

The joint data register can be written into, in which case the low 7 bits of the data byte are interpreted as an ASCII code and the corresponding character is displayed on the simulated teletype (tty) device. The teletype can print all printable ASCII characters and responds correctly to the newline character NL (ASCII code 0x0d).



The intention of the application is to receive from the keyboard newline-delimited, unsigned decimal numbers in character form, convert them to hex (modulo 256 if the number is greater) and print the result in character form on the tty. This could be accomplished, as we have done before, by repeatedly testing the ready flag. When the flag is up the program can safely read the next digit and apply the conversion procedure, but this would occupy the processor continuously with a kind of work that can and should be avoided.

Interrupts to the rescue. As mentioned before, bit 0 of the status register is the interrupt permission bit. When it is up, the keyboard interface requests an interrupt every time a key is pressed on the keyboard. When the interrupt is acknowledged, the interface signals vector 2 (vector address 0xf4) to the processor. A specially written ISR will pick up the key code and store it in what is known as a *cyclic buffer*, which is a convenient data structure that implements a queue of a fixed maximum size.

**The cyclic buffer.** Wikipedia has an article on cyclic buffers, which is recommended for further reading, but the principles of a cyclic buffer are simple enough for the reader to understand from the brief explanation below.

The buffer is implemented as a data structure comprising an array and two pointers (see figure 13.3). The pointers point to the head and the end of the queue. The end-pointer points to the array element that is free to use for putting an item on the queue. The head-pointer points to the next element to be processed. The pointers are advanced in a cyclical manner: the address is increased, but if this results in an address outside the array, the pointer is reset to the address of the very first element of the array. The array in the size of the cyclic buffer can profitably be chosen to be a power of 2. If that is the case, it is easy to see that cyclicity can be achieved by applying a bit mask after incrementing a pointer, provided that the buffer is aligned to a round enough address. Incidentally, a pointer can be backed up in the same way (decrement than mask).

Initially both pointers point to the same array element. Which element is not important, since the buffer is cyclic, but one could use the initial array element for certainty. Whenever the pointers point to the same element the buffer is, or has become empty.

In order to **put an item on the queue** the program must first check that the buffer has spare storage capacity. This is determined by checking that the advancement of the end pointer does not make it point to the same element as the head pointer. If the advancement results in both pointers having the same value, the buffer is considered full and the new item cannot be enqueued. Under such circumstances, either an error is signalled to the buffer's user, or the input item is simply ignored. If, on the other hand, the buffer has spare capacity, the new item is written into the element pointed to by the end-pointer and the end-pointer is advanced.

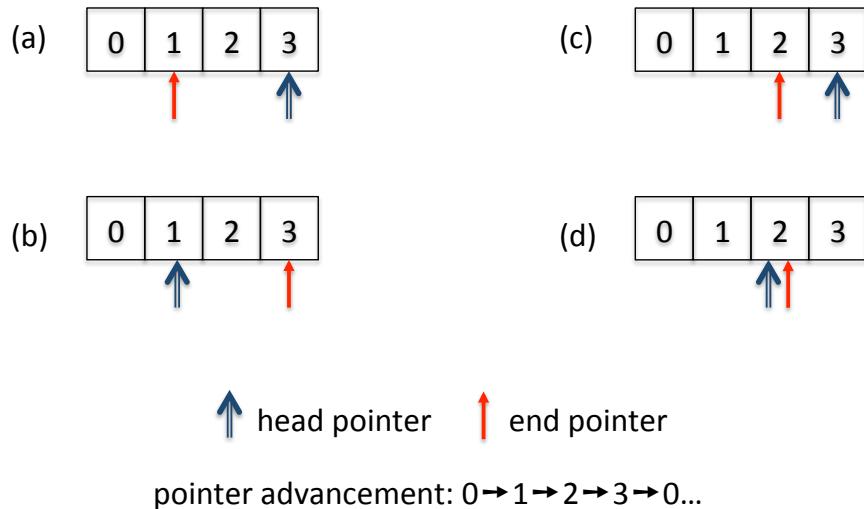


Figure 13.3: A cyclic buffer of 4 queue elements, of which any 3 can be used at any time. (a) The queue has two items on it, array elements 3 and 0, element 1 is currently free. (b) The queue has two items on it, array elements 1 and 2, element 3 is currently free. (c) The buffer is full: elements 3,0, and 1 have data to be read. (d) The queue is empty.

In order to **read an item off the queue** for processing, first the program checks that the buffer is not empty. If it is not, then the head item is read off using the head-pointer, and the head-pointer is advanced.

The reader may question the point of such a complex data structure in a simple matter of transferring keystroke codes byte-by-byte to the processing program. What is the intention of the queue between the data source (i.e. the interrupt service routine we will show below) and the data processing program?

The need for a queue lies in the timing of the events. The interrupts are not under the program's control, they may be requested at any time, which depends solely on the external agent, in this case the human pressing keys on the keyboard. The keyboard in our example has no buffering capacity: it holds only a single ASCII code of the latest keystroke. If the human presses a key again before the code has been processed, it will be overwritten with a new one, and the current key stroke will be lost. This makes it necessary for the platform to respond quickly to keystroke events, which limits the time that the processor may spend doing the work needed to process the key. Obviously if processing a keystroke takes much more time than it does on average, data may well be lost.

What we do to safeguard against data losses is simplify the interrupt service routine to reduce it to mere data acquisition. Once the key code has been read off the keyboard by the ISR, all the routine does is put it on a queue for processing. It is of course important that the ISR is not itself interrupted when it is doing its relatively small job, because if that were to happen, the ISR could not be fast enough to complete, besides the fact that the second character may overtake the first one on the way to the queue, invalidating the character sequence. That is why the ISR in our case must make sure that the interrupts are disabled until it has `rti`'ed.

It is important to understand that queues do not improve the processing capacity. In a hypothetical scenario of an extremely slow platform (our simulated one is one such) and a fast human typist, the latter will sooner or later fill up any space allocated for incoming characters, whether we use interrupts or flag testing in the program. Where interrupts and queues have a positive advantage is in a situation when the incoming stream of events is intermittent. At certain times several events may occur in an interval of some duration, and at other times the same interval may see no events happening at all, while on average the rate at which interrupts occur may be low enough for the processing to keep pace with it. In other words, queues, and our cyclic buffer is an implementation of a queue, have the ability to smooth over sudden peaks and troughs of incoming data without affecting the average and consequently the minimum performance requirements for the platform.

In this example we use a queue to demonstrate **asynchronous** behaviour: the human presses keys in her own time, while the program utilises the clock cycles left from the ISR for getting the data processed.

**The code** of the keyboard hex converted application is presented below. Only the essential part of it is included; the beginning and the end of the listing is omitted as it contains some OS instructions to be discussed in the next chapter.

**lines 13–15** define the interrupt vector consistent with the above.

**line 20** defines the name for the status I/O register. The data I/O register will be referred to as **I00+1**.

**lines 30–33** include the template definition that declares names of the three components of the cyclic buffer data structure.

**line 38** sets the stack below the I/O segment

**lines 40–44** initialise the cyclic buffer

**lines 48–50** set 0x01 in the KB status register to allow the keyboard to request interrupts.

**line 52** is an instruction (CdM-8 Platform 3<sup>1/2</sup>) that sets the processor status to 0x80. This is done via a system call, which is discussed in the next chapter.

**lines 56–57** print a prompt on the tty using the subroutine **print** coded in lines 196–205

**lines 60–135** constitute the main program.

**lines 61–69** are interesting. This is code for attempting to read the next character from the cyclic buffer.

If the buffer is empty, the processor enters a low-power-consumption state by issuing a Platform 2 instruction coded in assembler as **wait**. This results in the clock cycles being suspended until the next interrupt request, which completes the **wait** instruction. The while loop will then check if the buffer is empty and if it still is (e.g., when the interrupt was not from the KB device) will issue another **wait** etc.

**lines 70–135** represent a straightforward process of shifting the received decimal digit into **r3** that contains the current quantity the program is accumulating. The register is previously cleared (line 55), and the moment the newline is discovered in the cyclic buffer (represented as NULL) the accumulated quantity is printed on the tty using the **prtr1** subroutine found on lines 130–135. Then the main program prints a prompt and continues on the loop.

**lines 140–188** are our main interest. This is the Interrupt Service Routine, which is invoked automatically by the processor when it grants the keyboard an interrupt. The routine is sizeable, so it saves all registers (using **pushall**) and restores them (using **popall**) before the **rti** without affecting the speed of execution much. The code is straightforward and is carefully commented to ease the reading. It checks whether the cyclic buffer is full, and if not, puts the character read from the keyboard on the queue. If the buffer is full, it ignores the character and prints a message urging the typist to wait.

Here is the code.

```

13      asect 0xf4          # KB vector2 = 0xf4
f4: 6f      dc    KB_ISR  # KB interrupt service routine
f5: 00      dc    0        # PSR=0, interrupts disabled
16
17
18
19      asect 0xf0          # IOreg of KB/TTY interface
20  I00:           # I00: status reg:
21              #     bit7: got a char,
22              #     bit0: int on keypress
23              # I00+1: data register.
24              #     read: from KB
25              #     write: print to TTY
26
27 ##### Main program
28 # Plan data memory:

```

```

29  #
30      tplate q          # queue structure
00:  que:   ds    16      # space for queue
10:  hd:    ds    1       # head of queue for main prog
11:  nd:    ds    1       # end of queue for ISR
12:  34 ######
12:  35
13:  36      asect 0      # starting point
14:  37
00: cd f0 38      setsp 0xf0      # move the SP below the i/o page
15:  39
02: d0 00 40      ldi     r0,q.que    # r0->queue
04: d1 10 41      ldi     r1,q.hd # 
06: a4 42      st      r1,r0      # q.hd->q.que
07: d1 11 43      ldi     r1,q.nd #
09: a4 44      st      r1,r0      # q.nd->q.que
16: 45
17: 46      ##### the que is empty initially, hd=tl
0a: d0 f0 48      ldi r0,I00      # allow KB
0c: d1 01 49      ldi r1,1      #      to interrupt
0e: a1 50      st  r0,r1      #      from now on
18: 51
0f: c0 d0 80 db 52      setPS 0x80      # enable interrupts in PSR
13: 00 c4 53
15: 3f 54      clr r3      # keep the current number in r3
16: 55
16: d0 af 56      ldi r0,msg_dec
18: d6 9d 57      jsr print      # prompt for a number
19: 58
20: 59      ##### test the que and sleep if it is empty
21: start: 60
22: 61      while
23: 62      ldi r0,q.hd
24: b0 63      ld  r0,r0      # r0=hd
25: d1 11 64      ldi r1,q.nd
26: b5 65      ld  r1,r1      # r1=nd
27: 71 66      cmp r0,r1
28: e1 26 67      stays eq      # buffer empty
29: d5 68      wait
30: ee 1a 69      wend
31: 70
32: 71      ##### now not empty. read next char from queue
33: 72
34: b2 73      ld r0,r2      # r2==r0
35: 8c 74      inc r0
36: d1 0f 75      ldi r1,15
37: 44 76      and r1,r0      # r0= (r0+1) % 15
38: 77
39: d1 10 78      ldi r1,q.hd      # store hd pointer in memory
40: a4 79      st  r1,r0
41: 80
42: 81      ##### digit character in r2.
43: 82      # if not NULL, convert to bin and shift in r3:
44: 83      #           r3=r3*10+(r2-"0")
45: 84      # else print r3 in hex to TTY
46: 85

```

```

86         if
2e: 0a     87         tst r2
2f: e0 3d  88         is nz          # character != NULL
31: 0d     89         move r3,r1
32: 95     90         shla   r1
33: 95     91         shla   r1
34: 95     92         shla   r1
35: 97     93         shla   r3
36: 17     94         add    r1,r3      # r3=r3*8+r3*2=r3*10
37: d1 30  95         #
39: 39     96         ldi    r1,"0"
3a: 17     97         sub    r2,r1
3b: ee 55  98         add    r1,r3
3d: d0 b6  99         else           # character == NULL,
3f: d6 9d 100        # print r3
101        ldi r0,msg_hex
102        jsr print
103
41: 0e     104        move r3,r2
42: 9f     105        rol   r3
43: 9f     106        rol   r3
44: 9f     107        rol   r3
45: 9f     108        rol   r3      # high hex digit -> low hex digit
109
46: d1 0f  110        ldi   r1,0x0f
48: 4d     111        and   r3,r1
49: d6 67  112        jsr   prtr1      # print hi
113
4b: d1 0f  114        ldi   r1,0x0f
4d: 49     115        and   r2,r1
4e: d6 67  116        jsr   prtr1      # print lo
117
118
50: d0 af  119        ldi r0,msg_dec
52: d6 9d  120        jsr print
121
54: 3f     122        clr   r3
123
124        fi
125
55: ee 1a  126        br   start      # process the next character
127
57: 30 31 32 33 128 hextab: dc      "0123456789abcdef"
5b: 34 35 36 37
5f: 38 39 61 62
63: 63 64 65 66
129
67: d0 57  130 prtr1: ldi r0,hextab
69: 14     131         add r1,r0
6a: f0     132         ldc r0,r0
6b: d1 f1  133         ldi r1,I00+1
6d: a4     134         st  r1,r0
6e: d7     135         rts
136
137
138 ##### ISRs
139 #
140 KB_ISR:      # keyboard interrupt service routine

```

```

6f: ce          141      pushall
               142
               143
70: d0 f1      144      ldi r0,I00+1    # r0-> KB/TTY data reg
72: b2          145      ld  r0,r2        # read KB into r2
               146
               147
               148      if
73: d1 11      149      ldi r1,q.nd      #
75: b5          150      ld  r1,r1      # r1=nd
               151
76: 8d          152      inc r1
77: d0 0f      153      ldi r0,15
79: 41          154      and r0,r1      # r1 = (nd+1) % 15
               155
7a: d0 10      156      ldi r0,q.hd    #
7c: b0          157      ld  r0,r0      # r0=hd
               158
7d: 74          159      cmp r1,r0      # queue full?
7e: e0 97      160      is ne
               161      dec r1
               162      ldi r0,15
83: 41          163      and r0,r1      # r1 = ((nd+1) % 15 - 1) % 15 = nd
               164
84: d0 f1      165      ldi r0,I00+1  # KB/TTY data reg
86: a2          166      st  r0,r2      # echo the char on TTY
               167      if
87: d0 0a      168      ldi r0,0x0a    # new line (NL)
89: 78          169      cmp r2,r0
8a: e1 8d      170      is eq
               171      clr r2
               172      fi
8d: a6          173      st  r1,r2      # store character following nd pointer
               174
8e: 8d          175      inc r1
8f: d0 0f      176      ldi r0,15
91: 41          177      and r0,r1      # advance nd pointer: nd = (nd+1) % 15
               178
92: d0 11      179      ldi r0, q.nd
94: a1          180      st  r0,r1      # and store it in memory
95: ee 9b      181      else
97: d0 a8      182      ldi r0,msg_wt
99: d6 9d      183      jsr print
               184
               185      fi
               186
9b: cf          187      popall
9c: d9          188      rti
               189
               190      print: # dirty, uses r1 and r2!
9d: d1 f1      191      ldi r1,I00+1
               192      while
9f: f2          193      ldc r0,r2
a0: 8c          194      inc r0
a1: 0a          195      tst r2
a2: e0 a7      196      stays nz
a4: a6          197      st  r1,r2
a5: ee 9f      198      wend

```

```

a7: d7      199    rts
           200
           201
a8: 77 61 69 74 202 msg_wt:      dc      "wait!",0x0d,0
ac: 21 0d 00
af: 0d 64 65 63 203 msg_dec:    dc      0x0d,"dec: ",0
b3: 3a 20 00
b6: 68 65 78 3a 204 msg_hex:   dc      "hex: ",0
ba: 20 00

```

Now let us see how the required interfaces can be implemented in hardware.

### 13.5.4 Four-phase handshake

The key implementation concept of the interrupt subsystem is that of *handshake*. There are two actors engaging in the interrupt: the device requesting it and the processor granting it. Each can wait for the other to become ready to engage and each needs to know that the other *has* engaged. The device needs to know that the interrupt has been granted in order to assert the vector on the vector bus and thus identify itself to the processor. The processor needs to know that the device has detected that the interrupt has been granted and so the vector bus has valid data to be latched in order to start the interrupt sequence. The process of achieving this mutual assurance is called handshaking and the state thus reached is one of handshake.

There are more than one known protocol whereby handshake may be achieved, and the CdM-8 processor implements the simplest and most reliable one, the four-phase handshaking protocol, which is presented in figure 13.4. Two signals are involved: **IRQ**, the interrupt request line from device to processor, and **IAck**, the interrupt acknowledge line from processor to device. Both lines are kept down at zero by their respective drivers, the device and the processor, before the handshake. There is also a 3-bit vector bus that the processor reads from at times, and on which the device can assert; before the handshake this bus floats.

Here is the protocol laid out as a sequence of four phases<sup>7</sup>:

1. When the device needs an interrupt, it raises **IRQ** and starts watching **IAck**. The processor is monitoring **IRQ** and when it sees it rise it waits for the conditions inside the processor to becomes favourable for granting the request and then grants it by raising **IAck**. It then continues to watch **IRQ**.
2. Now **IAck** is up and the device has noticed it. It immediately asserts its vector (the *R* number mentioned earlier) on the 3-wire vector bus. Then after a little delay to make sure that the vector is fully asserted on the bus, the device drops **IRQ** and continues to watch **IAck**.
3. The processor has been watching **IRQ**, which has now dropped and the processor has noticed it. It immediately drops **IAck** and in the process of doing so latches the vector value in its dedicated *vector* register.
4. Meanwhile the device has been watching **IAck** and it has noticed that it has dropped. It immediately disconnects itself from the vector bus and now both signals are down and the vector bus floats — we are back where we started and the device is free to initiate the next interrupt at any later time.

Let us take a look at the implementation of the KB/tty interface for the program presented in the previous section. Starting from the east end of the circuit diagram, we can see the already familiar memory-mapped I/O signals **In/out'**, **I/Odat**, **I/Osel** and **I/Oadr**, which are supported by the CdM-8 Harvard architecture platform presented in section 13.4.2. The access to the I/O registers **I00** and **I00+1** is decoded by the north-west AND-gate which enables the decoder to its immediate south which derives four triggers (outputs 0-3) in this order: KB status write, KB status read, tty data write, KB data read. The “read” triggers drive controlled buffers that assert the correponding data on the **I/Odat** lines. That data in the case of the status register content is composed of bit 0, coming from the 1-bit register in the south west corner which holds the interrupt-enabled status of the keyboard, and bit 7, coming from the south side of the simulated keyboard where an output pin indicates whether the keyboard holds an unread keystroke. The rest of the bits are

---

<sup>7</sup>for the avoidance of doubt, these have nothing to do with the processor Sequencer, nor they depend on a common clock in any way. The phases are stages of the protocol.

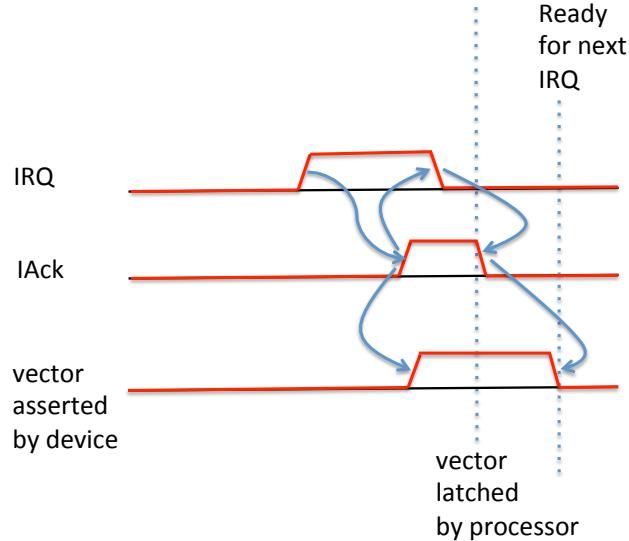


Figure 13.4: 4-phase handshaking protocol

grounded. The data from the data register comes straight from inside the keyboard (which is where the data register is physically located). The “write” triggers go to the aforementioned 1-bit register to cause it to latch the incoming interrupt status from bit 0 of `I/Odat` and to the “clock” input of the `tty` causing it to display the character asserted on its data pin (connected straight to `I/Odat`), respectively.

Our main interest being interrupts, let us direct our attention to the north east part, where the relevant part of the interface is laid out. Two edge converters drive an RS flip-flop. The moment the keyboard latches a key code and the interrupts are or become allowed, the rising edge converter sets the RS flip-flop, which raises `IRQ`. The rising edge of the `IAck` signal causes its edge convertor to reset the RS flip-flop, thus completing the four-phase protocol. Since in this example there is only one interrupt-enabled device, the vector bus is permanently wired to the vector, but it would take next to no effort to introduce a controlled buffer asserting the constant 0x2 on the vector bus only on high `IAck`.

This completes the KB/tty example both on the software and on the hardware side. What is still left unclear is the implementation of interrupt signalling in the processor itself. We will briefly consider this next.

### 13.5.5 Interrupt Master

We have all the necessary circuitry in the processor already to support the interrupt sequence: the saving of the PC and PS contents on the stack, the fetching of the vector (although we did not see the circuitry for that, we have met the signals `vec` and `odd` in section 13.5.2), and the latching of the new PC and PS. We also have a full implementation of the return instruction `rti`. What is missing in the picture is the unit that negotiates interrupts with devices, correctly grants them an interrupt, and which raises the `int` signal when the four-phase protocol has been successfully run to completion. That is the job of the Interrupt Master, which we mentioned earlier, and in this section we are going to present a hardware solution, which is interesting due to the fact that the circuit is semi-timed: some of the events are derived from the clock and the rest are not. Here is the circuit diagram:

## I/O bus

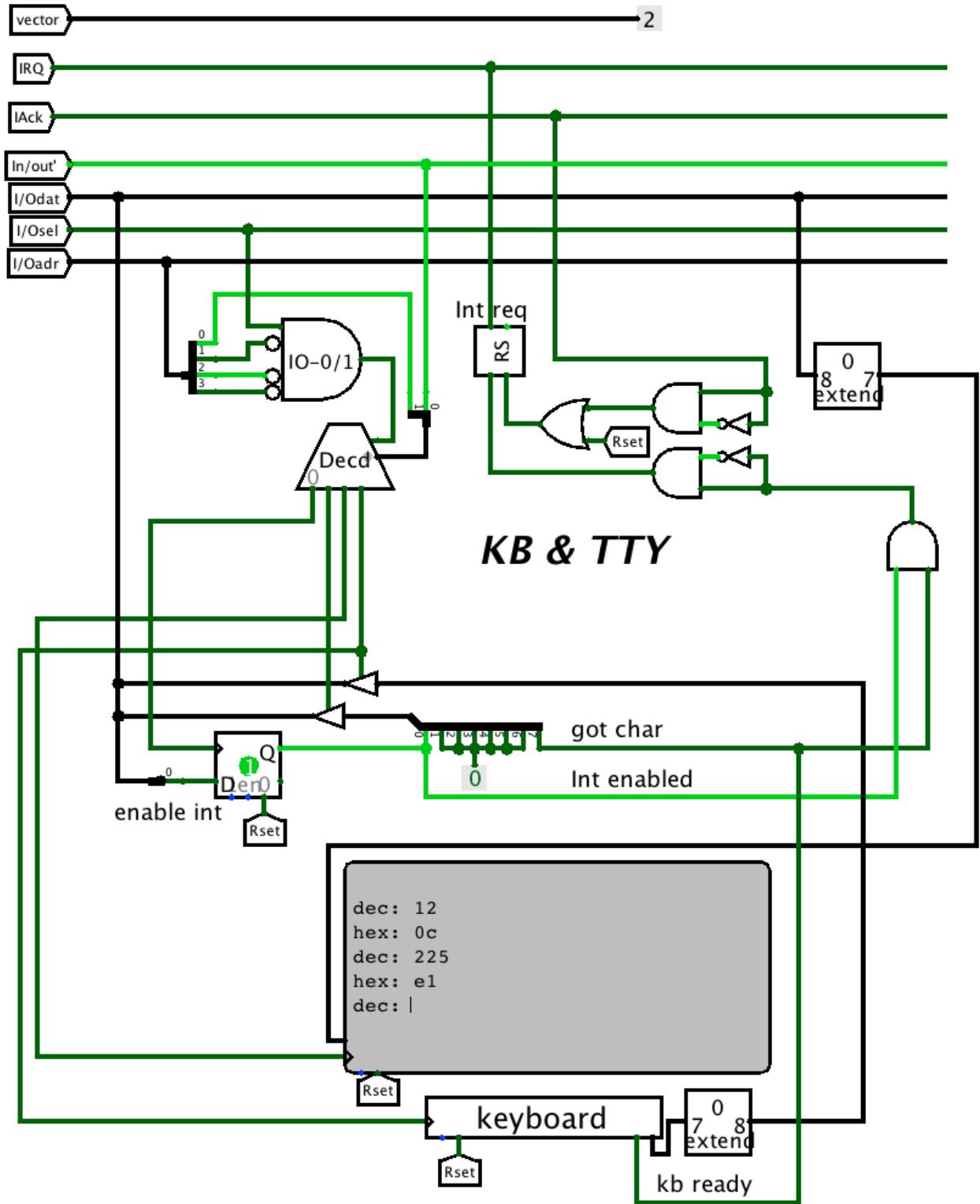
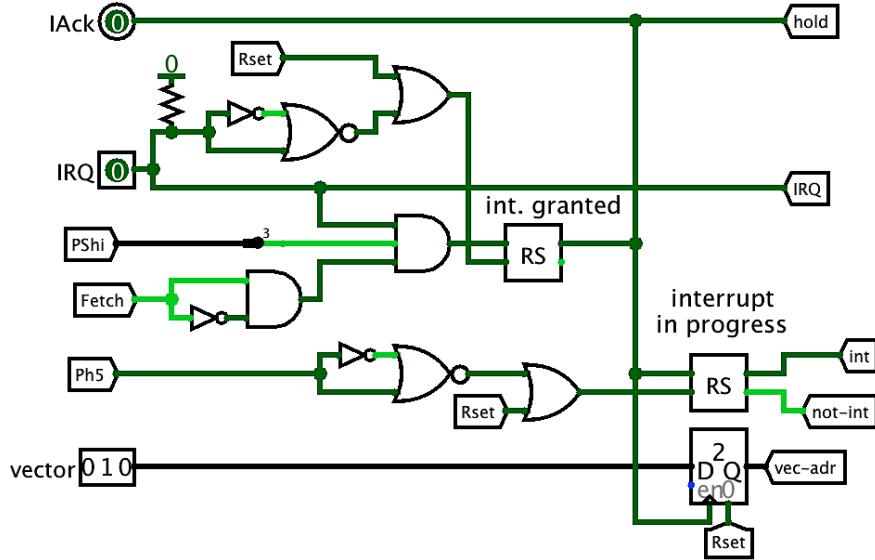


Figure 13.5: Keyboard and tty interface for the program of section 13.5.3



The **IRQ** signal comes to the IM straight from the eponymous processor pin. Its high level sets the first RS flip-flop (west of the vertical wire) which is an element that holds the “interrupt granted” state. The RS flip-flop is only set when bit 3 of the most significant 4 bits of the PSR, the bunch **PShi**, i.e. bit 7 of the PS register is up. That bit is the interrupt-enabled bit of the Processor Status. Another precondition for the interrupt to be granted is that the Sequencer is in the **Fetch** phase. Indeed, in any other phase there would be an instruction whose execution was in progress. Interrupting that instruction would not be possible without losing its effects since we have no means of saving the state of the Sequencer and various other data holders around the data path. We can only influence the order in which whole instructions are executed. That is why the set pulse for the RS-flip flop is actually derived from the rising edge of the **Fetch** signal. The **IRQ** enables the pulse to propagate across the AND-gate to reach the RS flip-flop and set it.

The falling edge of the **IRQ** resets the flip-flop (see the north part of the diagram). The output of the flip-flop is asserted on the processor’s **IAck** pin which has the following effect. As soon as the **IRQ** raises *and* the conditions are favourable for granting an interrupt, **IAck** goes up and as soon as the **IRQ** has been sensed to have gone down, the **IAck** signal goes down as well. This implements two causation arrows of the four-phase handshake in figure 13.4. The other three arrows in the figure are implemented by the interrupting device. The 3-bit register at the bottom of the above circuit diagram is clocked by the **IAck** and it latches the interrupt vector on the falling edge of this signal. It is, consequently, the **vector** register

Notice that the IM produces the **hold** signal for internal use, and that this signal is synonymous with **IAck**. The purpose of it is as follows. The IM is an unclocked device as far as the outside of the processor is concerned. It has no influence on peripheral devices. In particular, it cannot be guaranteed that the device response (the falling edge of **IRQ**) will follow the rise of **IAck** *within a single clock cycle*. The device may need to prepare for data transfer, or the wire connection to the device interface may be too long for it to charge up and down fast enough<sup>8</sup>. The **hold** signal prevents the Sequencer from progressing to the next phase; it keeps the **Fetch** signal high until the protocol has been completed by the peripheral device. This is achieved in a straightforward manner, by feeding the inverse of it to the phase register’s **enable** input.

The rising edge of *interrupt granted* sets the second RS flip-flop, whose meaning is *interrupt in progress*. This is to do with the synchronous (clocked) part of processing the interrupt vector. The output of this flip-flop is the main output signal, **int**, of the IM. It keeps high until the end of phase 5 when the Secondary Decoder has finished triggering the data pass along the interrupt sequence. During that time the signal **int** is used as a selector to enable the use of the vector now latched in the vector register.

### 13.5.6 Multiple interrupt sources and bus arbitration<sup>9</sup>

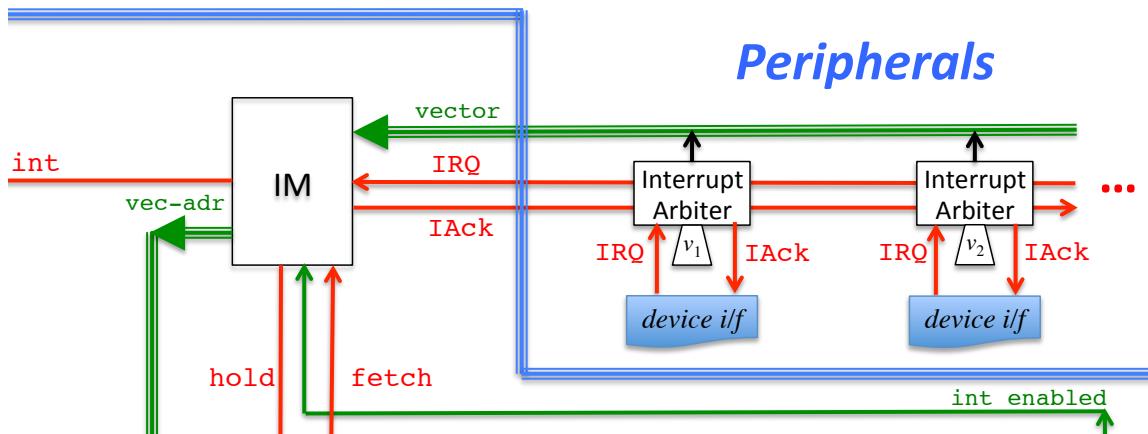
Our final concern in the study of interrupts is to do with the fact that the processor sees the outside world through a single hole: there is only one I/O bus for memory-mapped data exchanges, and there is only

<sup>8</sup>This is the reason why we have a request-response solution in the first place, rather than just using a *common* clock: a common clock would in most cases be too slow for the processor circuitry, effectively reducing its speed to the speed of the slowest peripheral device

<sup>9</sup>Skip this section at first reading

a single pair of IRQ/IAck signals. Given that interacting with a single interrupt-enabled device requires a four-phase protocol, how can a processor be interfaced with several such devices provided that they are completely independent and do not, by themselves, coordinate their interrupt requests?

The solution we have mentioned before, an Interrupt Arbiter (IA) is interesting for the fact that it is *completely asynchronous*, i.e. has no clock of any kind yet uses sequential logic. Let us return to figure 13.2 and recall what an IA is supposed to do. Here is the relevant part of the figure again:



When its client (connected to the south side of the device) is quiescent, the IA bypasses the IRQ signal east-to-west and the IAck signal west-to-east. The northern port of the Arbiter is effectively disconnected from the vector bus. As soon as the IRQ on the south rises, the arbiter starts to watch its east-to-west and west-to-east signals while still passing them:

1. next, as soon as IAck goes down (if it is not down already) on the west it prevents it from propagating west-to-east and holds the east IAck down;
2. next, it connects the south IRQ to the west (which raises the latter if it is not high already), prevents the IRQ from the east from propagating, and connects the west IAck to the south. Then it waits for both signals to go down. As soon as it happens, the IA detects that the client is quiescent and it switches back to bypass mode.

As a side activity, the IA also asserts a constant vector value read off the middle pin on the south side on the output pin on the north side whenever it sees that the client's IAck is up. This could be done by the client circuit itself, but it seems convenient to place that controlled buffer inside the IA so that arbiter chips could be distinguished from one another by their  $R$  constants.

A circuit implementing the above behaviour is shown in figure 13.6. The west side (up to the vector connection) contains a tiny automaton with a 2-bit state held in the register and triggered by signals coming from the east side of the circuit. After the initial reset, the register holds 0. Every time there is a rising edge on the clock input of the register, its value is incremented, but when it reaches 2, the next value is 0:  $0 \rightarrow 1 \rightarrow 2 \rightarrow 0 \rightarrow 1 \rightarrow \dots$ , thanks to the two AND-gates in the register's feedback. Those numbers have the meaning of *arbitration state*.

The decoder selects one of the three combinations realised as AND-gates on the east side depending on the state: S0,S1 or S2. Each AND-gate is only active when selected and then it defines a combination of the input signals that cause a change of the arbitration state.

We can now discuss how arbitration works.

**In state 0** the unit bypasses the signals east to west and west to east thanks to the IRQ signal path at the top and the *zero-trap* structure in the south-east corner. Let us dwell a little on the latter:

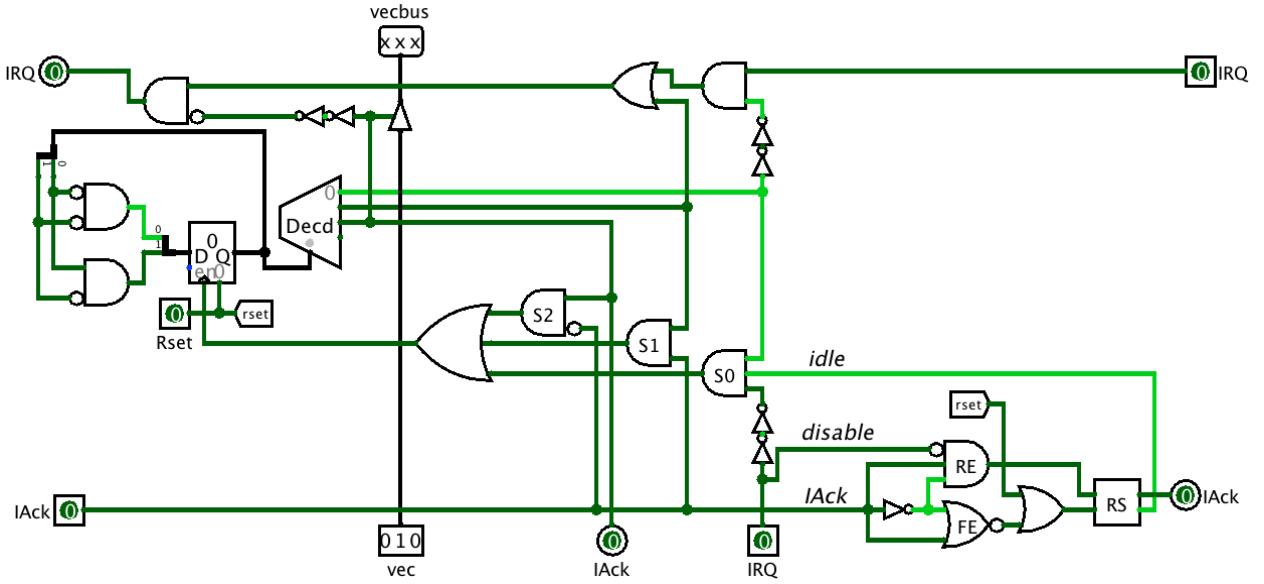
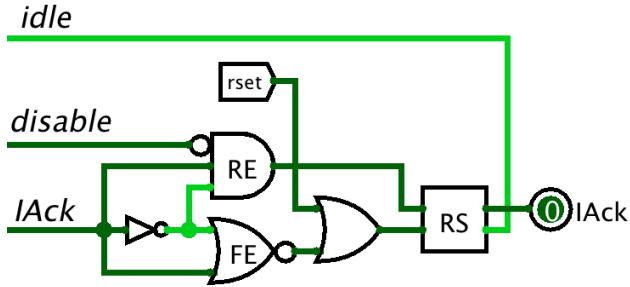


Figure 13.6: Interrupt Arbiter



When the *disable* signal (the south, i.e. client, IRQ) is down, the gates RE and FE represent a rising-and falling-edge-to-pulse converters, respectively. Those are fed to an RS flip-flop, which will be set and reset by the corresponding edges, thus acting as a follower of the west IAck signal. This will have the effect of propagating the west IAck eastwards with some (unimportant) delay. The situation changes dramatically when the *disable* signal goes up. This could be before a rising edge of the west IAck or after. In both cases the falling edge of the *enable* signal stops the production of pulses in response to the coming rising edges of the west IAck. A *falling* edge of the west IAck would still trigger the falling-edge converter FE and reset the flip-flop, but since rising edges are suppressed from this time on, the Q output of the flip-flop will, either immediately or later, be, or come to, 0 and will *stay* zero as long as the *disable* signal remains high. What this does is ensure that the chain east of the arbiter can *complete* its 4-phase protocol, but that it will be prevented from initiating another session. The quiescence of that chain is what the signal *idle* signifies.

Back to figure 13.6, and we conclude that in state S0 the automaton waits for the chain east of the arbiter to become idle *after* the client IRQ has gone up. The double-inverter delay is inserted before the S0 gate to ensure that the rising edge of the client IRQ does not affect S0 *until* its effect on the flip-flop via RE has fully propagated. That gives the automaton the much needed assurance that the transition to state 1 only happens when the east is fully isolated from the west IAck. Without the assurance it is possible in principle that the chain east of the arbiter sees the rising edge of IAck *after* the automaton has transitioned to state 1, with a disastrous effect on the vector bus (both the chain and the current arbiter asserting different R numbers causing signal conflict and an arbitrary value latched in the vector register of the Interrupt Master). The actual value of the delay needed at the south input of the S0 gate depends on technology and is usually established by simulation. It is very instructive though to see the kind of issues an asynchronous design, such as ours, gives rise to. We recall that we had no such problems in our design of the processor's engine since any propagation delays, as long as they were reliably shorter than the high-clock period, were of no concern. However,

the minimum duration of the clock cycle, and the speed of computing with it, are directly affected by such delays, so the issue of timing is never too far away, whether the circuit is clocked or not, as long as we remain at Level 2 of the platform hierarchy.

The rising edge of *idle* (or the client *IRQ* if idle is up already) will send the automaton to state 1. The two inverters above the S0 gate are there to ensure that between the times at which state 0 is left and state 1 is entered into, the west *IRQ* continues to follow the east one. This is essential because if the east *IRQ* is high at the time that the client *IRQ* becomes high, a switch from state 0 to state 1 may introduce a drop and a rise thus inadvertently signalling to the processor to advance the 4-phase protocol.

**In state 1** the west *IRQ* is high thanks to the OR-gate fed into by output 1 of the decoder. The output of the gate passes through the final AND-gate due to the fact that its other input comes inverted from the decoder's output 2, which is down in state 1. We conclude that in state 1 the west *IRQ* is up — the client's *IRQ* has been propagated westward. The automaton remains in state 1 until the processor's IM responds with a high *IAck* which energises the gate S1 and causes a transition to state 2.

**In state 2** the arbiter assumes that the client has been granted an interrupt. Consequently it asserts the interrupt vector, the *R* number *vec*, on *vecbus*, and, after a double-inverter delay which guarantees that the vector has been asserted, drops the west *IRQ* thus causing the latching by the IM of the vector value. The IM responds with the drop of western *IAck*, which energises the gate S2 via its inverted south input and causes the automaton to return to state 0.

### 13.5.7 Multiple interrupt sources: an example

As a summary and also to compensate the reader for skipping the previous section at first reading, we restate that the arbiter makes it possible to daisy chain interrupt sources and connect them to a single set of interrupt controls on the west side of the CdM-8 processor. From now on, we will not concern ourselves with matters of timing and implementation in general, we just assume that if a device interface requires the ability to cause interrupts, it will provide sufficient circuitry to engage in the 4-phase protocol and that this circuit can either be connected to the west side of the processor directly or via an arbiter as shown in the previous section. The former is done when only one peripheral device causes interrupts, and the latter when there are more than one. We will remember about the priority of interrupts (the closer to the processor on the daisy chain, the higher) but not much else as we discuss a multiple interrupt source application next.

The so-called *interval timer* is a very useful device often (if not always) found in a practical Platform 2<sup>One</sup>. One possible implementation of it is presented in figure 13.7. Its purpose is to measure a certain time interval and inform the program that it has elapsed. The interval is expressed in “ticks” which are some fixed time units. In our implementation we equated ticks with clock cycles for simplicity. The diagram in the figure shows a clock connected to the clock input of a register, but this is *not* the processor's main clock, since the timer is located outside the processor. Instead this is an independent clock, which should be assumed to be out of synch with the main one<sup>10</sup>.

At the core of the interval timer is a rising-edge-triggered register that holds the number of ticks to the end of the interval, see the east part of the diagram. The register acts as a down counter. Its feedback network consists of a decrementer (built, for simplicity, from library components and comprising an adder with the constant  $-1$ ) and a multiplexer that selects 0 when the result is negative. The other mux, which feeds directly into the register D-input, selects the decremented value or the content of the I/O register, which is written into from the I/O bus's *I/Odata* bunch. The apparent complexity of the rest of the circuit is to do with the fact that the timer is an asynchronous device (and by now we *expect* complexity if we cannot simply rely on the clock to command the necessary order of the events). Consequently, the processor cannot safely write into the timer's register since that can collide with its update via the feedback loop. Instead the processor writes into a separate I/O register, which is the source of a new interval setting. That is done conventionally: via decoding the I/O bus signals using the gate IO-2 and its attendant decoder. They raise output 0 whenever the timer's I/O register is written into by the processor, or output 1 when the processor reads from the I/O register. The former, as was mentioned, is required for setting a new time interval and the latter can be useful if the program needs to know how much time remains until the end of the interval.

Next to the decoder there is an RS flip-flop, whose purpose is to control the mux that selects the input for the timer register. Once the program has set a new value in the I/O register, the RS is set (after a

---

<sup>10</sup>Logisim does not support multiple independent clocks, so this aspect of the interval timer is not simulated adequately.

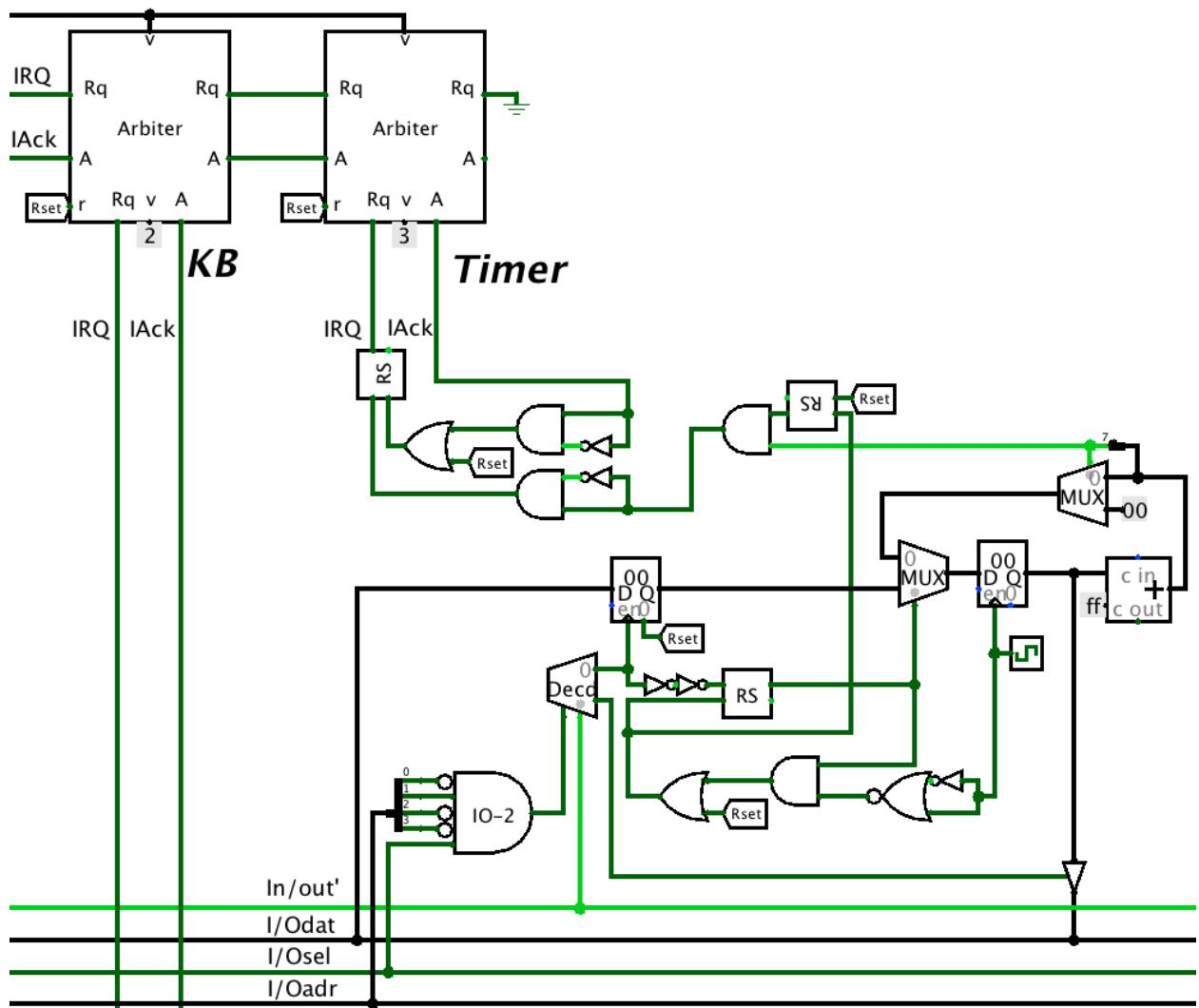


Figure 13.7: Interval timer on a daisy chain connected after the keyboard

delay allowing for the new value to be latched in the I/O register first). Since the timer register is triggered by the rising edge, we use the falling edge to trigger the return of the mux selector back to 0 to continue decrementing the interval after the new value has been copied from the I/O register.

The same signal (the falling edge of the clock following the setting of a new value in the timer register) is used to allow the interrupt part of the interface to activate. The problem is that the timer has content 0 initially, as well as after it has counted down to zero. In the former case no interrupt should be raised (since no timing has been done yet). Any subsequent coming to 0 should raise an interrupt. So the interrupt part is not active initially, which is achieved by wiring the reset input of the RS flip-flop there (north-west of the counter network) to system reset. After the first update of the interval timer, the RS is set and its output is AND-ed with the sign bit of the result, giving a rising edge; this goes through a converter to set the interrupt RS driving the arbiter. That RS is reset by the rising edge of IAck received from the arbiter. The device uses vector 3 (0xf6). The timer's arbiter sits behind the keyboard one, which makes the interval timer lower priority than the keyboard. This is acceptable if we wish to interpret the interrupt as an event signifying that a certain **minimum** time has elapsed, while still bearing in mind that any delay in reading the keyboard may result in loss of data, which is unacceptable.

Let us demonstrate how the timer and the keyboard/tty work together in a single application. The program segment below repeatedly sends a character in r2, initialised to "x" (lines 53–54) to the tty (lines 55–57) while keeping the interrupts enabled (line 47) and permitted to the KB (lines 43–45). The keyboard interrupt routine (lines 61–66) reads the ASCII code of the latest key press into r2 to change the character sent to the tty, and the timer interrupt routine (lines 68–77) resets the timer to 120 ticks and sets the current character to "\$".

```

f0: 2b      10      dc      OS_entry          #Â OS int routine
f1: 00      11      dc      0x00             # PSR, bit 0x00 enables interrupts
f2:          12
f3:          13      asect 0xf4          # KB vector2 = 0xf4
f4: 19      14      dc      KB_ISR           # KB interrupt service routine
f5: 00      15      dc      0                # PSR=0, interrupts disabled
f6:          16
f7: 1f      17      asect 0xf6          # timer vector3 = 0xf6
f8:          18      dc      TMR_ISR           # timer interrupt service routine
f9: 00      19      dc      0                # PSR=0, interrupts disabled
fA:          20
fB:          21
fC:          22      asect 0xf0          # IOreg of KB/TTY interface
fD: I00:          23      # I00: status reg:
fE:          24      #       bit7: got a char,
fF:          25      #       bit0: cause interrupts on key presses
fA:          26      # I00+1: data register. Write to it
fB:          27      #       to print to screen
fC:          28      # if program reads I00+1, it gets
fD:          29      #       last keypress in ASCII
fE:          30
fF:          31      asect 0xf2          # IOreg for the interval timer
f0: I02:          32      # writing a nonzero to I02 starts the timer
f1:          33      # every "tick" timer gets decremented
f2:          34      # when I02 content =0, timer requests interrupt
f3:          35
f4:          36
f5:          37 ##### Main program
f6:          38      #
f7:          39      asect 0          # starting point
f8:          40
00: cd f0      41      setsp 0xf0          # move the SP below the i/o page
f9:          42
02: d1 01      43      ldi r1,1          # allow KB to interrupt
04: d0 f0      44      ldi r0,I00
06: a1         45      st   r0,r1

```

```

        46
07: c0 d0 80 db 47      setPS    0x80          # enable interrupts in PSR
0b: 00 c4

        48
0d: d1 78    49      ldi r1,120       # give the timer its initial value >0,
0f: d0 f2    50      ldi r0,I00+2   # this will start it
11: a1      51      st  r0,r1
52
12: d0 f1    53      ldi r0,I00+1   # KB/TTY data reg
14: d2 78    54      ldi r2,"x"
55  loop:
16: a2      56      st  r0,r2
17: ee 16    57      br  loop
58
59 ##### ISRs
60 #
61 KB_ISR:
19: c0      62      push r0        # save r0
1a: d0 f1    63      ldi r0,I00+1   # r0-> KB/TTY data reg
1c: b2      64      ld  r0,r2       # read KB into r2
1d: c4      65      pop r0        # restore r0
1e: d9      66      rti
67
68 TMR_ISR:
1f: c0      69      push r0
20: c1      70      push r1
21: d0 f2    71      ldi r0,I00+2
23: d1 78    72      ldi r1,120
25: a1      73      st  r0,r1
26: c5      74      pop r1
27: c4      75      pop r0
28: d2 24    76      ldi r2,"$"     # set current char to "$"
2a: d9      77      rti

```

While running this program on a simulator one can observe that the timer can be blocked by actively typing text on the keyboard, which leads to the timer interrupts being delayed on the daisy chain. The IRQ's from the timer interface simply cannot get the IAck they need due to the fact that the keyboard ISR preempts the processor over and over again. However the more striking feature of this demonstration is the fact that it feels like two applications running concurrently: the update of the timer and the character update from the keyboard, which is due to the fact that the events in the program are closely interleaved. We allow this feeling to linger on, since it is precisely the illusion that motivated a whole area of technology, known as Operating Systems. An introduction to it is our final topic as we arrive to the destination of this long journey across computing platforms.

# Full Platform 2: Operating System

## 14.1 Processes

In the last chapter we familiarised ourselves with how a processor may be connected to peripheral devices and how their interaction may be programmed. In doing so we saw for the first time that the timing of events inside a computer may not necessarily be synchronised (i.e. juxtaposed in time) with the edges of the processor's internal clock. We came across the notion of interrupt, which is central to modern computer organisation. The interrupt is in fact the only *primary* event that may occur out of sync with the clock; any other such event, and there are many that occur at software levels supported by the hardware that constitutes Platform 2, is derived from an interrupt of some kind.

Closely related to the notion of interrupt is the notion of *process*.

### Definition 20 : Process

**A process** is a data record that comprises the contents of all processor registers visible to a program: SP, PS, PC and all the GP registers. A process has associated memory space for saving those and other changeable values, called its *private data*.

**Active process.** A process is said to be *active* if its record is loaded in the corresponding hardware registers. Otherwise it is *suspended*, and has to be stored in its private data like any other changeable value. Only one process can be active at any given time on a single-processor platform, and a process's record, being stored in *its* private data, cannot change while the process is suspended. A previously suspended process may be *resumed*, i.e. made active, and then suspended again, etc.

**Progression.** The processor always executes the next instruction (the one pointed to by the PC content) of the active process; as a result the process is *progressed*, i.e. moved towards *termination*. Accordingly, it changes and becomes a new active process. This may also affect its own private data but, for an ordinary process, must not overwrite anything else.

**Creation.** From the process progression point of view, an interrupt is seen as an event leading to the current active process being suspended and a new process being *created* and made active immediately. The termination of an active process does also lead to a creation or resumption of another, see below.

**Context switch.** Suspending the active process and either creating a new active one or *resuming* a suspended process is known as *context switch*, and it requires more than the hardware interrupt logic to shift full register records in and out, see also fig 14.1. Software, too, may be involved in choosing which process to resume or create.

**Termination.** Just as any data value, a process can be discarded. To be discarded, it has to first become suspended, and then the memory space for it is reclaimed, for example, by not storing the process's record anywhere. Since the processor must execute a process at any time, this usually involves an `rti` instruction resuming a previously suspended process. By contrast a program running on a single-process platform can only terminate by issuing a `halt` instruction and when it does, everything stops, including the platform itself.

Let us now list some of the corollaries of the above definition.

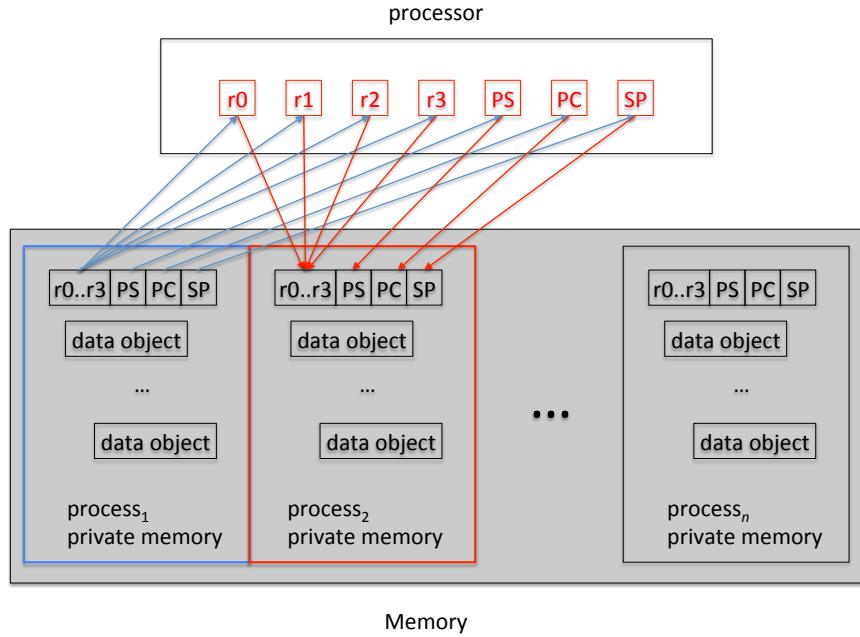


Figure 14.1: Context switch and memory partitioning. In the figure: first, process 2 becomes suspended and then process 1 is activated.

**Principle of Transparency.** The key property following from the definition is one of transparency of context switch. Imagine a process being suspended for whatever reason, and later resumed. Will the program that the process progresses “notice” this? In other words, will any of the results that the program produces change? The answer is no: the content of every register the program may depend on, which includes both general purpose registers utilised by instructions directly, and the special purpose registers PS, SP and PC, which affect program execution indirectly (through branches and stack operations), will be stored in the process’s private data (see the first bullet above) and then moved back into the hardware register without change (see the second bullet). The rest of the private data cannot change either: the data is private to the process in question and only it has access to it, and only when it is not suspended. Consequently, no matter how often a process is context switched in and out of the processor, it will compute *exactly* the same result as if it were never interrupted. Only the *wall-clock time* to termination depends on the frequency of context switches and on the duration of the suspension periods. We will call this observation the *Principle of Transparency* and will require a correctly engineered multiprocess platform to satisfy it.

**Concurrency.** So far we have only seen a platform run a single program from reset to halt, but the digital world is richer than that. A commercial desktop computer may run tens of applications “at the same time” according to the experience of the typical user; the reader may at this point begin to wonder whether our example platform, Cdm-8, is too primitive to be able to see this everyday magic in our demonstrations. As all other magic, this one is nothing more than a convincing illusion, and our platform is capable of fully supporting it as well. Indeed, what the user of a multiprocess platform sees is not a platform running many applications at the same time. Unless there is more than one processor available, and such *multicore* systems are beyond the scope of this book, what the platform does is context-switch between processes in rapid succession. Just like a video series consisting of discrete frames creates an appearance of continuous motion, so high-frequency switch between processes makes them look as if all of them were progressing at the same time but at a fraction of the processor’s clock rate. From that point of view, it is not even important how many processors, one or more than one, are available on a platform as long as their number is smaller than the number of processes that they must progress: either way context switch and its attendant illusion of simultaneity are unavoidable.

**Partitioned memory.** In the absence of interrupts there is obviously a single program and a single process, which is active: there is some content in each of the registers after reset, and there is no way that the process

can become suspended before it (and the platform with it) halts. In the presence of interrupts, processes represent individual “programs” that share the processor. That gives rise to the next question: what happens to the process (or program) *data* stored in memory? How could memory be used collectively?

While the processor, including all of its registers and clock cycles, is shared, for obvious reasons each process must have its private data space<sup>1</sup>. Consequently, in a multi-process system the memory is partitioned into processes’ private areas where mutable data may be held throughout the process’s execution. Part of the private space is reserved for storing the process’s record during suspension periods, the rest is used by the code to store data of any kind, see fig 14.1. The diagram should not mislead the reader into thinking that a process’s private memory has to occupy a contiguous segment of memory. It does not. In fact process records of all suspended processes are usually collated into a single *process table*, which also includes information required to decide on the context switch following an event (an interrupt or a system call). Most of a process’s private memory would, however, be contiguous for reasons of safety and security.

**Safety and security.** In a *multi-user* platform (such exist, but, alas, we must not go there in this introductory volume) processes’ private data must be protected from each other in case the code misbehaves and overwrites something that does not belong to it: after all, there is a single address space for all, and nothing would normally stop a program from writing to any memory cell. Furthermore, in a multi-user system there is a pressing need to *secure* private space of each process, possibly including the code, since a process may utilise malicious code that actively spies on others. We must limit ourselves to the very minimum of Platform 2 behaviour here, which includes none of those interesting issues, each worthy of a book of its own. So we will consider some of the unavoidable measures of safety, but none of security.

**Need for an OS.** All safety measures, and some of the security ones, boil down to the issue of memory virtualisation. What we want a program executed by a process *not* to be able to do is address other processes’ space or any I/O devices *directly*. Processes may legitimately communicate with each other and so can a process with an I/O device, but for ensuring safety and coordination a trusted intermediary is required. Such an intermediary is needed for context switching, too: after all, processes have no awareness of other processes’ needs or rights in regard of the processor. Both requirements are fulfilled by the same agent: the *operating system*, or OS for short, which happens to be yet another process or rather a set of processes that assist and oversee the execution of user processes on the platform.

The OS processes are never suspended and are only active for a short time before termination. It follows that only one OS process can ever exist at any given time. It is created when a service is required, either by a user process or by a peripheral device via an interrupt. Such a process tends to be unrestrained in its access to the hardware part of Platform 2, of which the OS itself is the software part. Together the OS and the processor/memory/peripherals appear to a user program as a single virtual machine that has the instruction set of the real processor *augmented* with *software* extensions known as *system calls*. Those are somewhat similar to subroutine calls but may result in context switch which is why they are implemented as *synchronous* interrupts, i.e. ones that occur following the fetch-and-decode of a special machine instruction rather than an external, asynchronous interrupt signal.

Recall that Chapter 2 started off with the Tannenbaum diagram in section 2.1, where Platform 2 was shown as a combination of software and hardware resources. Until now we have been unable to shed light on this last remaining missing link. We shall proceed with it in this chapter, but in doing so we will rely on the already learned correspondence between Platform 2 and Platform 3½ and define and use macros in order to present the part of the full Platform 2 instruction set implemented via system calls.

Any hardware, however, must come first. In order to appreciate the functionality of a full Platform 2, we require more hardware resources than the tiny circuit that we have been satisfied with so far can provide. Since processes’ private memory has to be partitioned, each process’s partition must be rendered inaccessible by other processes for reasons of safety, and finally, since the amount of memory space an address-consistent 8-bit machine affords all its concurrently run programs together is only 256 bytes, our only option is to introduce physically partitioned memory. In doing so, we will build our biggest CdM-8-based demonstrator platform so far, the Coccoone machine.

---

<sup>1</sup>Interestingly, code space can still be shared, since instructions as such are not part of the changing state.

## 14.2 The Coccone machine: an OS demonstrator

The technology behind modern operating systems and their conceptual fabric are easily a big enough topic to merit a separate tome. Our purpose being to introduce the principles and give the flavour of simple enough specimen of various platforms, we opt at this point for a specific case study rather than a broad overview of the subject. Of course, every system is different and the details of our chosen solution are not at all universal, it is also the case that the fundamentals of architecture, process control and I/O management are fairly well established to be common to many platforms of this level. In the study of computing platforms nothing aids understanding better than painstakingly following a reasonably large study example; most systems architects of today learned their craft from experiences or their predecessors expressed in products rather than academic case studies, and the reader of this book is already in a much better position. Naturally, specialist literature abounds to serve those wishing to explore the state of the art in greater depth after the case presented below is fully understood.

Coccone<sup>2</sup> is a small (simulated) computer, a whole system intended for use by a human (for demonstration purposes only). It includes a CdM-8 processor complete with a shadowed SP-register and paged memory support, a multibank memory subsystem, a Memory Management Unit (MMU), a Direct Memory Access (DMA) controller, a Process Table Monitor (PTM) and also various peripheral devices needed for demonstration of the functioning of an OS. Coccone is programmed in the language of the CdM-8 macroassembler in the manner described in earlier chapters. Compiled programs may be loaded onto a simulated flash card supported by the DMA controller and executed on the platform just like ordinary programs on ordinary computers are. The platform is multi-process and is equipped with an OS, named `cocos`.

Coccone is a reasonably large case study of the principles of operating systems, and it gives the reader a chance to peep into some of the subtle trade-offs that *system* design is typically concerned with. Despite its small size and limited facilities, it is well placed to expose the nature of both the problems and solutions employed in Platform 2 design, and besides it does it in both software and hardware. The presentation below will zoom out of the previously established scale in order to embrace circuits and code structurally and functionally, rather than explaining every little detail the way we did in earlier chapters. Nevertheless, the full schematics and assembly code are available from the book website enabling the reader to delve into the minutiae of the implementation to any depth she desires — as well as to modify and extend the design up to and including trying a completely different approach.

In this section we will focus on the systemic hardware structures, while the next section will expose the structure and functioning of the OS.

### 14.2.1 Memory subsystem

**Processor extensions.** An 8-bit address consistent processor can only address at most 256 bytes *at any given time*. This, in fact, means that there can be *several* lots of 256 bytes of memory available. The processor should be able to choose, from time to time, which one of those lots (called memory banks) is meant when it issues its `mem` signal. This can be done by some control circuitry associated with memory, which is placed outside the processor. However the decision still has to come from the program running on the processor. How?

First of all, it is clear that the choice of memory bank must be part of the process record. If the process is suspended and then resumed then according to the Principle of Transparency the program should find itself exactly in the same place in memory as it was before. For the processor to be able to use it in clock cycles involving memory, this information has to be physically kept inside the processor while the process is being progressed, which means that it must be in a register rather than private data. The only register that has spare capacity to keep the bank number is the Processor Status register.

We have already utilised bit 7 of the PS in order to be able to enable and disable interrupts programmatically. There are still three bits left unused in an 8-bit register, which can now be pressed into service. The new layout is as follows:

---

<sup>2</sup>The name is the Italian for “big coconut”. It is consistent with the Ecosystem naming style which is rooted in this particular plant.

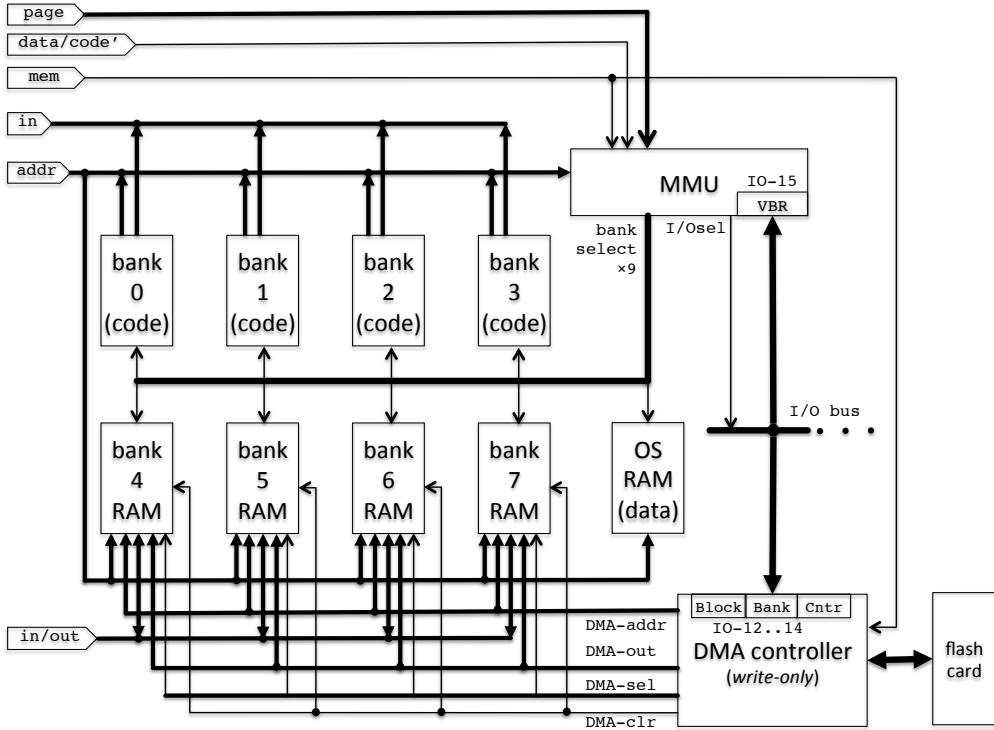


Figure 14.2: Coccone memory subsystem

Bit(s)	Description
7	interrupt enable
6–4	page no
3–0	CVZN flags

The new field in the PS is “page number”. The processor sees memory as one consisting of up to 8 size-256 “pages”, which could be separate memory units (banks) or segments of a single memory device addressed by  $8+3=11$  bits, or indeed anything in between. That is why we use the term page when we talk about the processor’s abstract view of the memory subsystem — and the term bank to refer to a physical memory device.

The difference between pages and banks would not merit an extra term if it were not for the fact that separate memory units are not just equivalent to a single unit in capacity and addressability, they surpass it in concurrency. While the single memory unit is perfectly capable of keeping all pages in it, it cannot be part ROM and part RAM, and different segments of it cannot be addressed simultaneously in the same clock cycle (as is the case in Coccone when its DMA is active). Another source of complications is the difference between Harvard and von Neumann architectures. Coccone utilises ... both, and for a good reason, which makes the relationship between pages and banks even less straightforward.

**Memory architecture.** The memory subsystem of the Coccone machine is displayed diagrammatically in figure 14.2. As one can see, it is a rather complex arrangement which pursues the goal of clean separation between the OS and the user space to achieve both safety and enhanced Platform 2 functionality. The subsystem includes as many as 9 banks of memory, four of which are ROM banks containing parts of an operating system. The arhcitecture is mixed: Harvard for the OS and von Neumann for user programs. The code for the OS resides in banks 0–3 and the OS data is kept in a single dedicated RAM bank marked as OS data (and is left unnumbered for that matter). The code in any OS code bank can also use banks 4–7 as data memory. This is necessary for the OS to be able to read and write user data during system calls. The data address space for the OS includes the I/O segment  $0xf0\text{--}0xff$ , which is, as before, used for interaction with I/O devices over the I/O bus in memory-mapped mode.

Memory for user programs (banks 4–7) is completely different. First of all, it is RAM and it is, as mentioned before, von Neumann. That is, all user code and data goes into the same memory bank. Secondly, all 256

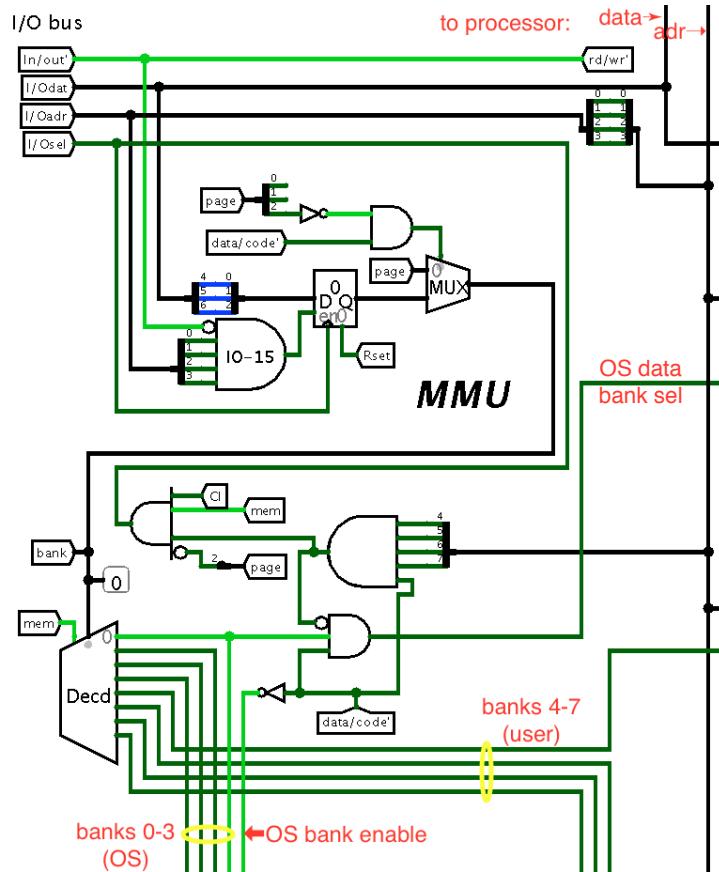


Figure 14.3: MMU implementation

addresses are used and there is no memory-mapping of the I/O. If a user program needs to interact with the outside world it has to issue system calls and the OS will do the interacting on its behalf. Another novelty is that the user memory is *multiported*, just like registers inside the processor. In any clock cycle the processor is able to perform a memory operation on a bank 4–7, but if no such operation is required, then the memory cycle can be stolen by an external agent, a *Direct Memory Access* (DMA) controller, which supplies address, data and selection signals on a second “port”, i.e. set of pins. Cocccone’s DMA controller is intended for reading the (simulated) flash card into a user memory bank in order to run programs stored on the card, so it does not support exchanges in the opposite direction: memory to flash. This makes the circuitry somewhat simpler. DMA is a common feature of computer organisation, without which storage devices and high-speed network interfaces would need to rely on machine instructions for data transfer — and that would be very inefficient.

**The Memory Management Unit (MMU)** in Cocccone replaces a few gates used previously to separate the I/O address space from genuine memory. Here the task is slightly bigger: the memory subsystem must decide whether the processor’s high-mem cycle is intended for memory or I/O and which memory bank to use. The latter may depend on the program being executed, and in order to support that the MMU incorporates an I/O register VBR (Virtual Bank Register) accessible as the address IO-15 0xff. Interestingly, programmatic access to the register is subject to the MMU itself: the page and mode (data or code) asserted by the processor in the current cycle will determine how that address is interpreted by the MMU. In particular, for safety reasons writing to the register is only possible in data mode from one of the OS code banks.

Given the content of the VBR the MMU acts as a simple combinational circuit decoding the quadruplet

$$(\text{page}, \text{data}/\text{code}', \text{mem}, \text{addr})$$

onto 10 output wires: 9 wires of bank-select signals going to individual memory units and the familiar I/Osel signal which now originates in the MMU. At most one of the wires will be up in any given clock cycle. The signal mem acts as an enable signal in that all 10 output wires are down when mem is down and one of them is up when mem is up.

The following table fully specifies the mapping of the MMU inputs onto its outputs. Here bank-select is the number of the select wire to be raised by the MMU (provided that mem is up).

page	data/code'	addr	bank-select	I/Osel
$p$ $0 \leq p \leq 3$	0	any	$p$	0
	1	$<0xf0$	VBR	0
	1	$\geq 0xf0$	none	1
	1	any	$p$	0

The bank-select value 0 represents two wires: the bank-0 selector if the signal data/code' is down and the OS data bank selector if data/code' is up. All of the above allows for a straightforward implementation. Note that the only values of the VBR that are legal are 0,4,5,6 and 7.

To summarise, pages 4–7 go through to the corresponding bank under all circumstances, and so do pages 0–3 for code. If a page number in the interval 0–3 is used for data, then if the address is below the I/O segment, the MMU decides which bank to use. Initially, after reset, the content of the VBR is 0 and consequently if the MMU is never touched the platform will direct all access to data for pages 0–3 to the OS data bank. Otherwise the VBR will select a user-space bank (4–7). Finally, for a page 0–3, data mode, and an address in the I/O segment, the MMU will ensure that the I/O bus is engaged rather than any RAM bank. A fragment of the Cocccone circuit diagram in figure 14.3 illustrates the design of the MMU for those who prefer a diagram to verbal descriptions.

**DMA controller** As mentioned earlier, the intention of the DMA controller is to support a flash card as a file system on which files with executable programs may be stored. In principle this can be achieved by attaching the card to the I/O bus via a simple interface allowing the processor to read one byte from the card at a time. The reason why this is not a good idea (and why most commercial systems utilise DMA controllers for their mass storage and other high-data-volume devices) is that the efficiency of the data transfer between memory and an external device via a processor is very poor indeed. In order to transfer one byte from device to memory a load and store instructions have to be executed; then there must be a simple UNTIL-loop counting the bytes in a multibyte transfer, requiring a decrement and a branch instructions. Finally the memory address for such a transfer is pointed to by a register pointer, which needs to be incremented:

```
loop:
    ld r0,r3
    st r1,r3
    inc r1
    dec r2
    bne loop
```

One iteration of this loop would require 10 clock cycles! This means that the transfer proceeds at 10% of the data memory throughput on our simple platform. Although modern processors are able to improve this poor result by utilising instruction-level parallelism, it is bad engineering to use a critical resource, such as the processor, for a considerable period of time (around two thousand clock cycles to transfer a single bank from flash to memory) on a routine task that can and should be supported by the peripheral device directly.

Enter the DMA controller. This is a simple machine which implements the above loop in hardware bypassing the processor altogether. It is based on 4 registers, 3 of which are accessible via the I/O bus.

FlashAddr is a 12-bit register, the senior 4 bits of which are writable via the I/O bus at IO-12. The flash card we use for Cocccone is an array of 16 blocks, each 256 bytes in size. Consequently the 12 bit value in the register points to the block (4 MSBs) and the address inside the block (8 LSBs). Only the former can be set by the program and when it is, the junior bits are set to 0 by the controller automatically.

Bank is a 2-bit register containing the bank number less 4 to which data must be transferred. So bank 4 is stored as 0, bank 5 as 1, etc. Notice that the DMA controller has no access to the OS data bank. The

two bits are writable via the I/O bus at IO-13. For convenience, the two-bit number is placed inside the byte following the format of the PS: bits 1 and 0 are taken from bits 5 and 4, respectively, where the page number would be found.

`MemAddr` is an 8-bit register not writeable via the I/O bus. However a write to `Bank` also resets its contents to zero.

`Counter` is an 8-bit register writeable via the I/O bus at IO-14. It counts down the bytes still to be transferred. A write to this register also triggers the DMA operation: the copying of that many bytes from the flash address contained in `FlashAddr` to the bank number indicated by the content of `Bank`. The transition from a nonzero content of `Counter` to 0 raises an interrupt with vector 4.

When the program has written into the three writable registers, the DMA becomes active. In each clock cycle it checks whether the target memory bank is free. The bank is free if either the `mem` signal is down or else the target bank is *not* selected by the MMU. If the bank is not free, the DMA waits for the next clock cycle. Otherwise it “steals” the memory cycle to transfer one byte.

Here is how it is done. The user banks 4–7 are multiported. This means that the usual four signals: `sel`, `ld`, `A` and `D` found on the RAM chip are duplicated (by additional combinational circuitry put around the chip), so that two actors can command the RAM chip to perform a memory operation: the processor and the DMA controller. Naturally, they must not *both* do it in the same cycle, but that is assured by the DMA logic, which only steals the cycles that are not used for exchanges with the given bank by the CPU. So in those cycles (see figure 14.2) the signal `DMA-sel` acts as `sel`, `DMA-addr` as `A` and `DMA-out` as `D`. Additionally the controller utilises the signal `DMA-clr` to asynchronously zero out the content of the target bank. This is needed because the transfer may be, and typically is, shorter than the full 256 bytes needed to overwrite the whole memory content, but when the target bank is run as a program, we wish the program to see clear memory as if it were running on a standalone single-bank machine after reset. This signal is pulsed at the end of the clock cycle in which the `Bank` register is written into, with the effect of clearing the corresponding bank. After the transfer is completed and the interrupt is raised, the controller becomes inactive until the next write into `Counter`.

Below is the summary of the transfer algorithm implemented by the DMA controller, written in pseudo-code.

```

while Counter>0
    if target bank is busy continue
    read byte at FlashAddr into memory bank Bank at address MemAddr
    FlashAddr=FlashAddr+1
    MemAddr=Memaddr+1
    Count=Count-1
Raise interrupt via vector 4 and wait for Counter>0, then repeat from beginning

```

### 14.2.2 Extended processor

The memory subsystem analysed in the previous section requires a processor that can maintain the page selector output consistently throughout its operation. While most instructions are completely oblivious of any aspect of memory architecture (with the exception of the data/code dichotomy), anything to do with interrupts, including system calls which we have not seen yet, requires a small extension to the Secondary Decoder and a different approach to stack management. We will start with the latter.

**Stack discipline.** Processes cannot usefully share a stack. Indeed, a process can get suspended and resumed at any time and, according to the principle of transparency, it should be unable to see any change in its accessible memory content after resumption. If a process is suspended and then resumed, and if two processes share a stack, it is possible for the first process to push new values onto the stack and then pop them off only to find that they are different, because the intervening process also, quite legitimately, pushed its own content onto the stack meaning to pop it off later — but was preempted by the first process. The LIFO discipline is destroyed by the arbitrary interleaving of processes accessing a LIFO queue, which any stack is.

As before, we must segregate stacks belonging to different processes. This is easy to do in a multibank architecture in view of the fact that each process runs in its own memory bank and is perfectly safe to use the top of the address space as its private stack. One difficulty still remains though. Whereas stack areas belonging to different processes are cleanly separated, the stack pointers are kept in the same hardware register, namely the SP. Not a problem, one would think, since processes have to share the GP registers anyway, why not the stack pointer? The answer to this question lies deeply in the instructions set of our demonstrator machine.

A processor designer is faced with a stark choice. Either the special purpose registers are included in the GP register file, which reduces the number of registers that can be used in a computation, but which allows SP and PC to be treated like other data, or they are not. In the latter case, further instructions are required to exchange data between special purpose registers and the GP ones. The op-code space is also limited, just as the size of the register file; both register operands and the opcode have to be encoded in an instruction of limited size. The compromise for CdM-8 was to opt for 4 GP registers (which is barely sufficient to support computation without needing to save/restore all the time), but the price the designer agreed to pay in doing so was the need to include `lds` and `addsp/setsp` in the instruction set to enable stacks to be placed in memory in a flexible manner and accessed directly depth-wise.

This trade off has another subtle downside, which becomes more evident now that we are studying the most complex platform we are to see in the book. In the course of a context switch, the *first* thing that a process must do is to save at least one register (where? without a single register at our disposal we cannot even load a memory address, so — onto the stack is the only choice). Pushing it onto the stack means altering the private data (of which the stack is part) of another process! That destroys transparency right off. And yet in order to save the stack pointer of the current process and to restore the stack pointer of a suspended process into the SP register *we must have access to at least one, possibly more, registers*. Only two options are available to us under the circumstances:

1. A protocol (agreement) between processes that an area of a certain size (a few bytes) below the stack pointer is *excluded* from the private data at all times. That is, the process (or rather its associated program) promises that no computation will at any point depend on data stored in memory within a certain distance *under* the stack pointer. This restores process transparency by allowing the interrupting process to defer to the OS for saving some registers on the current process's stack without overwriting its private data. Then, using the registers thus liberated the OS will be able to save the complete process record, *including* the content of the SP *before* registers were pushed (for example, by calculation). This way a clean context switch can be put into effect.

The upside of this method is that we can continue to use an ordinary processor as before. The downside is the time needed for context switch and, for small systems, such as CdM-8, a larger code space required to define all the save/restore actions.

2. The second option requires a hardware extension. If the segregation of the processes' private spaces is achieved by placing them in different memory banks, it is possible to use the page value (which is, we remind, a part of the PS content) to index an *array* of stack pointers. This way each process (and there will be a hardware limit on the number of processes the system is able to support) has its dedicated physical stack pointer register which never needs to be shared. This technique is called *register shadowing* and the replicas of the SP are called shadow SPs. The obvious upside of this solution is the simplicity and speed of context change, the main downside is the physical limit on the number of processes supported at any given time. For a tiny system, such as CdM-8 the downside is not really significant since all other resources are even more severely limited (the size of private data, the processor simulation speed, etc.). Unsurprisingly the CdM-8 designer opted for register shadowing.

The modifications needed in the register file in order to support SP shadowing are entirely straightforward and we are not going to dwell on them. Instead let us consider how the concept of paged memory changes the processing of interrupts.

**Interrupts.** In a multibank memory generally two stacks are involved in an interrupt: that of the active process and that of the process to be created as a result of the interrupt. Additionally there is a question of which bank's interrupt vector is going to be used for determining the initial state of the newly created process. Let us consider the design space again.

Instruction	PD signal	Ph1	Ph2	Ph3	Ph4	Ph5
ioi	ioi	SPact tmpg	SP2b0 data mem PC2b1 SPact tmpg	SP2b0 mem data PSR2b1 tmpg	vec latchPC mem ld tmpg	vec odd ld mem LatchPSR tmpg

Instruction	PD signal	Ph1	Ph2
rti	rti	SP2b0 data mem ld latchPSR SPact up/dwn' pglk	SP2b0 data mem ld latchPC SPact up/dwn' pglk

Figure 14.4:

An interrupt is signalled by a device providing only the request and, if it is granted, the vector number. Nothing in those two would indicate which page or pages should be involved. Consequently, the only option in a multibank memory is to assume a certain page number from which the processor will fetch the vector. Fortunately, since the latter contains the new PS value, it will also define which page the new process is required to be started in, and therefore which stack pointer will be used. In the CdM-8 design the assumed page for vectors is page 0.

Recall that a granted interrupt is processed by pushing the content of the PC and PS onto the stack, fetching the vector and loading the PC and PS with values from the vector. Our next question is: which stack? Should the processor use the current stack or the stack associated with the new PS?

The answer is quite obvious. Since the interrupt will need to be returned from in order to reactivate the suspended process, it will be necessary to determine which page's (data) memory the values of the PC and PS were saved in. That information is contained on top of that stack, and the stack pointer pointing to it is shadowed out in the new process's page, making an `rti` impossible.

We conclude that the saving of the current PC and PS should be done on the destination stack. That is still not straightforward, since the destination page is not available until the interrupt vector is read. However, it is to be read into the PS register overwriting the old content that by that time is assumed to have been saved. So the saving of the two registers should both precede and succeed the reading of the vector, which seems to be impossible. The obvious solution: to read the vector twice, once for determining the destination page and the second time to actually put the interrupt into effect seems quite inefficient, due to its lengthening the already long sequence of the Secondary Decoder. We will not even consider the complications of adding special registers to the Data Path to avoid the second reading of the vector since that would involve additional control signals, etc., which we must avoid altogether as simplicity is one of our primary design goals.

A compromise solution is available nevertheless, which the CdM-8 processor adopts. It is to always save the current PC and PS in page 0, i.e. the page from which the vectors are fetched. If the ISR is also placed in page 0, the solution is fully consistent. If the vector suggests that the ISR is located on a different page, then the hardware will first save the PC and PS on page 0's stack and then load them from the vector. It will result in the ISR not having the PC and PS pushed down the local stack for it to be able to `rti` at the end. That, however, is a problem which can be solved in software, should page 0 become too small to accommodate all ISRs.

A complete implementation of paged interrupts in the Secondary Decoder is presented in figure 14.4:

It only takes two additional signals, `tmpg` and `pglk` to achieve the desired result. The first one tells the Data Path to use a temporary page register for the output pin `page`. The register is reset to 0 as soon as the `ioi` instruction is decoded. The temporary page register is used in system calls as well, so we will return to it later.

For the `rti` instruction it is important that the loading of the PS register, which happens in its Phase 1

does not immediately affect `page` (and does not immediately shadow the current SP). In order to prevent it, `page` is connected to the relevant section of the PS register via a page trap register. The latter is triggered by the low level of the signal `pg1k`, so as long as `pg1k` is down `page` follows the corresponding PS content. However, the high level of `pg1k`, which is established at the beginning of Phase 1 before the new content of the PS register is latched effectively locks the current page until the end of Phase 2.

**System calls and osix.** We are prepared to see a special process, the operating system, taking control by interrupt from time to time in order to manage processes and I/O. However, a request for OS services may come from different sources. The principal source is of course input/output, since those are events that are out of synch with any processes. However, a process itself may need OS services for initiating I/O (remember that user processes do not have device interfaces memory-mapped onto their address space), or for scheduling events. One example of this is process termination, where a process is telling an OS to context switch to another process and never return. The OS would need to reclaim any resources (the memory bank and any system areas) associated with the process as well. Process creation is another example. This requires allocating an unused bank, initialising a new process and possibly making it active.

Even activities associated with input/output can be synchronous. For example, it is at a definite point in the program that a process may require a keyboard input, and it will request it by allocating some data space for the string to be entered on the keyboard and telling the OS to suspend it until the string has been entered.

All such requests for services are termed *system calls* and we must now focus on their possible implementation in our demonstrator machine.

The first thing that comes to mind in conjunction with all kinds of system call is that we already have a mechanism to support them. We saw in earlier chapters the subroutine mechanism and its Platform 3 support via object file linking. One might think that all that is required here is a number of object files, a library of sorts, containing the required functionality, which would then be linked with compiled user code thus implementing the parts of the OS relevant to the program.

Unfortunately such a solution would have numerous disadvantages rendering it completely unsuitable. First of all, the services in question are often to do with I/O and invariably need to access OS data. A subroutine executing as part of the user program would share the context of the latter: the value of the PS including the page, and consequently the mapping of the page via the MMU onto the user space. The context of a user program would not support access to either the I/O segment of the address space or the OS data bank. Finally, even if we overcame those difficulties by various software and hardware means, we would also have to ensure that the callee subroutine had access to the caller's context, most importantly the page value of the PS, to establish the provenance of the caller process.

At this point it is easy to see that all such problems would go away if we were able to induce an *interrupt* rather than subroutine execution programmatically whenever a program required an OS service. Indeed, the context of the requester process would be cleanly preserved by the interrupt logic (the saving of the PC and PS on the OS stack), and the OS would be able to find out which process called and at some point resume it by `rti`'ing. To make this happen, an OS-interrupt instruction is required, which would cause such interrupts, and which would be able to convey additional information, namely: which page to use and possibly some indication of what service is required. The former is needed because now we do not have to assume that there is only one bank in which the vector is to be found (as was the case with I/O interrupts). A machine instruction has a bigger information capacity than the combination of the IRQ signal and the integer representing the vector address. It is still convenient to utilise a single (assumed) vector for system calls in every bank, since the vector space is extremely limited (no more than 8 vectors of any kind are supported by CdM-8). As for the choice of page and the service parameter, both are adequately served by the PS content. The page is already a part of it, and the CVZN flags permit as many as 16 types of system call to be easily distinguished on the page.

Enter the CdM-8 `osix` instruction. It is a 0-register instruction that has an immediate operand in which the low-order 7 bits are used as follows:

Bit(s)	Description
7	<unused>
6–4	new page
3–0	new CVZN flags

This instruction causes an interrupt using vector 0, 0xf0, fetched as before, from code memory on the *new page*. The current PC and PS are saved on the *new page* stack, and the new PC and bit 7 of the PS are fetched from the vector. The rest of the PS bits are set according to the immediate operand. Since bit 7 is fetched from the vector, the OS has the freedom to do its processing with interrupts disabled or enabled (the latter being a dangerous and rarely used option, exercised mainly with long system calls in order to avoid data loss from delayed interrupts). Since the rest of the bits come from the immediate operand, the system call ISR can analyse the flags using one or more suitable branches.

For example, in the OS *cocos* deployed on the Coccne machine, the instruction

```
osix 0x02
```

tells the OS to terminate a process. The immediate operand points to bank 0, CVZN=0b0010 and the system call is immediately identified by the OS ISR using the **bz** branch instruction.

Here is the implementation of the **osix** instruction in the Secondary Decoder. It is a whopping 6-cycle sequence laid out as follows:

Instruction	PD signal	Ph1	Ph2	Ph3	Ph4	Ph5	Ph6
ioi	osix	SPact PC2b0 PCinc ld mem <b>L-tmpg</b>	SPact tmpg	SP2b0 mem data PC2b1 SPact tmpg	SP2b0 mem data PSR2b1 tmpg	vec latchPC mem ld tmpg	vec odd ld mem LatchPSR <b>tmpg2PSR</b> tmpg

Here we have two new signals. In Phase 1, four familiar triggers: PC2b0, PCinc, ld and mem trigger the fetching of the immediate operand from code memory, and the signal L-tmpg latches it in the temporary page register, which is 7 bits in size and so takes all of the immediate operand in. The signal tmpg2PSR in Phase 6 latches the content of the temporary page register in the 7 low order bits of the PS register, and the remaining MSB of the register is also latched, but from bus 1, on which the odd vector byte is asserted by the familiar triggers vec, odd, ld and mem.

## 14.3 OS *cocos*

In this section we describe a specific OS developed for the Coccne machine discussed above. We begin with the central component of any operating system, the scheduler.

### 14.3.1 Scheduler

The job of the scheduler is to choose which process to run after a *major* interrupt. A major interrupt is such that it must result in a scheduling decision. Some interrupts do not require scheduling, for example those that simply cause the ISR to transfer the next portion of data and wait for the device to become ready again. Major interrupts happen when the job of an I/O device (or a process in the case of a system call) is in some sense complete, and a context switch may be required.

The interrupt itself does the job of suspending a process, as we have seen earlier and the **rti** instruction resumes one. The scheduler complements that in two stages

**Swap-out** is the first stage at which the current process is fully evicted from the processor and saved in its private space. The hardware only deals with the bare minimum, saving the PS and PC content onto the correct stack; the scheduler must also copy that information to the process table and save the rest of the registers there in case the scheduling decision causes a different process to be resumed. This is called *preemption*, and the process suspended by the scheduler is said to be preempted by the new active process.

Swap-out is part of the scheduler in the general case, but for *cocos* we adopted a strategy that makes this procedure so simple that it is relegated to the ISR itself. The trick is to use the shadowed SP to

offset:		+0	+1	+2	+3	+4	+5	+6	+7
address	priority	suspension flags	<reserved>	r0	r1	r2	r3	PS	PC
0x00	0	0b1000 0000		...	...	...	...	0x21	...
0x08	1	0b0000 0000		...	...	...	...	0xc4	...
0x10	2	0b0000 0001		...	...	...	...	0xf2	...
0x18	3	0b0000 0000		...	...	...	...	0xd0	...
0x20	4	0b1111 1111							
0x28	5								

Figure 14.5: Structure of the process table

keep the *operating system* stack pointer pointing to the end of the current record of the process table. This way any interrupt will write to the PC/PS part of the record automatically, and the GP regs can then be saved in the right order to the right place by a single `pushall` instruction. At this point, the current process is properly swapped out, and all that is required of the ISR before the next Swap-in is not to use the stack. If the ISR needs to use the stack, the stack should be relocated to avoid overwriting the process table.

**Swap-in** is the final stage at which the scheduler decides which of the suspended processes should be made active and then restores it to all the hardware registers. The swap-in part of the Scheduler is given a single parameter: the event that caused the interrupt. Before making the decision about which process to resume, the Scheduler marks all processes awaiting the event as no longer awaiting it. In *cocos* it uses the stack pointer to traverse the process table. All interrupts are disabled when Swap-in is active, so the SP is safe to use.

Notice that the SP shadowing is managed purely by the hardware, since saving and restoring the PS automatically changes which of the shadow SPs becomes the current Stack Pointer.

If the nature of the interrupts is unclear at launch, the ISR issues `pushall` first of all not to destroy the data in any of the GP registers irreversibly. It will then receive or send data and determine if the interrupt is major. If it is not major, the interrupted process is resumed. This is done by reverting the stack to where it was after the `pushall` (if the ISR is such that it needs to use stack operations, in which case the stack pointer was moved to that auxiliary space), and executing a `popall` followed by an `rti`. For example the keyboard ISR must first determine whether the key pressed on the keyboard is Enter or Ctrl/L. If neither, the interrupt is minor, the ISR only needs to store the key code (ASCII value) of the character in memory and the interrupted process may continue.

If the interrupt does turn out to be major, the control should be passed to the Scheduler's Swap-in procedure, which marks any processes waiting for this interrupt as no longer waiting, and which makes active the highest priority process that is not waiting for an event.

In *cocos*, for simplicity, no ISR is ever suspended, since they do not run with interrupts enabled anyway, but in bigger systems, the scheduling protocols are much more complex.

**The process table.** The key data structure of the scheduler is the *process table*. The *cocos* process table `proctab` is placed at address 0 and is a size-six array of process records, each 8 bytes long with the structure shown in figure 14.5

Each entry (array element) of `proctab` corresponds to a process priority at which at most one process may exist. The index-0 entry is reserved for the shell, which is the part of the OS that talks to a human via the teletype. The shell has the highest priority (0) according to its `proctab` entry index. The number of entries in `proctab` is in fact variable. In order to indicate how many entries exist at any given time, the entry *after* last is marked with the end-marker `susp_flags=0b11111111`. This limits the maximum number of processes on the Coccone platform to 5: the shell and up to 4 user processes, which coincides with the number of RAM banks on the platform available to user programs. In fact, the array element marked up by the end-marker has dual purpose: it also represents the `idle` process of the OS, which has no work to do, never terminates, and when active, it executes a single machine instruction `wait` in an infinite loop.

We must stress that a `proctab` entry is only *valid* when the corresponding process is suspended. If it is active, then the byte `susp_flags` is still valid (and contains the value 0), but the rest of the fields correspond to the process at the last event that suspended it. So the process table should be called the table of *suspended* processes, but we keep the short name for convenience. Despite the fact that the entry of an active process is in a sense out of date, the page portion of the PS content is accurate, since processes cannot migrate between pages and always occupy the same page (and consequently the same bank, since the MMU only deals with data memory) until they terminate. For that reason it is convenient to associate a process with its bank of residence and that is how processes are identified when a human interacts with the `cocos` shell.

So what are the suspension flags `susp_flags`? There are 8 of them, 8 bits in a byte, but the next byte is reserved in case more are needed in the next version of the OS (or any modifications that the reader is welcome to try by herself). Each bit in the `susp_flags` byte corresponds to a type of event. Here are the types of event currently supported by the OS (while user processes can define more event types for their own purposes):

```
define EVENT_shell, 0x80
define EVENT_DMA, 0x20
define EVENT_released, 0x10
define EVENT_EOL, 0x01
```

(The `define` macro used here is the standard macro library macro that assigns symbolic names to numeric constants.)

An `EVENT_shell` happens when the character Ctrl/L is pressed on the keyboard. When the shell has no work to do, it requests suspension pending this event type, and the keyboard ISR (part of the keyboard *driver*) raises an `EVENT_shell` upon receiving Ctrl/L from the (simulated) keyboard. This way the shell only works for one keyboard command at a time, but is available at any time at request.

An `EVENT_EOL` is raised when the keyboard ISR detects that the Enter key has been pressed. This allows a user process to get suspended pending this event.

An `EVENT_DMA` is raised when the DMA controller has finished transferring data from flash to memory. It is important to wait for this event before running the program thus transferred. Failure to do so would result in a partially transferred program being launched with unpredictable consequences.

Finally, an `EVENT_released` is used by the shell in order to be able to add an entry to `proctab` for reserving a bank. This is needed because when a program is to be launched, the shell will scan `proctab` (the PS field of each entry) to determine which banks are being used. So, unless a bank is on `proctab`, the shell will assume that it is available. On the other hand, if a bank is referred to by a `proctab` entry, it is in fact a process and can be made active (even if no valid record of it exists) when it is no longer awaiting some event.

In order to facilitate observation of events that arise and processes that evolve on the platform, a special hardware facility was added to the Cocccone machine, which spies on the OS data bank in order to visualise changes made to the process table, see figure 14.6. Since the content of the PC and of the GP registers included in the process record is of little use in observing a rapid progression of a set of processes, those fields are not visualised. By contrast the suspension flags go up and down infrequently, reflecting much more course-grained events, such as key presses on the keyboard or creation and termination of processes. Similarly, since each process on the Cocccone machine is associated with a memory bank and that, for code, coincides with the `page` field in the PS register, it is useful to be able to see the “home bank” of a process juxtaposed with the suspension flags.

In the figure one can see that the system is running two processes, both are suspended, which means that the active process is the once called “idle”. The top priority process is the shell in bank 2 (bank 2 is permanently allocated to the shell). It is suspended awaiting the event 0x80, which is what we defined earlier as `EVENT_shell`. This event is raised by the keyboard driver indicating that it has detected the keystroke Ctrl/L on the simulated keyboard. The shell will be resumed as soon as this happens. The priority-1 process resides in bank 4 and is suspended awaiting the event 0x01, which is defined above as `EVENT_EOL`, or end-of-line. This event is raised by the keyboard driver as well, as soon as the Enter key has been pressed.

Interestingly, the bank number of a process almost never changes despite the fact that the program may use system calls in which case the PS field `page` will definitely change, reflecting the fact that one of the OS banks is being used to progress the requested service. Nevertheless for that PS content to find its way to the process table an interrupt must happen, and our little OS runs all its user services with interrupts disabled.

## Proc Table Monitor

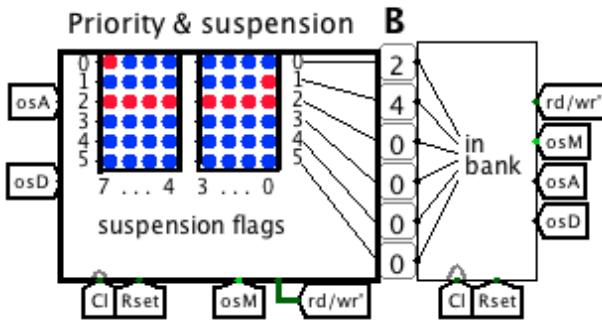


Figure 14.6: Coccone process observation facility

The exception is the shell, which uses system calls for loading programs. One of its system calls is answered by bank 3 of the OS which deals with extensions, in this case a file system. When the file system engages the DMA controller to copy a program from the flash drive to memory, it enables interrupts for the duration of the transfer. This is in order to register the interrupt from the DMA controller that signifies completion, but also to utilise the considerable number of clock cycles that the transfer takes by allowing other processes to progress in the mean time.

Having discussed what the Scheduler does and how its activities may be visualised for monitoring purposes, we now turn to its implementation, which is a part of the OS code deployed in bank 0.

**Scheduler code.** Since any OS processes cannot be suspended, the OS does not need to have a persistent stack either. The stack is required for short bursts when the OS is active. However, due to SP shadowing and the fact that the OS occupies a separate bank, its stack *pointer* is persistent. The key idea of the scheduler is to **always set the OS stack pointer on top of the proctab entry** of the user process about to become active, before context-switching to it.

For example, the memory snapshot presented in figure 14.5 indicates that the active process is the process with priority 1, since it is the highest priority unsuspended process. The scheduler, when it was about to activate this process, set the SP to the address  $0x08+6 = 0x0e$  and issued an **rti**. As a result the processor read locations  $0x0e$  and  $0x0f$  into the PS and PC registers, respectively, which caused a context switch to the priority-1 process, and the OS stack pointer was left pointing to location  $0x10$ , which is right on top of the **proctab entry** of the active process.

The code fragment in fig 14.7 shows the complete scheduler, which only takes a few lines. When an ISR branches to **sked**, it supplies in **r3** the event that it has recognised the interrupt as. It can potentially pass up to 8 events at the same time since the event bit-strings have a single 1 in different positions. In the ‘while’ loop, suspension flags at the beginning of the process record are cleared using **r3** as a mask.

Also notice the program labelled as “idle”. It represents the idle process which is created whenever the scheduler has not found a resumable process in the process table as it cleared the suspension flags. This is done by a combination of a **jsr**, **push**, and **rti** in the code. The idle process can never be resumed by the scheduler since it never processes an entry with *all* suspension flags up — it is the end marker. However, any process can be resumed by an ISR directly if it judges the interrupt to be minor. Consequently, the **wait** instruction in the idle process is placed in an infinite loop (**br idle**) to enable continued waiting after a minor interrupt.

### 14.3.2 Drivers

We have already mentioned drivers in the previous sections in conjunction with scheduling. So far we only needed the ISR operating as part of a driver, and that is the main part. Drivers also contain code needed for initiating I/O exchanges (whereas the ISR is something that completes one) as well as pieces of code

```

sked:  not r3  # make mask for clearing events, events now marked by 0

        setsp 0 # sp->proctab

        while
            pop r2      # r2=procstat0, inc sp (compensate later if complete loop
            inc r2      # procstat0== -1? (scan complete, reached process "idle")
            stays ne    # no,
            dec r2      # r2=procstat0
            and r3,r2    # quench bits assoc with posted events
            push r2      # store procstat0 adjusted for events
            if
                tst r2      # see if procstat0=0
                is z          #       yes!
                addsp 2        # found the *first* ready process in pri. order
                popall         # run
                rti           #       it!
            fi
            addsp 8
            wend

            # complelinate for inc sp by pop earlier
            # addsp -1        # commented out since the next instr is also addsp

# no ready process, all waiting for events yet to occur
# enable interrupts and wait
        addsp 7      # SP -> process idle; 7, not 8 to compensate for inc sp earlier
        jsr launch_idle      # SP->[idle]
idle:  wait
        br idle      # in case of minor interrupt

launch_idle:
        ldi r0,0x80 # PS, int enabled, bank 0
        push r0      # SP->[0x80,idle]
        rti

```

Figure 14.7: *cocos* scheduler

implementing control functions in respect of a peripheral device. *cocos* is a tiny system, so we are unable to demonstrate the full range of drivers and the variety of driver structures. We confine ourselves primarily to interrupt processing, leaving the rest of the I/O code in the domain of application-related system calls.

The main interest for the purposes of this book is the keyboard driver, which supports the keyboard/tty interface, see fig ??, and which we will look into in some length in this section. It is interesting because it makes full use of the Scheduler and process signalling using events.

The relevant segment of the OS code is presented below.

```

219 ##### KB driver
220 KB_ISR:
221
5a: ce      pushall      # swapped out! all regs available
5b: cc 80   addsp        128      # move stack to reserved area
224          # for no-interrupt processing
225
226
5d: d1 f1   ldi r1,KBtty_data
5f: b7      ld  r1,r3      # read a char from KB
229
230      if
60: d2 0c   ldi r2,0x0c      # ASCII <FF> (Form Feed), calling shell
62: 7e      cmp r3,r2
63: e1 69   is eq
65: d3 80   ldi r3,EVENT_shell # post event to wake up shell
67: ee 3a   br sked        # back to scheduler
235
236      fi
237
69: d2 32   ldi r2,data.KBbuf
6b: ba      ld  r2,r2      # r2->current char in user buffer
6c: 0a      tst r2        # is there a user buffer?
6d: e0 ab   beq KBexNS    # no, ignore interrupt, no proc waiting
242
243      if
6f: d2 08   ldi r2,0x08 # ASCII <bs>, backspace, need to rub out 1 char
71: 7e      cmp r3,r2
72: e1 8b   is eq
74: d0 31   ldi r0,data.KBcnt
76: b1      ld  r0,r1      # r1=KBcnt
248
249
77: 8c      inc r0
78: b2      ld  r0,r2      # r2=KBbuf
252
79: 8c      inc r0
7a: b3      ld  r0,r3      # r3=KBsize
255
7b: 77      cmp r1,r3
7c: ea ab   bge KBexNS    # KBcnt=KBsize, buffer empty, ignore <bs>,
258          # exit without sked'ing
259
7e: 88      dec r0 # r0->Kbuf
7f: 8a      dec r2
80: a2      st   r0,r2      # KBbuf=KBbuf-1
263
81: 88      dec r0 # r0->KBcnt
82: 8d      inc r1
83: a1      st   r0,r1      # KBcnt=KBcnt+1
267
84: d1 f1   ldi r1,KBtty_data

```

```

86: d3 08      269          ldi r3,0x08
88: a7      270          st   r1,r3           # echo to tty
271
89: ee ab      272          br KBexNS        # exit without sked'ing
273          fi
274
8b: a7      275          st r1,r3           # echo char to tty
276
277          # store it
278
8c: d1 31      279          ldi r1,data.KBcnt
8e: b4      280          ld   r1,r0           # r0 = count of bytes left in buffer
281          if
8f: 88      282          dec r0            # space for 1 char?
90: e0 97      283          is eq,or         # this one is last character to fit or...
92: d2 0a      284          ldi r2,0x0a
94: 7e      285          cmp r3,r2
95: e1 98      286          is eq            # is this one <NL>?
287          then
97: 3f      288          clr r3            # make current char NULL for string termination
289          fi
290
98: a4      291          st r1,r0           # r0 is cnt, save it in data.KBcnt
292
99: 8d      293          inc r1            # r1 -> data.Kbuf, is next byte
9a: b5      294          ld   r1,r1           # r1 -> next available byte
295
296
9b: d2 30      297          ldi r2,data.KBusr    # memory bank
9d: ba      298          ld   r2,r2           # r2=memory bank
299
9e: d0 ff      300          ldi r0,MMU         # r0 -> MMU I/O reg
a0: a2      301          st r0,r2           # set data memory bank
302
a1: a7      303          st r1,r3           # write char to user buffer
a2: 8d      304          inc r1            # advance buffer pointer
305
a3: 3a      306          clr r2            # MMU reset to page 0
a4: a2      307          st r0,r2           #
308
a5: d0 32      309          ldi r0, data.KBbuf    # store buffer pointer
a7: a1      310          st r0, r1           # to data.KBbuf
311
312          if
a8: 0f      313          tst r3            # no scheduling required, there
a9: e0 af      314          is ne             # set SP back to where it was (128+128=0
ab: cc 80      315  KBexNS:     addsp 128
ad: cf      316          popall
ae: d9      317          rti
318          fi
319          # scheduling required
af: d3 32      320          ldi r3,data.KBbuf    # no longer filling this buffer
b1: 30      321          clr r0
b2: ac      322          st r3,r0           # NULL it.
323
b3: d3 01      324          ldi r3,EVENT_EOL    # event for keyboard completion
b5: ee 3a      325          br sked           # go to scheduler

```

As suggested in the previous section the first thing the driver does is issue a pushall and arrange a stack area

elsewhere. The latter is done by `addsp 128`, which allocates a stack area with the top of the stack between the addresses 128 and  $128 + 40 = 168$ . This is an unused area and the expected stack size requirement of an ISR is just a few bytes. The advantage of using an `addsp 128` is that we do not need to be concerned about how to return the stack pointer to where it was if the interrupt has to be minor; a second `addsp 128` will do the trick.

Back to the program, the ISR immediately reads the ASCII key-code from the keyboard via the I/O register `KBtty_data`. The next step is for the driver to determine whether the character pressed on the keyboard is one of the control characters that the driver should process specially. It checks them one after another.

**Ctrl/L (ASCII <FF> character, “form feed”, interpreted by the teletype as a “screen erase” control character).** The keyboard driver interprets this as a “shell call”, i.e. the human’s command to activate the shell process. This is done via the event `EVENT_shell`, which is raised with the Scheduler by loading the event mask in `r3` and proceeding to swap-in (lines 231–235).

**Client interface.** At this point the driver must store the incoming character from the keyboard. The memory for that purpose, referred to as the “line buffer”, is statically (i.e. permanently, at design time) allocated in the OS data bank. Three labels in the template `data` are defined to label relevant items.

name	description
<code>KBusr</code>	requesting process (bank number in bits 6–4)
<code>KBbuf</code>	line buffer address in that bank
<code>KBsize</code>	line buffer size in bytes
<code>KBcnt</code>	remaining capacity (in chars) in the line buffer

The buffer is allocated in a user program and the above parameters are passed to the keyboard driver via a system call. When the driver’s ISR determines that a character has been pressed that requires a line buffer to process, it first checks whether the line buffer address stored at `KBbuf` is not null (lines 238–241). If it is null, then no process expects keyboard input, in which case the driver treats this interrupt as minor, ignores the character just received, disposes of the return address on top of the stack (line 363), pops off the pointer to the `proctab` entry `+2`, and passes the control to the section of the scheduler that pops off all the GP registers and does the `rti`.

**Backspace (ASCII <BS> character).** This should delete the last recorded printable character from the line buffer. Lines 243–273 deal with the processing of the backspace key, including echoing it to the `tty` for the purposes of rubbing the last character out. The code takes care not to violate the buffer boundary if the human operator attempts to press backspace when the buffer is empty.

**Ordinary character or <NL>.** Lines 279–310 store the character just received in the line buffer. This is less simple than one would expect given that the machine is operating in Harvard mode, so the MMU must be manipulated to map the current data page alternately onto the user process bank and the OS data bank. Another thing to take care of is to recognise the newline character `<NL>`, which is an event-raising key press (the Enter key on the physical keyboard) since the client process expects a NULL-terminated string of characters at once.

**Swap-in.** Now that the main job is done, what remains is to swap-in either via the scheduler (if a newline was received from the keyboard) or immediately (if it is any other character). This is done by lines 312–325, and includes nulling the `KBbuff` pointer after a newline to prevent the reception of any further characters until the next system call that sets this pointer.

**KB-driver: initiating keyboard read** The remaining part of the driver is placed in bank 1 and intended for initiating keyboard I/O. Here is the implementation of the `readln` system call, supported by `cocos`, which initiates input of one line of text via the keyboard (`osix 0x11`):

```

92          # osix 0x11
93          # r0' -> usr buffer
94          # r1' = max count
2e: c3      push r3
2f: c2      push r2
97
30: d2 32   ldi r2,data.KBbuf
32: bb      ld r2,r3
100         if
33: 0f      tst r3
34: e1 48   is eq
36: 8e      inc r2          #     r2-> KBsize
37: a9      st r2,r1        #     KBsize=r1'
38: 8a      dec r2          #     r2-> Kbbuf
39: a8      st r2,r0        #     Kbbuf=r0'
3a: 8a      dec r2          #     r2->KBcnt
3b: a9      st r2,r1        #     KBcnt=r1'
109
3c: 09      move r2,r1      #     r1->KBcnt
3d: 89      dec r1          #     r1->KBuser
112
3e: c6      pop r2          #
3f: c7      pop r3          #     restore regs: r2=r2', r3=r3'
115
40: c4      pop r0          #     PS' in r0
117
41: a4      st r1,r0        #     KBuser=PS (MMU ignores all bits
119                  #             in this byte, except page bits)
120
42: d1 fe   ldi r1,0xfe #     mask to clear N (signal success)
44: 44      and r1,r0        #
45: c0      push r0          #     back on stack (and in r0)
124
46: ee 4f   else
48: c6      pop r2
49: c7      pop r3
4a: c4      pop r0          # get PS' in r0
4b: d1 01   ldi r1,0x01 # mask to set N (signal failure)
4d: 54      or r1,r0
4e: c0      push r0
132         fi
4f: d9      rti

```

Here most of the code is entirely straightforward, but some lines merit additional explanation. (Lines 92–94) The system call expects the client program to place the address of the line buffer in `r0` and its size in bytes in `r1`. The apostrophe in the comments refers to the register values prior to the interrupt (a convention we adopted in OS code). It then uses the already familiar fields of the data template to set the ISR-related data: `KBusr`, `KBbuf`, `KBsize`, and `KBcnt` by moving a single pointer around rather than by name (more efficient but much less readable, hence extensive commenting, lines 103–111).

Lines 116–123 service normal completion, by popping the PS value prior to the interrupt off the stack and clearing the N flag in it. Lines 126–131 service abnormal completion, when the user program is trying to input a line at a time when another process is already using the keyboard. This is signalled back to the client program by setting the N flag.

Notice the virtualisation aspect of system calls. Since the instruction `osix` causes a synchronous interrupt, the OS resumes the caller via an `rti` rather than `rts`, and since the former restores the processes PS from the stack, the OS is able to update it in order to manipulate the CVZN flags. That makes a system call a proper “virtual instruction” indistinguishable (but for the execution time) from a real hardware-only Level 2

instruction. It is the system calls supported by the OS that make the language of a full Platform 2 different from the the language of the processor instruction set.

### 14.3.3 Shell

In common parlance, when someone refers to an OS, what is usually meant is the environment in which a programmer or computer user does their work. Phrases like "this application is available under Linux" refer not to an operating system, called in this case Linux, but the whole environment available to a computer user and based on it. Those things are often confused. We have already had a taste of what the OS in the narrow sense is expected to do, and that was to support multiple processes and I/O. Everything else runs on top of an OS-enhanced Platform 2, including any software that the user interacts with. The OS in the narrow sense is often referred to as the "kernel" and the human-computer interface program which assists the user in running applications on top of the kernel is often called the "shell", at least in the Unix<sup>3</sup> millieux, so we adopt the same terminology.

What is the purpose of the shell? In the simplest case it is a program that talks to the user over the keyboard and teletype in dialogue mode. The user types a command and presses the Enter key. The command is an order instructing the shell to perform a certain action. The shell recognises it and carries it out, usually by interacting with the kernel with the help of system calls. It is generally possible to write your own shell, since it is nothing more than a user program<sup>4</sup>.

In the case of the Coccoe machine the shell has to be a small program as it occupies a single memory bank (bank 2 of the OS instruction memory). Our needs are also quite modest. We would like to be able to

- run programs fetched by name from the flash drive (and also get a directory listing if we do not remember the name of the program we wish to run). For that we also require the shell to be able to locate a user-memory bank (4–7) not currently used.
- forcefully terminate a process
- lower the priority of a process

If we are able to do all those things using the shell, the rest can be done by user programs placed on the flash drive. So the above represents the bare minimum of shell services.

**Shell commands.** In many OS environments the shell is a program that has a *graphic user interface*, presented as a collection of windows, menus, buttons and other controls drawn by the system on a graphic monitor. Our tiny Coccoe machine has nowhere near sufficient power to support the complexity required for this, nor would our interest in platforms be well served by a profusion of interface details. Fortunately, along with the graphic facilities, most systems also support a simple line-based interface emulating the old teletype, and so does *cocos*.

For reasons of severe resource limitation, the commands have the most primitive form and errors are reported rather laconically. Here is a complete list of shell commands:

**The run command.** Runs a named program. If the name is not supplied, prints the directory listing on the teletype. Syntax: `r[Enter]` for the directory listing, and `r[SP]filename[Enter]` to run the program contained in the file *filename* on the flash card. We will return to files and file names later.

**The kill command.** Kills (terminates) the process running in bank *n* and deletes its record from *proctab*. Syntax: `kn[Enter]`.

**The nice command.** The name is a verb and it reflects Unix heritage. To nice a process means to lower its priority. The process becomes "nicer" in the sense that it permits preemption more readily as more processes take a priority higher than its own. The command lowers by one the priority of the process in bank *n*. This is achieved by exchanging the priority of the process in bank *n* with the priority of the process immediately below by swapping two consecutive *proctab* entries. Syntax: `nn[Enter]`.

<sup>3</sup>Unix is an OS, or rather a family of OSs used on most large machines and data centres

<sup>4</sup>but for matters of computer and communication security; in reality there is a whole security dimension to a high enough level platform which makes the shell a *privileged* program. Still, a user of sufficient privilege is able to write her own shell, which may have the same capabilities as the one supplied as part of the OS environment.

The shell prints `err` on the teletype if any error occurs: the process bank specified does not have a process associated with it, the priority cannot be lowered because it is already the lowest, etc. The implementation of the shell is straightforward. For most functions it relies on system calls implemented in the two parts of the OS kernel (banks 0 and 1), but since all OS instruction banks, including the one for the shell, have access to I/O and the MMU, some functions are implemented directly in the shell, for example, the `kill` command. The code of all `cocos`, including the shell being available from the book website, we will leave the analysis of the implementation to the reader as an exercise.

A final remark. From the point of view of the Tannenbaum classification of platforms, which was the starting point of the present book, the shell itself is ... a computing platform. It satisfies the definition since it is a collection of resources for manipulating digital data, and it has a language which is implemented by means of interpretation. The interpreter (the shell program itself) is produced using Platform 3<sup>1/2</sup> on top of an extended Platform 2 so it would not be an exaggeration to claim that it is a Level 4 platform, albeit a very simple one.

#### 14.3.4 File system

Almost any OS environment intended for interaction with a human contains a facility that associates names of some structure, represented as character strings, with data sets held in a storage device behind an I/O interface. The data sets are persistent in that they are kept intact despite the computer powering on and off, the OS loading and shutting down, etc. Persistent storage is what makes the programming effort worthwhile, since, once written, a program can be called in again and again and used with different data sets or in different control situations. Needless to say, data storage and retrieval, as well as organisation of data in persistent storage are absolutely fundamental to the modern day computing and communication technology.

However, the Cocccone machine is too small to support applications that process persistent data. It has no space for management software that could allocate, reclaim and reorganise space on a device such as a flash drive. Nor would it have sufficient resources to support a fully-fledged interface: directories, volumes, file links, etc. Our intention being extreme simplification for the purposes of a “systemic” overview, we have included only two functions without which the Cocccone platform could not operate.

**Directory listing.** This is a list of file names for which a data set exists on the flash card. Without it, the operator would have to know the exact names of all files available.

**File fetch.** Normally files are read (and written) a portion at a time, such portions being of a size convenient for a program to handle at once. The purpose of file fetch here is different: we are fetching files in order to run them as programs. The simplest and most efficient way of doing this is by copying the whole program to a specific RAM memory bank, of which we have four.

OS `cocos` is a *microkernel*-style operating system. This means that some of its parts are dynamic: they come from outside and operate in user space. Although microkernels are not yet mainstream for big machines, where a monolithic OS has enough space to fully implement any service a user program may require, for our small demonstrator they have many advantages in structure, simplicity and ease of understanding.

In a microkernel system the main part of the OS only takes care of process and memory management. Accordingly, the `cocos` shell uses system calls supported by OS *extensions* (bank 3) to copy the *loader* from block 0 of the flash card and run it. The loader then takes care of both filesystem functions listed above. The advantage of this approach is that the way files are packed into the card and any search facilities associated with it is only known to the loader. If necessary, the loader can be rewritten to allow more sophisticated techniques to be used, e.g. data compression, hierarchical file directories, etc, without the rest of the system or user programs noticing it.

**Utility `fsys`.** In order for the file system to be of any practical use on the Cocccone machine, we require a means of producing flash cards out of compiled programs. As was mentioned in an opening chapter, CdM-8 is too small to run its own design and programming tools, so any such tools are in fact cross-utilities: they require a standard desktop or laptop computer to be used. Production of simulated flash cards is achieved via the `fsys` utility, which is included in the CdM-8 Ecosystem along with the macroassembler, linker, emulator and the rest of the tools. It is run on a command line of a proper computer. The utility’s only command-line parameter is the name of the directory in which `.img` files are found. `fsys` creates a single file, `flash.img` which is a Logisim-format content file for a ROM or RAM, and as such it can be loaded into the simulated flash card in the circuit using Logisim’s component menu.

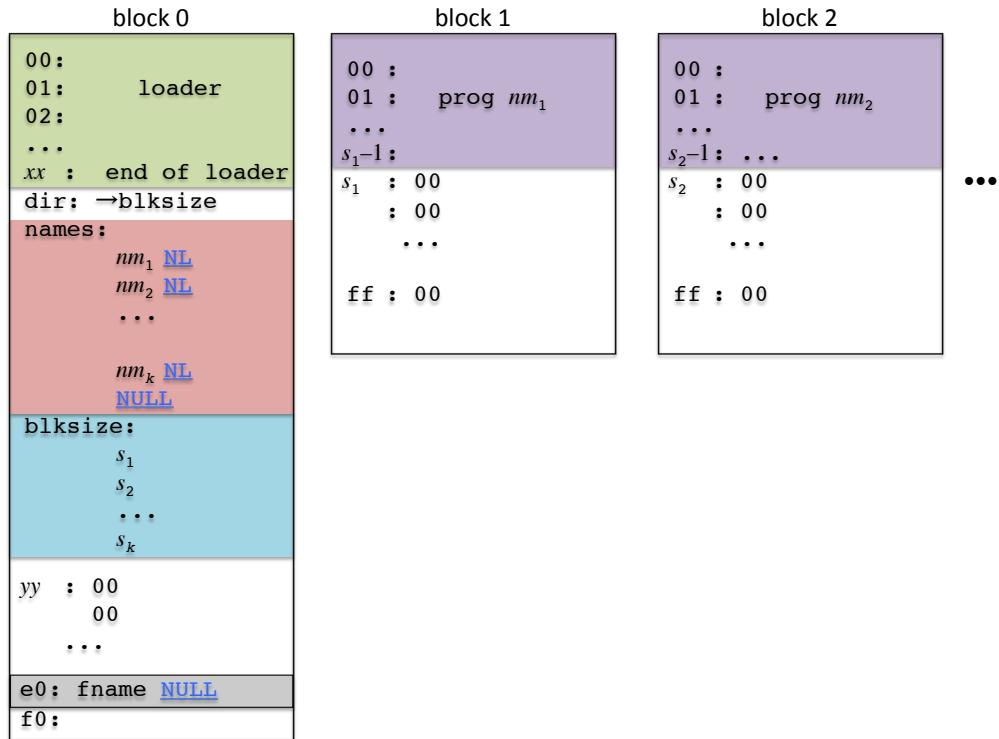


Figure 14.8: Structure of the flash card

**Structure of the flash card.** The structure of the file `flash.img` produced by `fsys` is presented in figure 14.8. The utility supports deployment of  $k$  programs,  $k \leq 15$  on the card, each in its own 256-byte block. Block 0 contains the loader program, which occupies the first  $xx$  bytes,  $xx$  depending on the specific loader program. When `fsys` places the loader in block 0, it first works out its size by determining how many consecutive zeroes can be deleted at the end of the image file. Once the loader has been placed, the utility records in the next byte a pointer to a size- $k$  array `blksize` that contains the sizes of all the programs to be placed on this card in some order. Then it places, in the same order, the names of those programs, separated by a newline and terminated by a NULL. Finally, it places the array `blksize` itself. The rest of the block is filled with zeros. The image files of the programs named in block 0 then go into their respective blocks in the same order as the names. The loader is aware of this structure and is able to extract the names and the array by addressing the values that immediately follow its code. The file system is parametrised with the loader, which is contained in an ordinary image file `_ldr.img`, and which the reader is welcome to inspect and/or modify.

The file system is used as follows.

1. When the shell has received the `r` command, with or without parameters, it issues a system call `osix 0x32` to the OS extension bank 3, which results in setting up the DMA controller to copy block 0 to a free user bank.
2. The executor of the system call `osix 0x32` places the (NULL-terminated) file name received by the shell (or an empty string if the `r` command had no parameters) at a fixed address in the newly acquired user bank: `0xe0`. The file name should not exceed 16 characters since the Coccone architecture does not permit the OS to write anything to a user bank directly above the address `0xf0`<sup>5</sup>.
3. Once the DMA transfer, which proceeds with interrupts enabled, is complete, the OS simply adds an entry to the process table, which refers to that bank in the PS content and has zero in the PC content. The entry is made active which results in the loader being run as a user program (this is precisely the microkernel style of operation). The loader may not be run immediately, since the process has a priority and can be preempted by another, but sooner or later it will be activated.

<sup>5</sup>those are I/O registers to an OS bank, not RAM locations. The DMA controller has no such predicament, and so up to 256 bytes of the program will be transferred, which is important since the program will be operating in von Neumann mode and can utilise its memory bank in full.

L3 Call	L3 <sup>1/2</sup> Call	Params	Protocol	Description
osix 0x01	await	r0: event mask	clean	suspends caller process pending events in mask
osix 0x02	exit	—	—	terminates caller & removes it from proctab
osix 0x11	readln	r0: ->buff, r1: cnt	clean*	reads 1 line from KB into buffer, $\leq$ chars
osix 0x12	setit	r0: number of ticks	clean	starts interval timer
osix 0x13	getit	number of ticks to r0	clean*	reads interval timer
osix 0x14	print	r0: ->buff	clean*	prints NULL-term string
osix 0x18	(findbk)	—	dirty	free bank no to r1, proctab last entry to r0
osix 0x1a	(nice)	r1: priority	dirty	proc with priority in r1 downgraded by 1
osix 0x31	(runprg)	r1: cnt, r2: block	—	load & run prog from block in current bank
osix 0x32	(fsysld)	—	dirty	load sys loader to a free bank & run it

\* except for parameter regs

— does not return

Figure 14.9: System calls

When the loader starts to execute, the following two-phase transaction is put into effect.

1. it checks whether the file name at the aforementioned address is the empty string or not. If it is, the loader prints the long string placed by `fsys` from the address `names` using the system call `osix 0x14` (print) and terminates using the system call `osix 0x02` (exit). Otherwise it searches for the same name on its name list. If it does not find it, an error message is printed and the program terminates as before, otherwise the loader proceeds to the next stage.
2. The loader initiates another DMA transfer! This time the system call is `osix 0x31`, again to the OS extension bank (bank 3), but the parameters passed to the OS are the block number on the flash card and the number of bytes to copy. Before issuing the system call, the loader resets the stack to 0. The transfer overwrites the loader and the OS “resets” the process. Specifically the OS initialises the GP registers and the PC with 0, sets the CVZN flags in the PS content to 0 as well, and schedules the process as active.

As a result, the user program just loaded in the bank starts on a clean virtual machine almost as if it were running on a stand-alone von Neumann CdM-8 system after a hardware reset. The difference is that the Platform 2 on top of which the program is running is an extended Platform 2, in which an OS interprets system calls in addition to the processor interpreting the machine instructions.

We are finishing the chapter with the summary of `cocos` system calls in the next section.

## 14.4 `cocos` system calls

The system calls can be made to 3 banks out of 4 (since bank 2 is occupied by the shell rather than the kernel of the OS). Bank 0 is the most critical resource since all I/O interrupt vectors must be set up in that bank. Even though the PS part of the vector may direct the processor to a different bank for ISR, if that were to happen, the code running in the other bank would have a different stack whereas the current PC and PS would be saved on the stack associated with bank 0. This is a complication that can be overcome in software, but there is a definite advantage to accommodating all ISRs in bank 0. Longer processing, which is sometimes required can be relegated to system calls to other banks to save bank-0 space. Still, for simplicity bank 0 is self-contained. It includes *all* `cocos` ISRs and a couple of system calls that manage processes and it does not put out calls to other banks.

Bank 1 is entirely dedicated to OS system calls. Its code supports keyboard and teletype I/O for user applications and the shell. In a way device drivers are split between banks 0 and 1: the former accommodates the drivers' ISRs and the latter the parts that initiate I/O. The latter are implemented as executors of system calls, allowing user processes to request I/O and then (using calls to bank 0) to await its completion.

Bank 2 accommodates an implementation of the shell. No system calls to this bank are supported.

Bank 3 contains OS extensions, currently the file system. It contains file related system-call implementation, including that of the most sophisticated service, program loading.

The table of system calls supported by *cocos* is presented in fig 14.9. Some of the system calls, such as `print` or `exit` are intended for general use, as they are part of the extended Platform 2 seen by a user program. Others, such as `fsys1d` are there for the OS's internal use. In the latter there is no need to support the `osix` call with an L3<sup>1/2</sup> macro, since those instructions are used in a very limited way.

We would like to stress that *cocos* is an open-ended OS, and so the table represents the starting point for the reader's personal exploration, including removal, modification and extension of system calls. For example, launching programs is currently only possible via the shell, since our focus is on the interaction between processes as well as between the processes on the one hand and the I/O on the other. Accordingly, all process control issues are addressed by human intervention via the shell.

It is also quite conceivable to focus instead on the task of making process management more adaptive. The Scheduler does the fast part of the management job by context switching, as quickly as possible, based on priorities and suspensions. One could develop an observation process, whose purpose is to watch other processes and try to learn their behavioural patterns. Some processes may be purely computational, and some might interact with the environment while presenting little load to the processor per unit (wall clock) time. Processes may also depend for progress on other processes' actions. *cocos* can easily be updated to allow a user process to manage priorities of other user processes by utilising the interval timer (which is included in the Coccone platform and is fully supported by *cocos*) to quantify such behaviours. The OS would have to make `proctab` data available to the management process and support a system call for priority changes.

Another possible case study (among tens that can be set up without difficulty) is to explore security properties of the platform. Clearly the current design of the CdM-8 core, even with SP shadowing, does not isolate processes from each other at all reliably. A process only needs to push two bytes on a stack and issue an `rti` to completely change its page, SP and PC content to say nothing about disabling and enabling interrupts. The system calls used by the shell can, without let or hindrance, be used by any other process to cause damage to OS structures. It is possible, and even easy, to extend the CdM-8 Primary Decoder with a *privilege mode*, something that most modern "big" machines have. For example, when the current page number is  $\geq 4$ , the processor could treat `halt` and `rti` as synonymous to `ioi`, and given that real I/O interrupts never utilise (OS) vector 0, and `ioi` if issued programmatically would, use that as an indication that a privileged instruction has been executed. Unlike `osix`, `ioi` sets CVZN from the vector, so the OS will "know" what happened if the vector has a CVZN combination that never occurs in a real system call.

To finish the section let us discuss the system calls a user program is supposed to use, referring to them by their L3<sup>1/2</sup> name.

**await** This instruction suspends the caller pending the events indicated in the event mask in `r0`. More than one event can be waited for, in which case the process will be ready to become active after *all* of them have taken place. For example, the process can wait for an Enter press on the keyboard, `EVENT_EOL`, and the timer counting down to zero, `EVENT_IT`. If the operator presses the Enter key before the time interval has elapsed, effectively it is the timer that will wake the process up, otherwise the keyboard. The mask to be used is the bitwise OR of the two (or the sum, since a unique bit position is used for each event).

**exit** Terminates the process. It is to an individual Platform 2 process the same as `halt` to Platform 2 hardware in that the progression of the process/program stops. The process entry is removed from `proctab` and the memory bank occupied by the program becomes available for program load.

**readln** This is an interesting system call supported by the KB driver. As mentioned earlier, it attempts to preempt the keyboard on behalf of the caller, and if the keyboard is already in use by another process, the executor clears the N flag in the PS before returning to the caller. The caller can `await` `EVENT_EOL`, which is in fact the completion of input of a single line, and after awakening try again to preempt the keyboard. When it is finally successful, the process is guaranteed that the keyboard will not be preempted until a string of characters ending with the Enter keystroke has been typed.

There is an exception to the behaviour described above: the shell can be called in by the KB driver at any time when the operator presses Ctrl/L on the keyboard. The shell will then preempt the keyboard which may at this stage have been used to type part of the input string. The shell has a higher priority

and it does not get suspended even when it is awaiting a command line. After the Enter key has been pressed and the command line entered to the shell, it will process it and will suspend itself, after which interrupts will be re-enabled. Since the last key pressed was Enter, the effect of this will be completion of the previously entered partial string.

**print** The purpose of this call is to print a null-terminated string supplied by the user program as a pointer in **r0** on the teletype. No interrupts take place (since the simulated teletype has instantaneous response, as fast as the clock — one of Logisim’s wildly unrealistic simplification which we have to live with). The system call runs with all interrupts disabled and does not require scheduling at the end as it passes the control back to the caller.

**setit** and **getit** are used to get and set the interval value from/in the Interval Timer’s I/O register. Notice that these actions are asynchronous with respect to the IT clock. This means that the value read from the timer does not represent its state accurately, in particular, the program cannot rely on the fact that the value is greater than zero. Still it gives some indication of when an interrupt may take place, which might be useful. Setting the interval will overwrite the current number of ticks being counted down and (re-)launch the down count.

For completeness, we present the system calls as implemented by Platform 3<sup>1/2</sup>, in macro library format.

```
*print/0
    osix 0x14
*print/1
    ldi      r0,$1
    print
*readln/0
    osix 0x11
*readln/2
    ldi  r0,$1
    ldi  r1,$2
    readln
*await/0
    osix 0x01
*await/1
    ldi  r0,$1
    await
*exit/0
    osix 0x02
*setit/0
    osix 0x12
*getit/0
    osix 0x13
```

The reader will recognise **print** (**osix 0x14**), which prints the NULL-terminated string pointed to by **r0**, and **readln** (**osix 0x11**), which requests a line from the keyboard of a length not exceeding the value in **r1** to be placed from the address given in **r0**. Two more system calls have been added: **await** (**osix 0x01**), which expects the event type to have been placed in **r0**, and which suspends the current process awaiting that event type. It has a straightforward implementation. Finally there is the virtual “halt” instruction, which kills the process that issues it, **exit**, also implemented as deletion from the **proctab** of the entry corresponding to the current process.

**A user program utilising the driver.** Let us now take a look at a “hello world” program<sup>6</sup>.

```
1   asect 0
2
3 while # true
4
```

---

<sup>6</sup>The reader might at this point gasp in surprised exasperation: how come the “hello world” program is presented so close to the end of the book?! We would share the feeling if this were a book on programming. Clearly it is not.

```

5      print msg
6      while
7          readln buf,6
8          stays mi
9              # if we are here, another process is using the KB
10             await EVENT_EOL           # wait until Enter pressed
11         wend                      # and try to grab KB again
12
13     # KB is ours
14     # the human operator is entering text on KB
15     # this program could do something meanwhile
16
17     # assume work is done at this point...
18     await EVENT_EOL #
19     if
20         ldi r1,buf
21         ld r1,r1
22         tst r1
23         is z
24         break        # exit if buf empty
25     fi
26     print resp
27 wend
28     print bye
29     exit
30
31 msg: dc 0x0a,"Enter a string: ",0
32 bye:   dc 0x0a,"Bye!",0x0a,0
33 resp: dc 0x0a,"You entered: "
34 buf:    ds 6
35     end

```

The code looks compact and high-level, thanks to a combination of macro aggregation and OS interpretation, which have together contributed toward a more sophisticated virtual machine, an extended Platform 3<sup>1/2</sup>. Let us see what it does.

The program is an infinite loop which first prints a message inviting the operator to enter a string (Line 5). Immediately it requests line input from the OS (line 7) which may or may not succeed. If another program is currently awaiting line input from the keyboard, the system call on line 7 will fail raising the N flag. If that happens, the program issues an `await` call, which suspends it until the key Enter is pressed on the keyboard (line 10). The other program, which was waiting for keyboard input will at this point have received it, and may not need to request more input for some time, or if our program is running as a higher priority process, it will preempt and the next `readln` may succeed. Of course our program might also have to yield to yet another one running at yet a higher priority, so the success of the `readln` is not guaranteed. Nevertheless, after a finite number of attempts, each of which only costing a few machine instructions, the system call *will* succeed, since any work that other programs do is finite and will be done at some point. That is when the inner while loop terminates and we now await *our* line input to be provided by the OS (Line 15). We print it right away, and continue on the infinite loop, unless the string is empty, in which case we exit (line 29)

Notice that other work could be done by the program between lines 11 and 18 without any negative effect on the timing. That is, while the human operator is typing on the keyboard (an eternity in the world of machine instructions) the program can busy itself with some computation which will not, thanks to the principle of process transparency honoured by the OS, affect the functioning of the driver in any way. What we are seeing here is the classical overlap of computation with I/O, which is a major advantage of an OS-enhanced Platform 2.