Microprocessor Systems

CSU23021 (aka CS3D2)

Mike Brady
brady@cs.tcd.ie
O'Reilly Institute Room G.41

Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

Microprocessor Systems

CSU23021 (aka CS3D2)

Mike Brady
brady@cs.tcd.ie
O'Reilly Institute Room G.41

Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

CSU23021 Timetable

Day	9:00—10:00	10:00—11:00	11:00—12:00	12:00—13:00	13:00—14:00	14:00—15:00	15:00—16:00	16:00—17:00	17:00—18:00
Monday		Lecture *Live Online*					Lecture *Live Online* Engineers Only		
Tuesday									
Wednesday							Lab Session *Live Online*		
Thursday			Lecture *Live Online*						
Friday			Lecture *Live Online*						

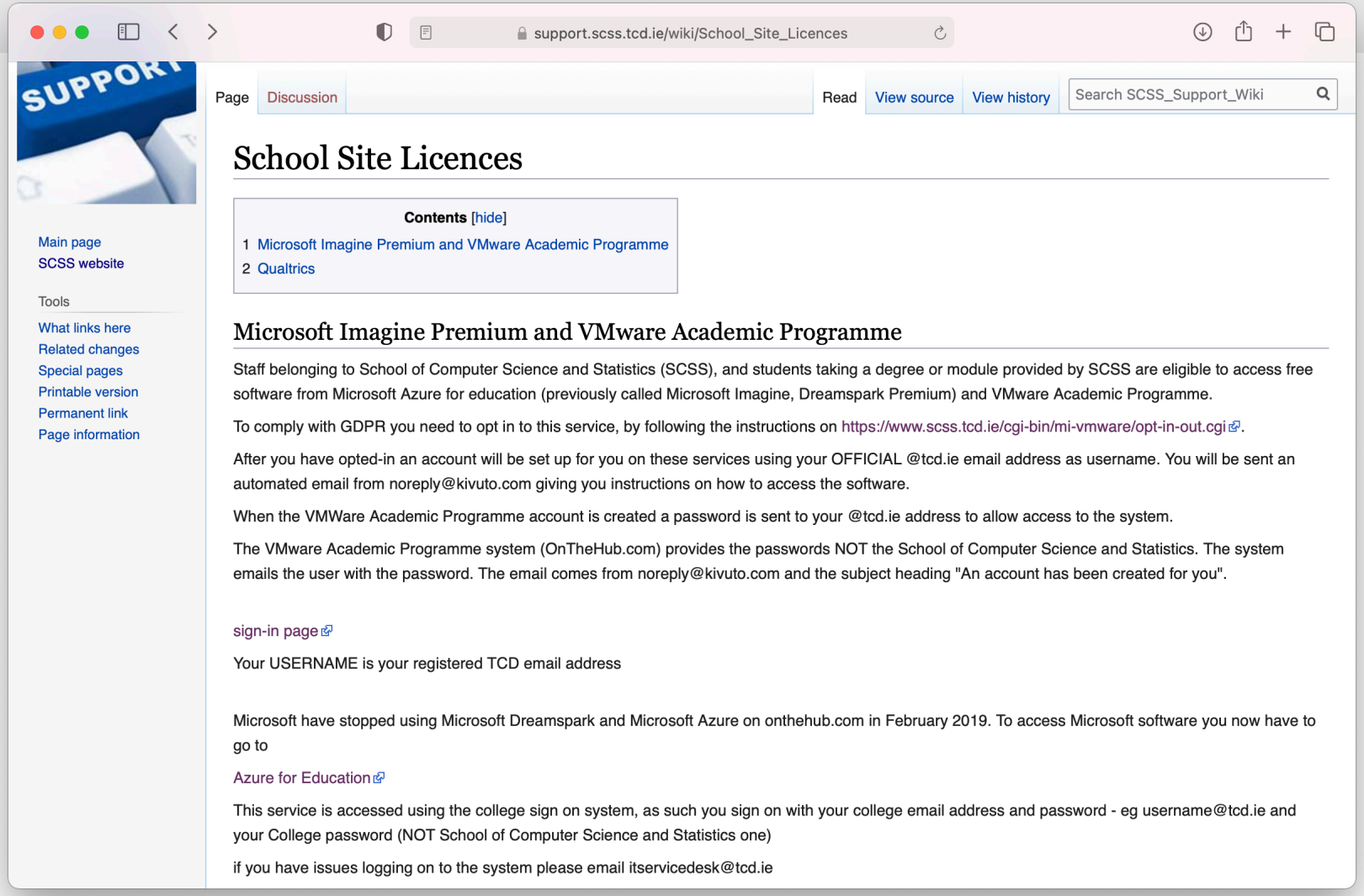
Software

- Keil Development System — Free "Lite" Edition
  - Runs on Windows only
- For Mac users — Install a Windows VM (Intel Macs only)
  - VMWare Fusion
  - Windows

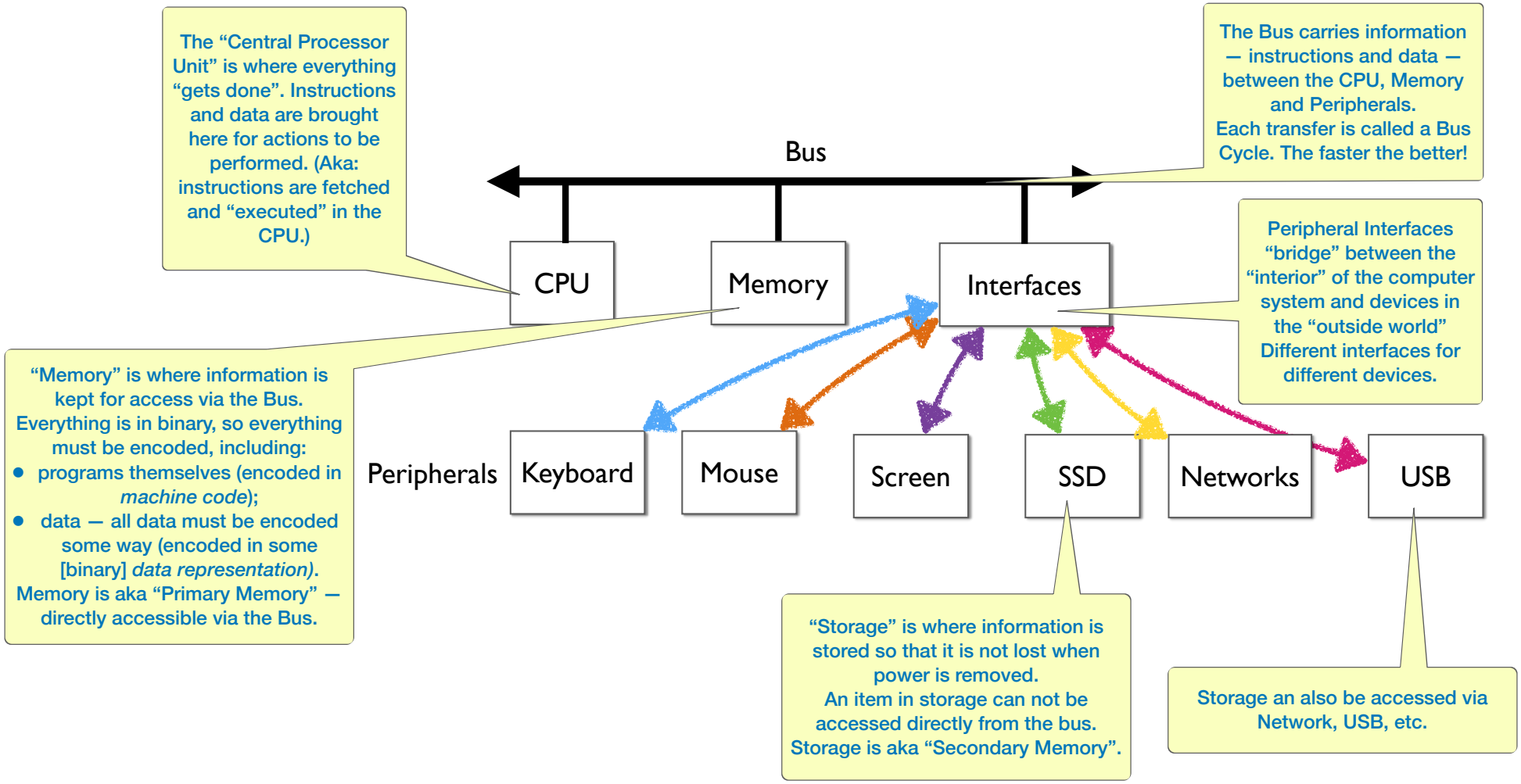
Software

- Keil SDK available on Blackboard — it's a slightly older version with a more suitable license.
- See http://support.scss.tcd.ie/index.php/School_Site_Licences for information and access to some Microsoft developer tools and VMware products.
- VMware products allow you to run "virtual machines":
  - On a Mac (Intel only, not Apple Silicon):
    VMware Fusion allows you to run Windows, Linux, FreeBSD, etc.
  - On a Mac (Apple Silicon M1)
    Parallels Preview edition and Windows on ARM via Windows Insider
  - On a PC:
    VMware allows you to run Linux, FreeBSD, etc.
- VMware products are free to students of SCSS modules, including CSU23021.
  - Engineers — check that your SCSS login credentials work!

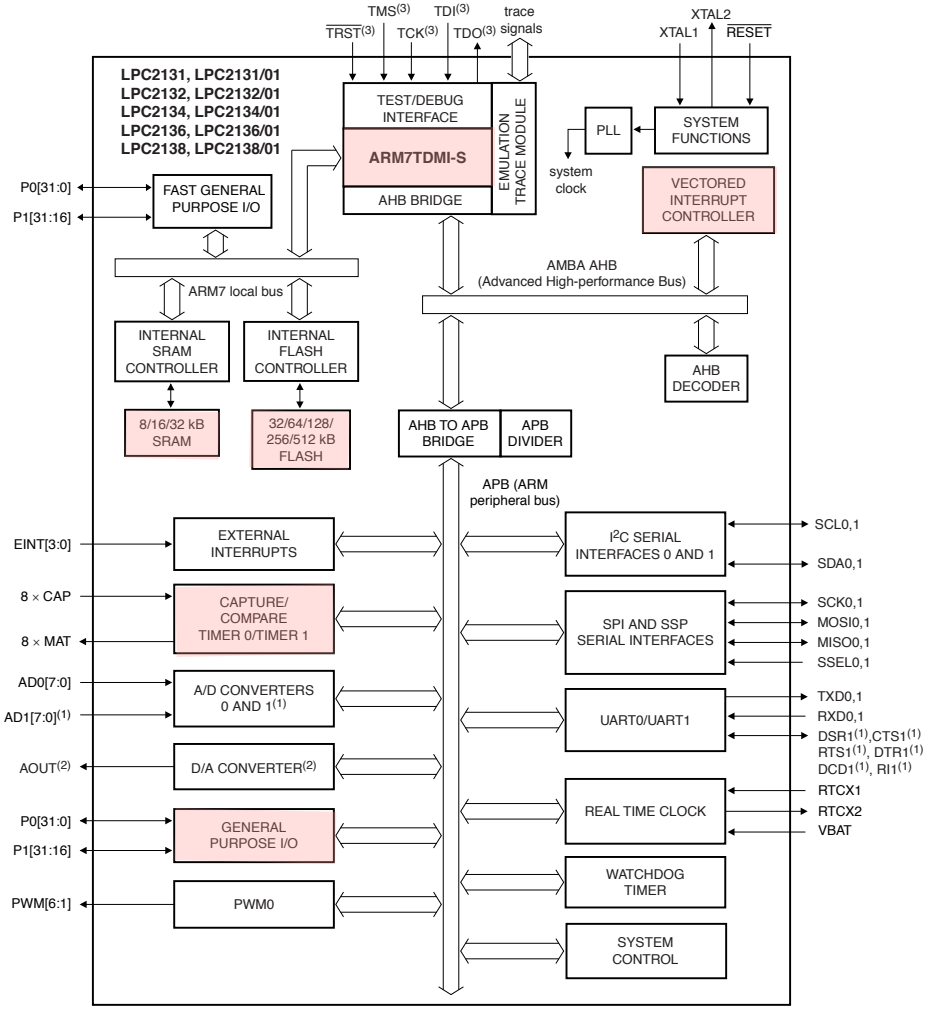
Software


support.scss.tcd.ie/wiki/School_Site_Licences
SUPPORT
Page
Discussion
Read
View source
View history
Search SCSS_Support_Wiki
Main page
SCSS website
Tools
What links here
Related changes
Special pages
Printable version
Permanent link
Page information
School Site Licences
Contents [hide]
1 Microsoft Imagine Premium and VMware Academic Programme
2 Qualtrics
Microsoft Imagine Premium and VMware Academic Programme
Staff belonging to School of Computer Science and Statistics (SCSS), and students taking a degree or module provided by SCSS are eligible to access free software from Microsoft Azure for education (previously called Microsoft Imagine, Dreamspark Premium) and VMware Academic Programme.
To comply with GDPR you need to opt in to this service, by following the instructions on https://www.scss.tcd.ie/cgi-bin/mi-vmware/opt-in-out.cgi.
After you have opted-in an account will be set up for you on these services using your OFFICIAL @tcd.ie email address as username. You will be sent an automated email from noreply@kivuto.com giving you instructions on how to access the software.
When the VMWare Academic Programme account is created a password is sent to your @tcd.ie address to allow access to the system.
The VMware Academic Programme system (OnTheHub.com) provides the passwords NOT the School of Computer Science and Statistics. The system emails the user with the password. The email comes from noreply@kivuto.com and the subject heading "An account has been created for you".
sign-in page
Your USERNAME is your registered TCD email address
Microsoft have stopped using Microsoft Dreamspark and Microsoft Azure on onthehub.com in February 2019. To access Microsoft software you now have to go to
Azure for Education
This service is accessed using the college sign on system, as such you sign on with your college email address and password - eg username@tcd.ie and your College password (NOT School of Computer Science and Statistics one)
if you have issues logging on to the system please email itservicedesk@tcd.ie

Architecture of a Typical Computer



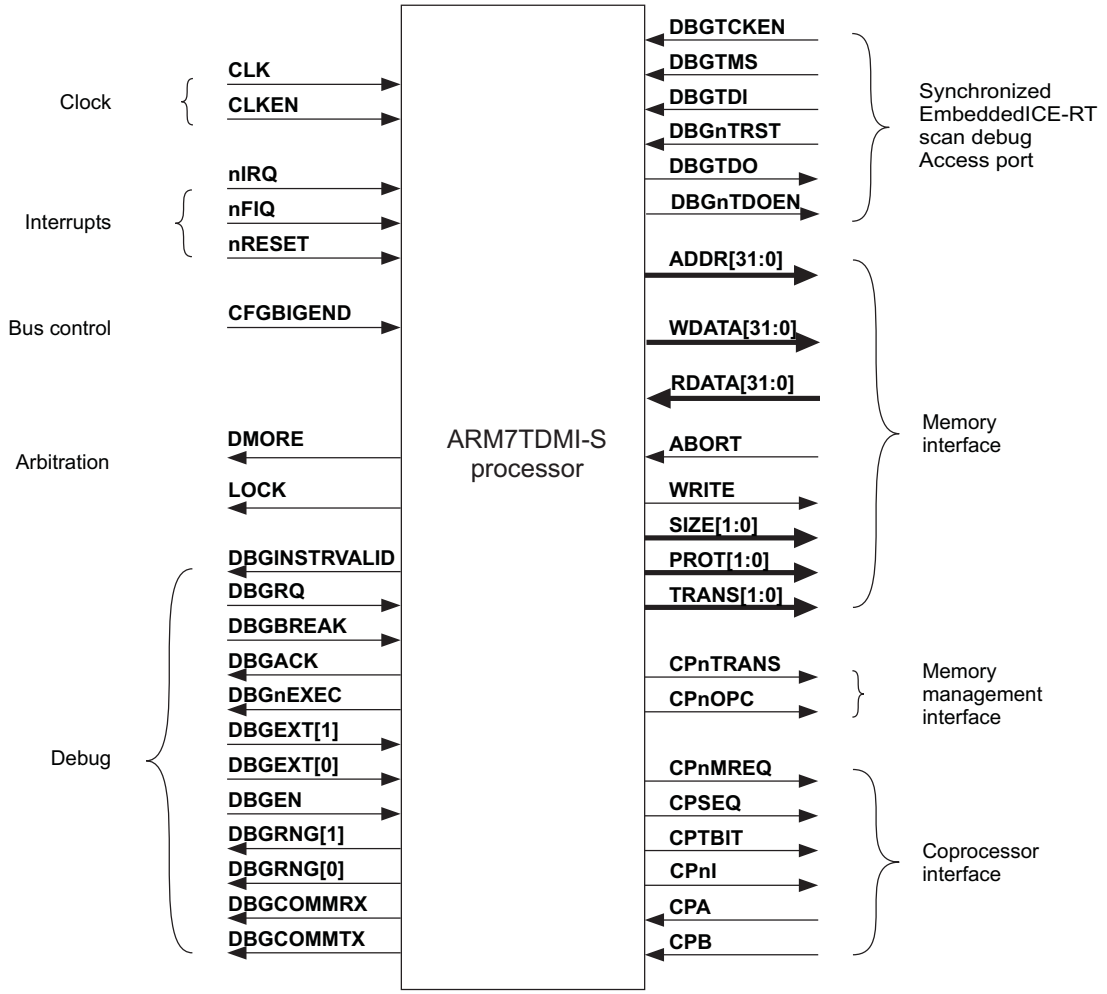
The LPC2138 Block Diagram



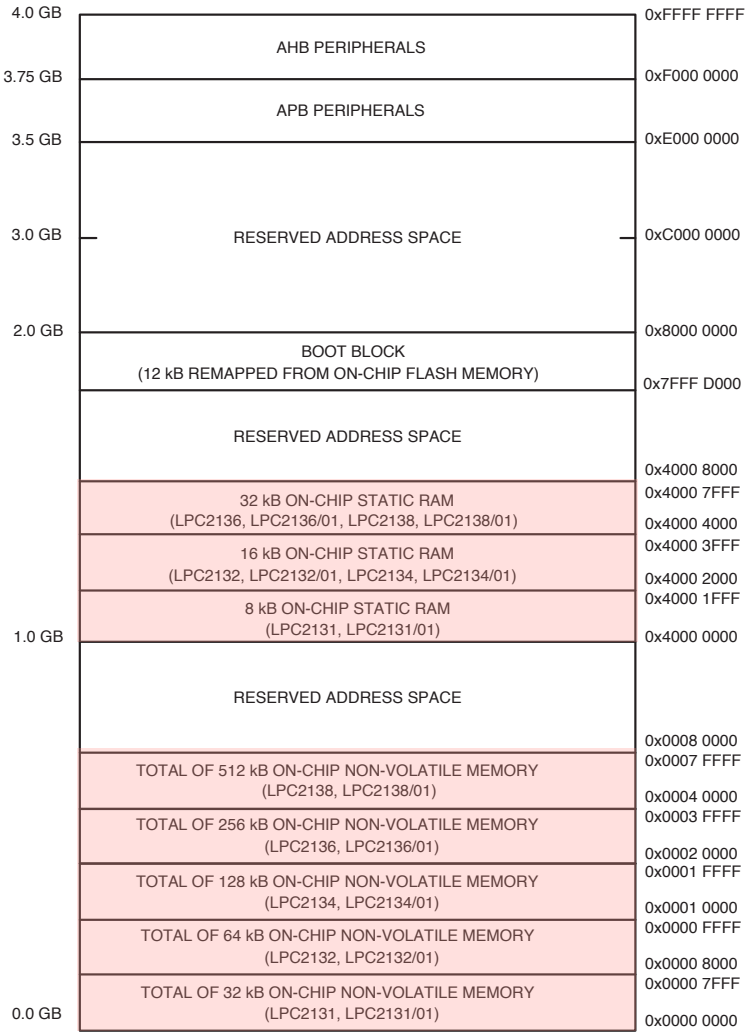
LPC2138 Single-chip "Microcontroller"

- ARM7TDMI-S
  - On-chip Flash Program Memory (512 kB). 100,000 erase/writes.
  - On-chip Static RAM (32 kB).
  - Lots of peripheral interfaces.
  - Vectored Interrupt Controller.
  - Actual Part No: LPC2138FBD64/01.
  - Enhanced UARTs, ADC, fast I/O, Brown-Out-Detection.

The Processor Core



Memory Map of the LPC2138 & Family



Binary Representation

- Everything in the machine must be in binary
  - Programs
    - Programs are sequences of machine-code instructions stored in memory
    - Instructions are binary patterns that directly cause the hardware of the machine to give event to the "semantics" of the instruction.
    - Every instruction in the ARM is a 32-bit instruction
  - Data
    - All data is stored in binary using some kind of *data representation.*

Date Representation of Numbers

- Integers
- Closest mathematical concept: integers but with a finite range.
- Floating Point Numbers
  - Closest mathematical concept: reals and rationals.
    - Reals because they look like reals,
    - Rationals because they don't have infinite precision and can't represent irrational numbers (e.g. π or $\sqrt{2}$) faithfully.
    - Represented as fractions.

Integer Representation

- Straight Binary with the precision of the computer: 8, 16, 32, 64, 128… bits. Thus the ranges could be:
  - 0 – 2^8-1 i.e. 0 – 255
  - 0 – 2^{16}-1 i.e. 0 – 65,535
  - 0 – 2^{32}-1 i.e. 0 – 4,294,967,295
  - 0 – 2^{64}-1 i.e. 0 – 18,446,744,073,709,551,615
  - 0 – 2^{128}-1 i.e. 0 – 340,282,366,920,938,463,463,374,607,431,768,211,455

Integer Representation

- Signed Binary
  - Twos Complement
  - Negative numbers stored in 2's complement form, with MS bit = 1.
- Sign + Magnitude
  - One bit (usually MS bit) indicates sign, remaining bits magnitude.

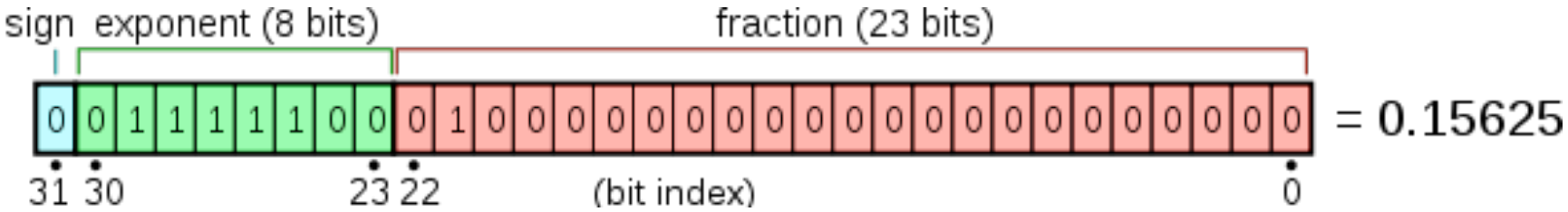
Floating Point Representation

- A floating point number is a number of fixed precision
  - Four items to store
    - The digits (the *significand* or *mantissa*)
    - The sign of the number
    - The exponent
    - The sign of the exponent
- Lots of different representation schemes
- Most popular nowadays is based on IEEE standard 754

IEEE 754

- IEEE = Institute of Electrical and Electronic Engineers
  - US based, ≠ IEE, which is UK based.
- Standard 754 is a "Standard for Floating-Point Arithmetic"
  - Revised periodically
  - Latest is 754-2019

Example – 32 bit "binary" form



- Sometimes called "single precision IEEE"
- MS Bit is (almost) always taken to be 1 (why?), thus no need to store it.

Five Basic Formats

- Three "binary" formats
  - 32 bit: 1 sign bit, 23 mantissa bits, 8 exponent bits (1/23/8)
  - 64 bit: 1/52/11
  - 128 bit: 1/112/15
- Two "decimal" formats

Single Precision IEEE

- The value V represented by the word may be determined as follows:
  - If E=255 and F is nonzero, then V=NaN ("Not a number")
  - If E=255 and F is zero and S is 1, then V=-Infinity
  - If E=255 and F is zero and S is 0, then V=Infinity
  - If 0<E<255 then V=(-1)**S * 2 ** (E-127) * (1.F) where "1.F" is intended to represent the binary number created by prefixing F with an implicit leading 1 and a binary point.
  - If E=0 and F is nonzero, then V=(-1)**S * 2 ** (-126) * (0.F) These are "unnormalized" values.
  - If E=0 and F is zero and S is 1, then V=-0
  - If E=0 and F is zero and S is 0, then V=0

Integers, Characters, Addresses, etc.

- The binary calculation machinery for integers – adders, comparators, shifters, etc. can be used for:
  - Whole numbers,
  - Addresses
  - Binary fixed point arithmetic
  - Character Representations

Assembly Language

- Assembly Language is a formal language in which programs are specified in statements that are [mostly] assembled into machine code.
- Statements come in three broad categories:
  - Comments: these do not generate code or affect how it is generated
  - Instructions: these are converted into machine-code instructions, generally one machine-code instruction per assembly language instruction.
  - Directives: these don't generate machine-code instructions, but can generate data and generally affect the way assembly is carried out.

Assembler

- The assembler is part of the Keil Development System, sometimes called a System Development Kit, an SDK*.
- The SDK is already tailored to the System on a Chip (SoC) device — in our case the LPC2138. Using this tailored information, it can relieve you of some of the tedious aspects of low-level programming.
  - It has device-specific information about where read-only memory and where read-write memory starts and stops. This is used by the AREA directives (see later).

Labels

- Very important part of any assembler.
- A *label* is an identifier that has a value.
- Labels are always defined by putting them in column 1.
- Typically, a label is automatically given a value equal to the value of the *location counter* of the statement it is defined in.
- More about this later when we discuss Two-Pass Assemblers

The AREA directive

- This allows you to break your program up into regions with particular names and properties:
- AREA <name>,<attribute>[,<attribute>…]
- Attributes include CODE, DATA, READONLY, READWRITE, …
- The SDK knows the memory map of the SoC, hence knows where to place AREAs depending on whether they are READONLY or READWRITE.
  - READONLY areas will be located where there is Read-Only Memory (ROM).
  - READWRITE areas will be located where there is Read/Write memory (RAM).

Equate Statements

- An EQU statement simply defines the label to have the value of the expression:
- <label> EQU <expression>
- The value of the <expression> must be calculable at assembly time.

DCD, DCW, DCB

- These *directives* do two things:
- They reserve memory space, (e.g. for variables).
- They give those memory spaces initial binary values.
  - DCB — reserve bytes
  - DCW — reserve half-words (16 bits), aligned.
  - DCD — reserve words (32 bits), aligned.
- The values must be calculable at assembly time.
- Labels can also be used with DCB/W/Ds
- Note: if you want a variable to be really changeable, make sure it's in a READWRITE area!

A "Two-Pass" Assembler

- Assembly language is translated into machine code by an Assembler.
  - A compiler translates a high-level language like C into machine code,
  - An assembler does the same job for assembly language programs.
- Assemblers normally take two passes over the text of the program. (Sometimes three passes for assemblers with 'macro' facilities.)
- First pass evaluates all labels — all labels are given their values at the end of Pass 1.
- Second pass generates code.

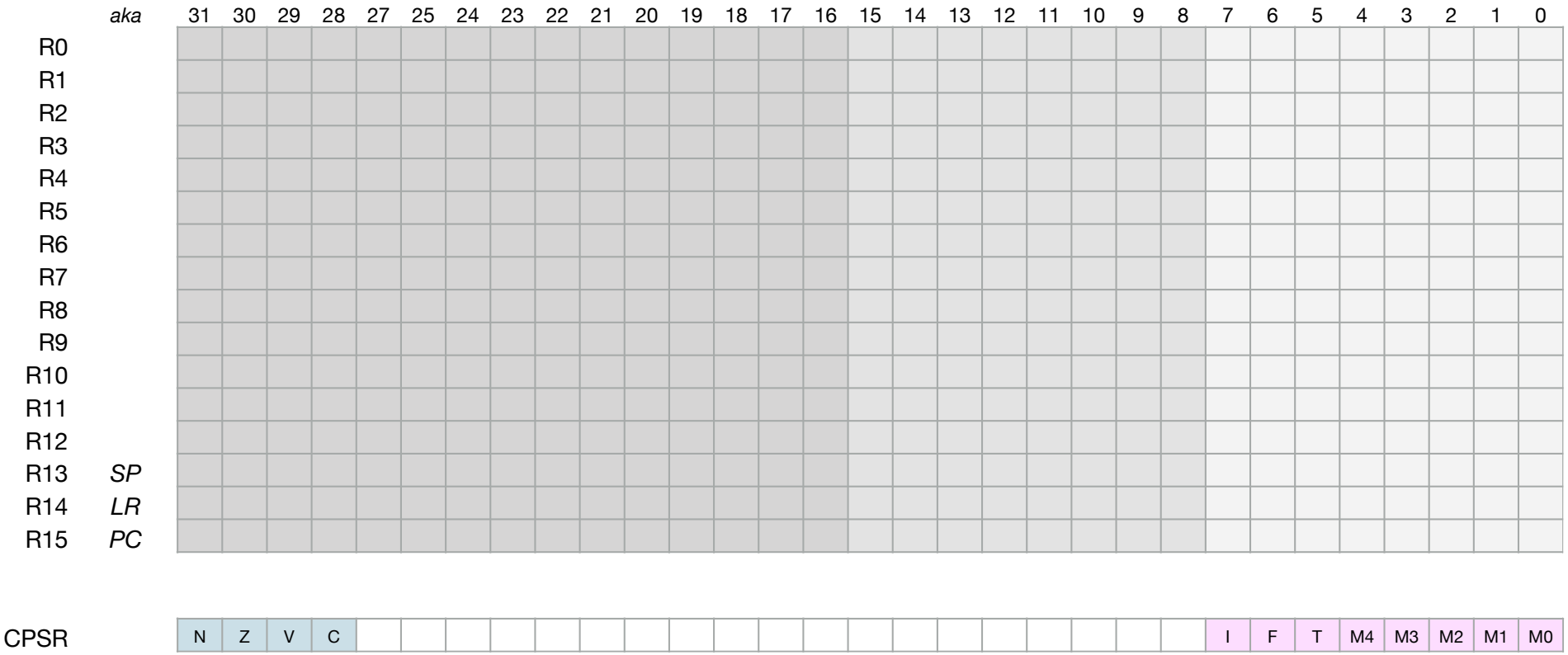
Lab 2 — Running & Testing Programs

- Lab 1 was installing Keil, etc.
- Today, write a simple program to add five 32-bit signed integers together, store the result in a 32-bit variable called RES.
- Use the debugger:
  - Single Stepping
  - Breakpoints
  - Examining Memory
- Make sure the program is correct!

Controlling Program Flow

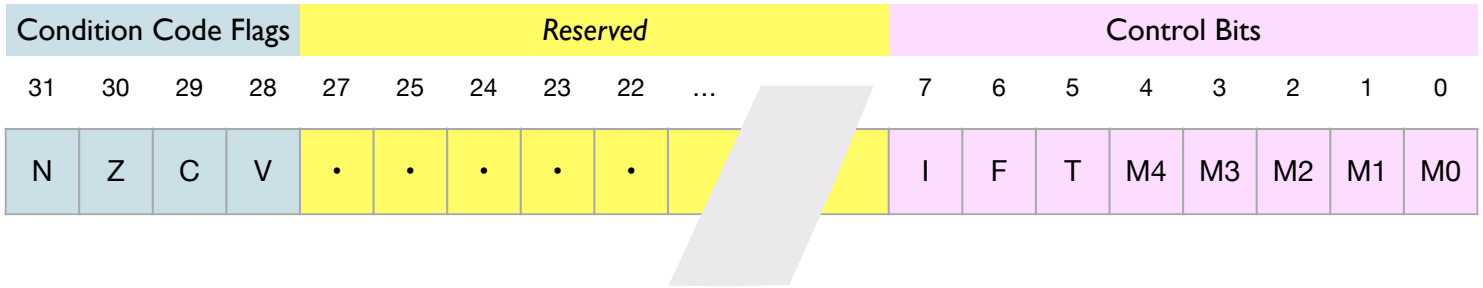
- High-Level Language constructs like `if-then-else`, `while`, `for`, `switch` and iteration, are all implemented using the same hardware in the CPU — hardware for doing conditional program execution.
- The core idea is to make it possible to make program execution data-sensitive, i.e. that the flow of execution of instructions can be made to vary depending on the values of items of data.
- Typically*, this is done with three components:
  - Condition Bits, often grouped into a Condition Code Register or in the case of the ARM, a Current Processor Status Register (CPSR).
  - Instructions that affect the Condition Bits
  - Instructions that are affected by the Condition Bits

Registers in the ARM7 TDMI



R15 is the Program Counter (PC); R14 is used as a Link Register(LR); R13 can be used as a Stack Pointer (SP).
"aka" means "Also Known As"

Current Program Status Register (CPSR)



- N — Negative
- Z — Zero
- V — oVerflow
- C — Carry/Borrow
- I — IRQ Disable
- F — FIQ Disable
- T — Thumb Enable
- M4-M0 — Mode

Condition Tests ARM7

Suffix	Flags	Meaning
EQ	Z set	equal
NE	Z clear	not equal
CS	C set	unsigned higher or same
CC	C clear	unsigned lower
MI	N set	negative
PL	N clear	positive or zero
VS	V set	overflow
VC	V clear	no overflow
HI	C set and Z clear	unsigned higher
LS	C clear or Z set	unsigned lower or same
GE	N equals V	greater or equal
LT	N not equal to V	less than
GT	Z clear AND (N equals V)	greater than
LE	Z set OR (N not equal to V)	less than or equal
AL	(ignored)	always

Wait, there's more…

- Remember: if a bit is "Set" it means it is equal to 1; "Clear" means equal to 0.

Source: ARM Document ARM DDI 0029E pp 4-5

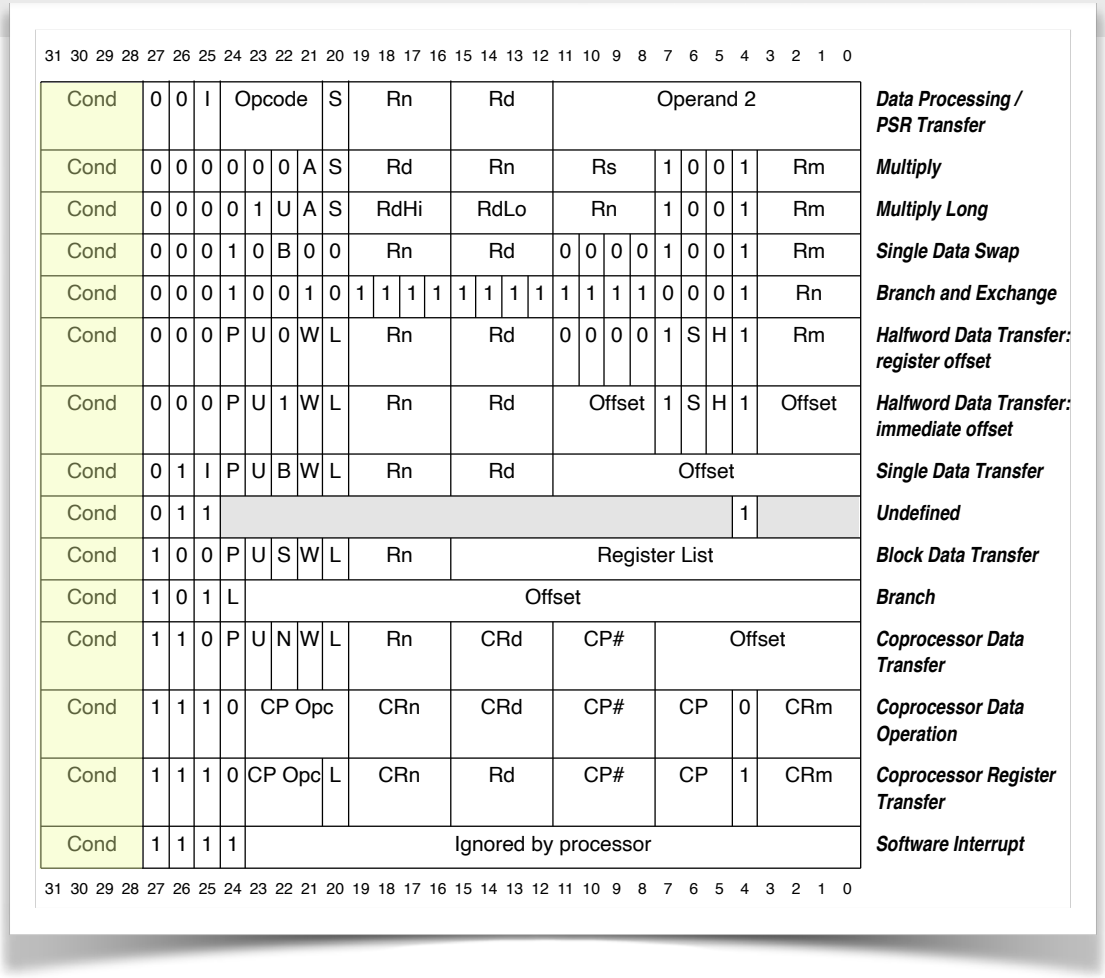
Example of if-then-else

```
            ldr     r1,=50      ; get the value 50 into r1
            cmp     r0,r1       ; subtract r1 from r0, set Conditions Flags but don't store result
            bcs     xismore     ; branch if no borrow occurred. NB in this case C will be set!*
            <then>              ; here the code to be executed if the condition was true is executed
            b       onward
xismore     <else>
onward      …
```

- Implement if-then-else, e.g. if X < 50 then <then> else <else> …
  - Assuming X is an integer — BTW, is X signed or unsigned? Assume it's unsigned, in R0.

- Important Note: on *subtraction* or *comparison*, the C bit will be clear if a borrow occurred and set if no borrow was needed. This is somewhat counterintuitive, so watch out!

Instruction Encoding Summary



- All instructions on the ARM are 32 bits wide. This is how they are encoded. Notice that four bits are reserved for *Cond*

Source: ARM Document ARM DDI 0029E pp 4-2

Condition Tests ARM7

Code	Suffix	Flags	Meaning
0000	EQ	Z set	equal
0001	NE	Z clear	not equal
0010	CS	C set	unsigned higher or same
0011	CC	C clear	unsigned lower
0100	MI	N set	negative
0101	PL	N clear	positive or zero
0110	VS	V set	overflow
0111	VC	V clear	no overflow
1000	HI	C set and Z clear	unsigned higher
1001	LS	C clear or Z set	unsigned lower or same
1010	GE	N equals V	greater or equal
1011	LT	N not equal to V	less than
1100	GT	Z clear AND (N equals V)	greater than
1101	LE	Z set OR (N not equal to V)	less than or equal
1110	AL	(ignored)	always

- Remember: if a bit is "Set" it means it is equal to 1; "Clear" means equal to 0.

Source: ARM Document ARM DDI 0029E pp 4-5

Conditional Instruction Execution

- The ARM instruction set is very unusual in that it, apart from the usual conditional branch instructions, almost all its ordinary instructions are conditional too.
- That, is, most instructions will only actually be performed if the condition expressed in the four-bit *Cond* field is true.
- This can simplify some common programming situations.
- It can also *smooth the flow* of instructions into the CPU. This could potentially be important for improving the speed of the computer.
- However, such is the pressure for alternative uses of the four bits, and such have been the improvements in compiler technology, in more recent ARM instruction sets, most of the conditional instruction functionality, apart from branches, has been dropped.

Subroutines

- A subroutine is simply a block of code that performs a task based on some arguments and optionally returns a result.
  - Functions and methods in HLLs are generally implemented as subroutines in machine code.
    - An exception to this is if the function is declare to be "inline", in which case the code to implement the function is copied into everywhere the function is called. This avoids the overhead of calling and returning from a subroutine, but the size of the program may increase due to number of times the function code is replicated.
  - Subroutines are normally used to encapsulate some functionality that is generally useful.

Subroutines

- Typically, the first instruction of a subroutine has a label, so you call a subroutine by branching to its first instruction. But wait…
  - The problem is, if a subroutine is generally useful – and therefore called from different parts of a program – how do you go back to where it was called from?

Subroutines

- The solution is to store the address of the next instruction before going to the first instruction of the subroutine.
  - This is done by a specialised branch instruction, the `BL` (Branch and Link) instruction. It puts it in `R14`. E.g:

    ```
    BL FOO
    ```

  - Then, at the end of the subroutine, branch back to the address stored in `R14`.
    This done by:

    ```
    BX R14
    ```

    or, simply,

    ```
    MOV PC,LR.
    ```

  - There are other ways too — see later.

Well-Behaved Subroutines

- If subroutines are to be generally useful, they must be safe to use and "well behaved".
- In particular, they must not damage the *context* of the calling program.
  - The context is the environment in which a program is executing, especially
    - Register Values of registers being used.
    - Memory Contents of memory locations being used.
  - If the writer of the subroutine has no knowledge of the caller's context, then total precautions must be taken:
    - Any register whose contents are to be changed inside the subroutine must be saved beforehand and restored before the subroutine returns, so the caller sees the same values after the subroutine returns.
    - Only *definitely safe* memory locations can be used.

The Stack

- The ARM (and virtually all CPUs) maintains a "stack" for the cheap and automatic allocation and deallocation of memory. It's a LIFO arrangement.
- The stack is a place from which memory may be safely and "cheaply" allocated and deallocated.
- By convention:
  - R13 is used as "the Stack Pointer (SP); it points to "The Stack" i.e. the stack normally used by the system,
  - The Stack is of the *Full Descending* type, i.e:
    - *Full*: the SP points to the top element in the stack, not the next free space,
    - *Descending*: the Stack grows downwards.
- Use the stack by "pushing" data into it and "popping" information from it.
- Use of the stack must be balanced — i.e. amount pushed must equal amount popped.

Lab 3

- Write a main program and a subroutine.
- The main program should call the subroutine called ARRADD.
- The subroutine should form the sum of an array of 32-bit signed numbers and return the result in R1.
  - The first number is the number of numbers in the array. The array should be defined with DCDs
  - For the subroutine, the array should be pointed to by R0. Return the result in R1.
  - The main program should store the result in a memory variable called SUM.
  - Set the C bit to 0 in the CPSR if the result is okay.
  - Set the C bit to 1 if the result is not okay.

Stack Frames & Subroutine *Instances*

- The Stack is typically used for automatic memory management in a High Level Language (HLL) like C/C++/Java etc.
  - When a subroutine (or function or method) is called, space is allocated in the stack for some or all of the following:*
    - Return Address, Return Value (cf. C), Parameters, Saved Registers.
  - When a variable (e.g. an int — 4 bytes) comes into scope i.e. when it is defined, space is allocated for it on the stack by decreasing the SP by the right amount — 4 in this example.
  - When the variable goes out of scope, its space is deallocated simply by increasing the SP by the right amount — 4 in this example.
  - The combination of all these items is often called a Stack Frame — a bundle of pieces of information in the stack associated with a *single invocation* or *instance* of the subroutine.
- You could have many separate instances of a subroutine, all sharing the same (read-only) machine code instructions but each with their own individual stack frames.

*An optimising compiler may eliminate some of these to improve speed and space usage.

Useful Instructions

- The STore Multiple instruction & The LoaD Multiple instructions
  - e.g.

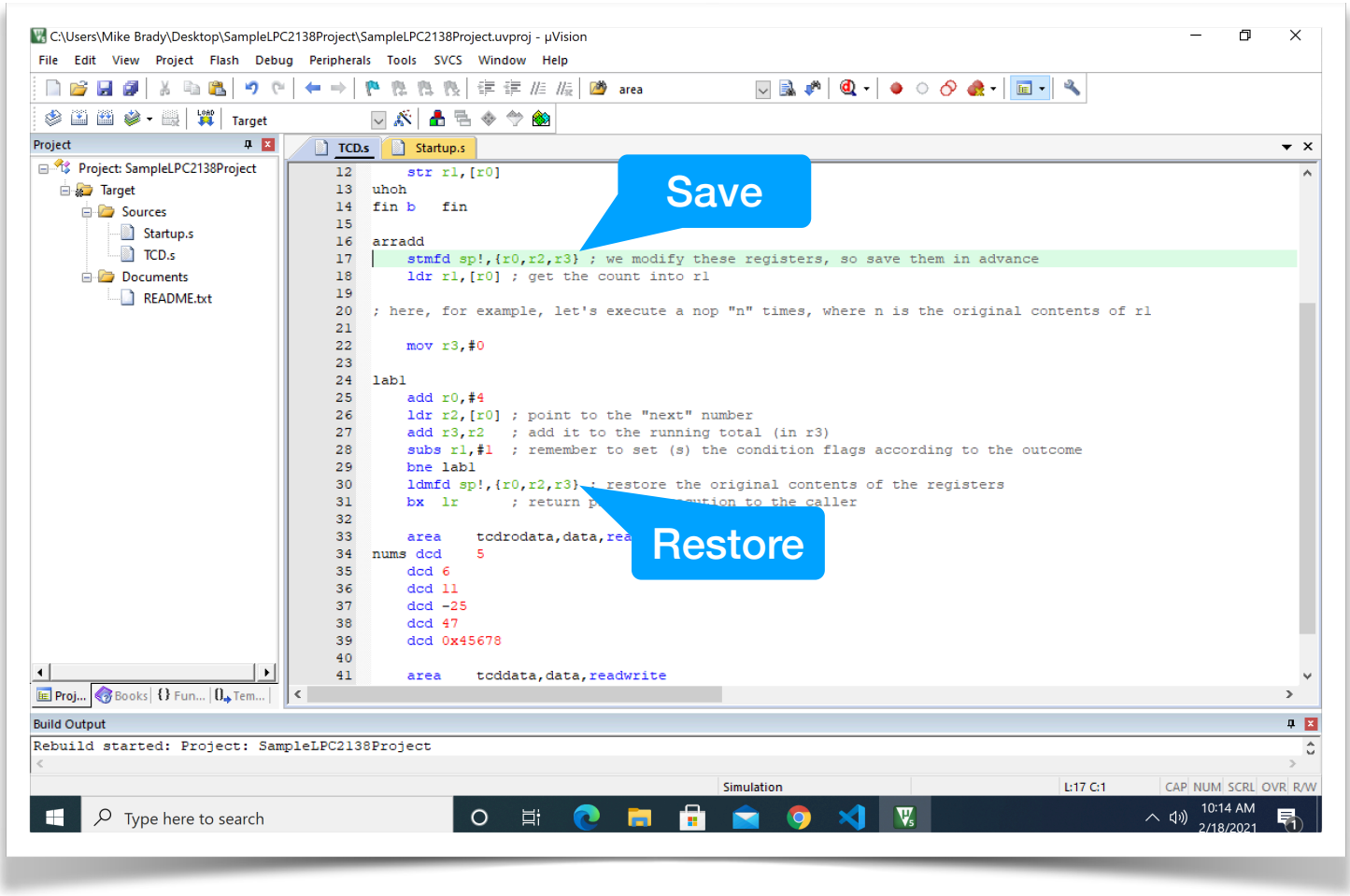
```
stmfd sp!,{r0,r2-r6}.   ; push copies of the contents of r0,r2,r3,r4,r5 & r6 onto the stack

…

ldmfd sp!,{r6,r2-r5,r0} ; copy contents of the stack into the registers listed
```

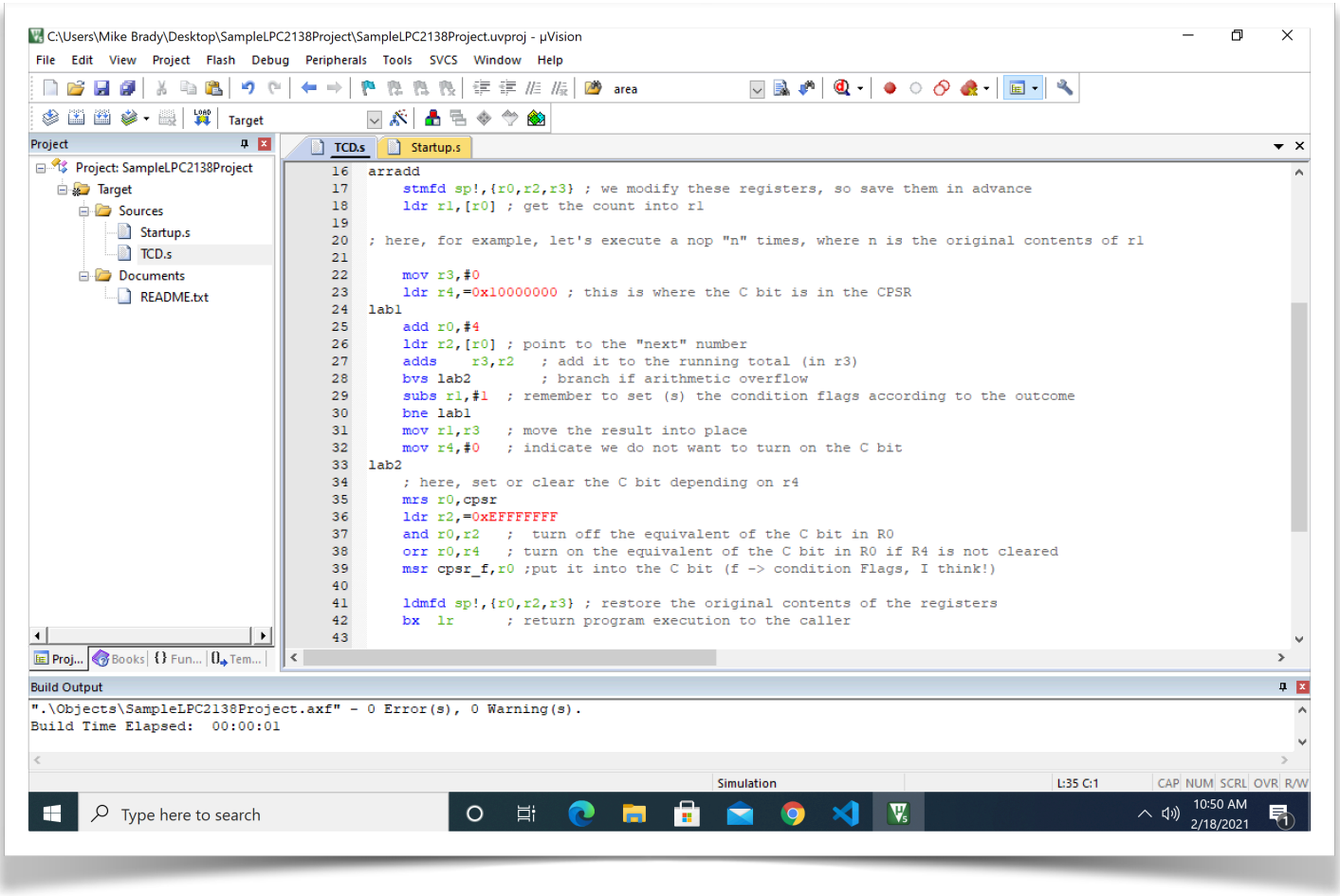
  - Note: the order in which the registers are pushed/popped is fixed by the machine. It is independent of how they are listed in the instruction.
  - STMFD means STore Multiple registers on a Full Descending stack. Similarly LDMFD.
  - The ! means automatically update the value of the register being used as a pointer to accommodate the change in the stack.
  - These instructions are expensive — each register requires a memory write or read.
  - (We need to talk about *Operand Specification*, aka Addressing Modes.)

Let's apply this to the Lab3 Subroutine



Note: the program still isn't finished!

More Complete, Not Tested!



GCC Optimisation Levels (Information Only)

- -O0 Reduce compilation time and make debugging produce the expected results. This is the default.
- -O1 Optimize. Optimizing compilation takes somewhat more time, and a lot more memory for a large function.
- -O2 Optimize even more. GCC performs nearly all supported optimizations that do not involve a space-speed tradeoff. The compiler does not perform loop unrolling or function inlining when you specify -O2. As compared to -O, this option increases both compilation time and the performance of the generated code.
- -O3 Optimize yet more. -O3 turns on all optimizations specified by -O2 and also turns on the -finline-functions, -funswitch-loops, -fpredictive-commoning, -fgcse-after-reload and -ftree-vectorize options.
- -Os Optimize for size. -Os enables all -O2 optimizations that do not typically increase code size. It also performs further optimizations designed to reduce code size.

-Og is another setting to facilitate debugging.
See also https://gcc.gnu.org/onlinedocs/gcc-3.4.6/gcc/Optimize-Options.html

Guide to Operand Specification

- Instruction execution requires hardware, obviously. E.g. ADD needs an adder, etc.
- Operand Specification also needs hardware, thus will be limited.
- Two Broad Aspects to Operand Specification:
  - What you know about it at Assembly Time
  - What modifications you can perform on it before or after it it used at run time.
    - The ARM architecture has a rich set of modification facilities.
- Slight complication: the ARM can not directly accommodate "bulky" operands in an instruction — i.e. operands that take many bits to represent — because there isn't enough space in the 32-bit instruction. There are a number of elegant and inelegant tricks to work around this.

Operand Specification

- Three broad categories:
  - *Immediate*, for when you know the value at assembly time
  - *Direct*, for when you know the location.
  - *Indirect*, for when you (or the assembler) know how to calculate the location at run time.

- In principle, the way you specify an operand depends on *what* you know about it and *when* you know it:
  - If you know the value when assembling the program, use an immediate mode, designated with a #.
  - Otherwise, if you know where an operand will be when the program is running, use a direct mode.
  - Otherwise, if you (or the assembler) know how to calculate where the operand might be at runtime, use an indirect addressing mode.

Operand Specification

- If you know the *value* when assembling the program, use an *immediate* mode.
  - Immediate modes are always designated by a preceding # symbol. The value can be given as any expression that can be evaluated to a 32-bit expression as assembly time. It may then be truncated or modified by the assembler to fit into an instruction, if possible*.
  - If the operand is too bulky, the ARM assembler converts the operand and addressing mode into an indirect mode — see later.

*Recall that the ARM has a problem representing "bulky" operands.

Operand Specification

- If you know where an operand will be when the program is running, use a *direct* mode.
  - E.g. if you know that an operand will be in a specific register, you can simply specify that.
  - (In other (non-RISC) architectures, if you know where an operand is in memory, you can specify its memory location directly. Why can't you do this on the ARM?)

Operand Specification

- If you (or the assembler) know how to *calculate where* the operand it might be at runtime, you use an *indirect* addressing mode.
  - An indirect addressing mode will specify one or more registers to be used as pointers, designated by the use of square brackets, e.g. [r0].
  - The register to be used as a pointer can include R15, the Program Counter.

Modes and Examples

Name	Alternative Name	Example
Register to Register	Register Direct	mov r0,r1
Absolute	Memory Direct	ldr r0,mem
Literal	Immediate	mov r0,#15
Indexed, base	Register Indirect	ldr r0,[r1]
Pre-indexed, base with displacement	Register Indirect with Offset	ldr r0,[r1, #4]
Pre-indexed, autoindexing	Register Indirect with Pre-Increment	ldr r0,[r1, #4]!
Post-indexed, autoindexing	Register Indirect with Post-Increment	ldr r0,[r1] ,#4
Double Register Indirect	Register Indirect Indexed	ldr r0,[r1, r2]
Double Register Indirect with Scaling	Register Indirect Indexed with Scaling	ldr r0,[r1, r2, lsl #2]
Program Counter Relative	PC Relative	ldr r0,[pc, #offset]

Source: https://www.cs.uregina.ca/Links/class-info/301/ARM-addressing/lecture.html

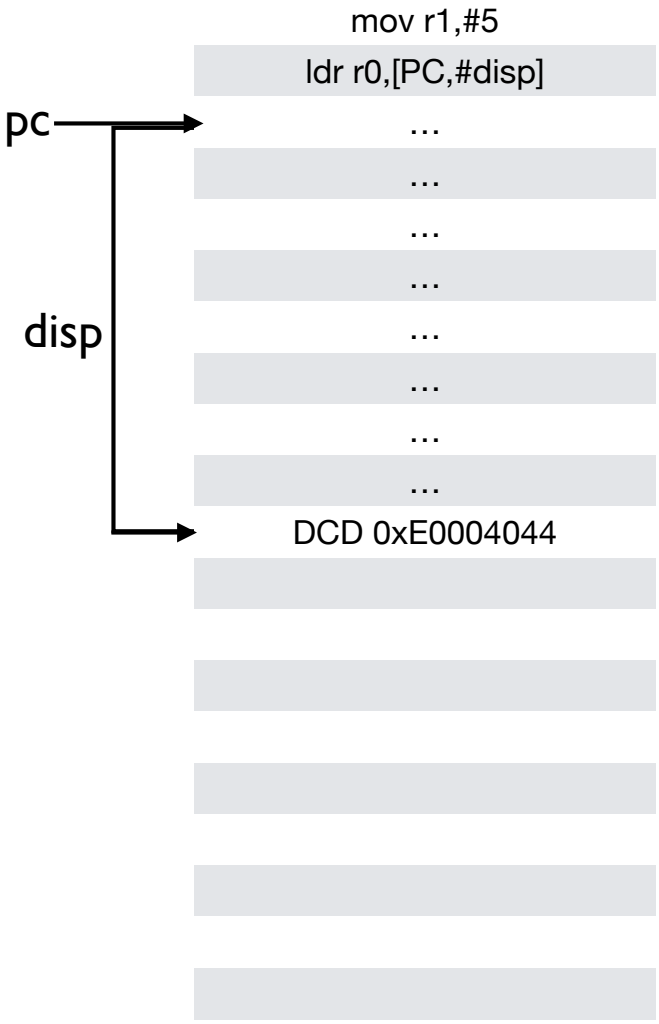
Notes

- The availability of addressing modes, the range of possible values of literals/ immediates and the type and range of scaling varies from instruction type to instruction types.
- For the full instruction-type-by-instruction-type detail, please refer to the ARM Technical Manual, e.g. ARM Document ARM DDI 0029E pp 4-2 onwards.

Yeah, but…

- A literal value is encoded in 12 bits — 8 bits of a value and 4 bits of a rotation of that value — number of rotations is the 4-bit value multiplied by 2.
- This can be used to represent many useful values, but not any arbitrary 32-bit value.
- So, how can you represent an access a 32-bit value.
- e.g. this would be useful but it can't be done:
  `mov r0,#$0xe0004044`
  because the immediate operand can not be represented using the 12-bit scheme.
- So, what do you do?
- The solution is to use "literal pools" — groups of values stored memory where they can be referenced from the code.

Replace the instruction with another



- Instead of the [impossible] instruction
  `mov r0,#0xE0004044`
  We have an instruction
  `ldr r0,[PC,#<disp>]`
  where at the location pointed to by `PC` plus the value of the displacement value `<disp>` we have placed a `DCD` with containing the value `0xE0004044`.

- We must make sure that the DCD can not be mistaken for an instruction, so we place it, along with others, in pools outside areas of executable code.

- It turns out that calculating the displacement is a little tricky.

Pseudo instructions

- So, the ARM assembler has one further (small) category of statement, a pseudo instruction.
- E.g. the statement:
  `ldr r0,=<operand>`
  is a pseudo instruction. It looks like a specific instruction, but actually the assembler will substitute an appropriate instruction, or an instruction referring to a literal in a literal pool.

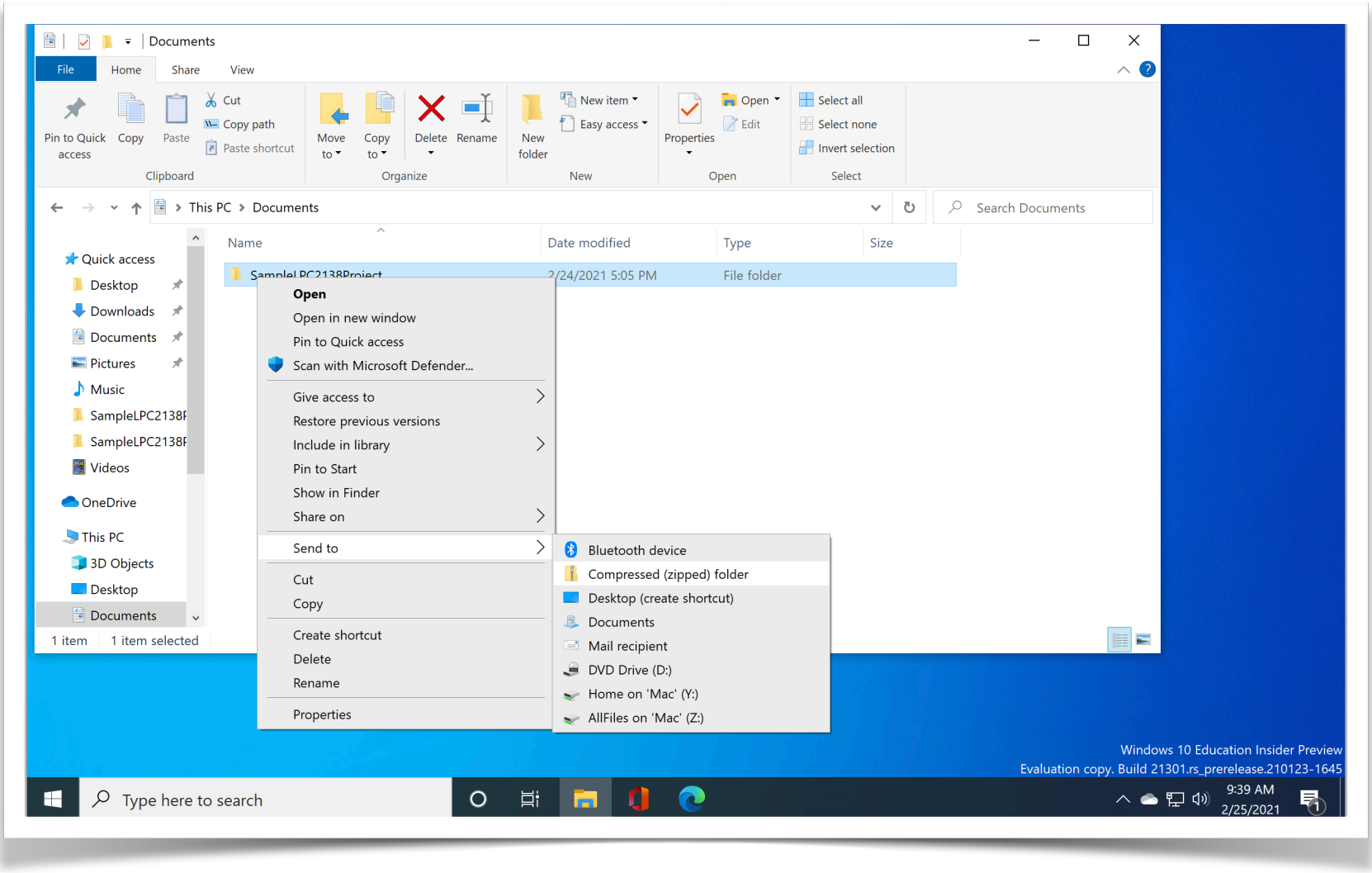
Lab 4

- Write a subroutine to return, in R0, the binary integer equivalent of the decimal integer, represented as a character string pointed to by R1.
  - The result should be returned in R0
  - R1 should point to the next character after the characters representing the number
  - Set the C bit of the CPSR if there were any problems.
  - Clear the C bit of the CPSR if everything was okay.

Assignment 1

- 64-bit recursive factorial subroutine.
- Assignment is online now.
- Worth 8% — may go to 10% if we can change CA from 20% to 30%.
- Rigid specifications — structure, results, etc.
- A few related questions to be answered in a separate file "qanda.txt". (Use Notepad or similar).
- Scored out of 20 points.
- Two-week deadline.
- Submit the complete project folder as a Zip archive…

Creating a Zip archive on Windows



Note: you can also access the Zip facility under the "Share" panel in the ribbon menu

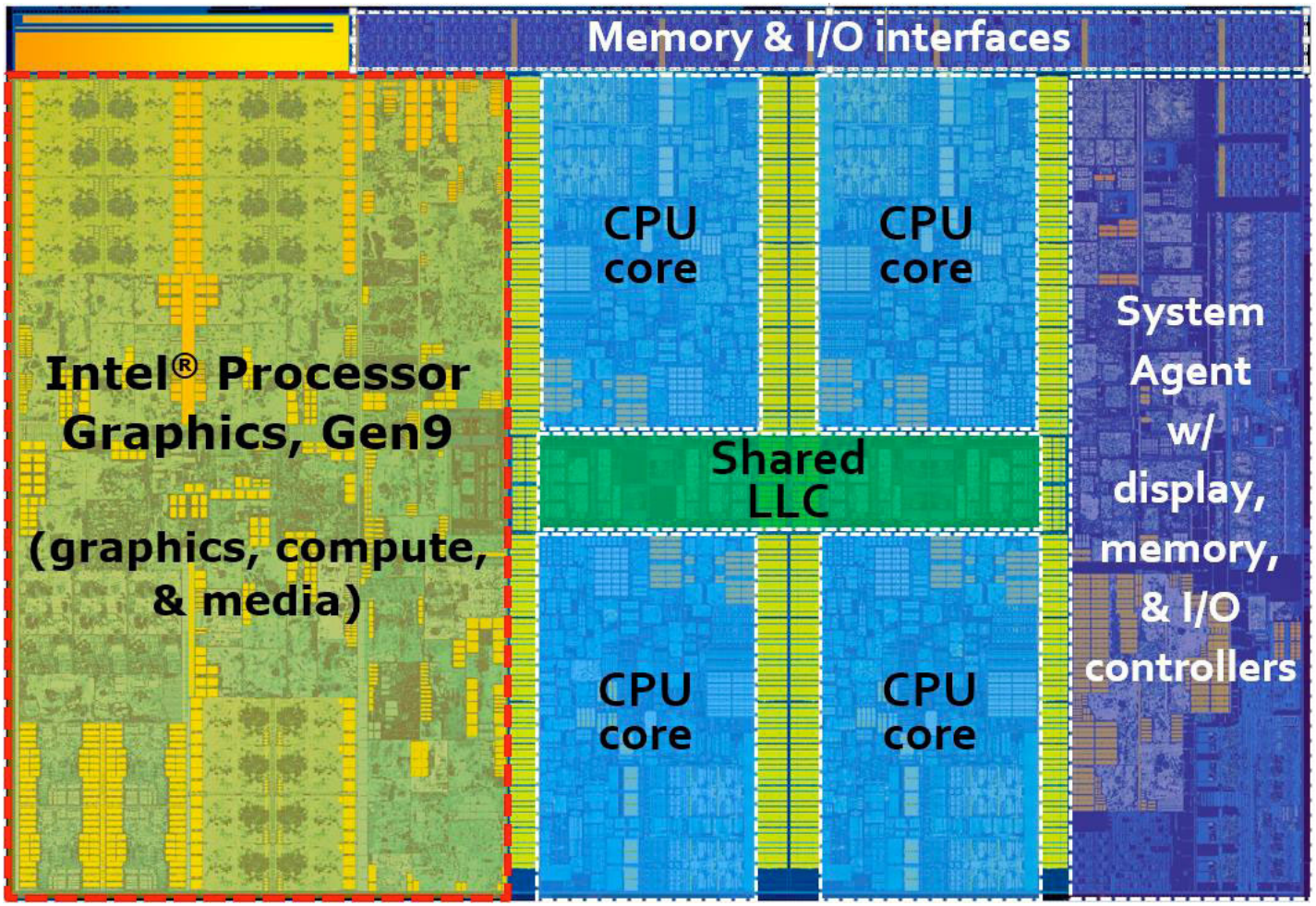
Architectural Enhancements

- The Von Neumann architecture remains the basis for thinking about how a computer with a single CPU works, often referred to as the *"programmer's model"* of the CPU, or of program execution in general.
- Most programmers and developers reason about program execution on the basis of the Von Neumann architecture:
  - Instructions are fetched from memory and executed in order of fetching (*"in-order instruction execution"*).
  - The next instruction to be executed is, by default, the instruction following the current one in memory.
    - Branch and Jump instructions alter this assumption – the next instruction is fetched from the address specified by the branch (if it is taken) or the jump.
- In the pursuit of performance, a number of key enhancements are available that, mostly, do not affect the programmer's model too much.
- You can ignore the enhancements, but it is very useful to be aware of some of their "side effects".

Architectural Enhancements

- Architectural Enhancements
  - *Pipelining* – working on different parts of many instructions at the same time in a sort of multistage assembly line. This may include limited* Out-of-Order Execution.
  - *Caching* – automatically keeping copies of frequently used instructions and data in very fast very specialised "cache memory".
  - *Hyperthreading* – keeping a CPU busy by giving it a second instruction stream to execute.
  - *Multicore* – placing more than one CPU on a single chip.
- *Pipelining, multiple execution units, caching and OoOE all go to great lengths to preserve the Von Neumann semantics.
- Hyperthreading and Multicore generalise from one Von Neumann CPU to many.

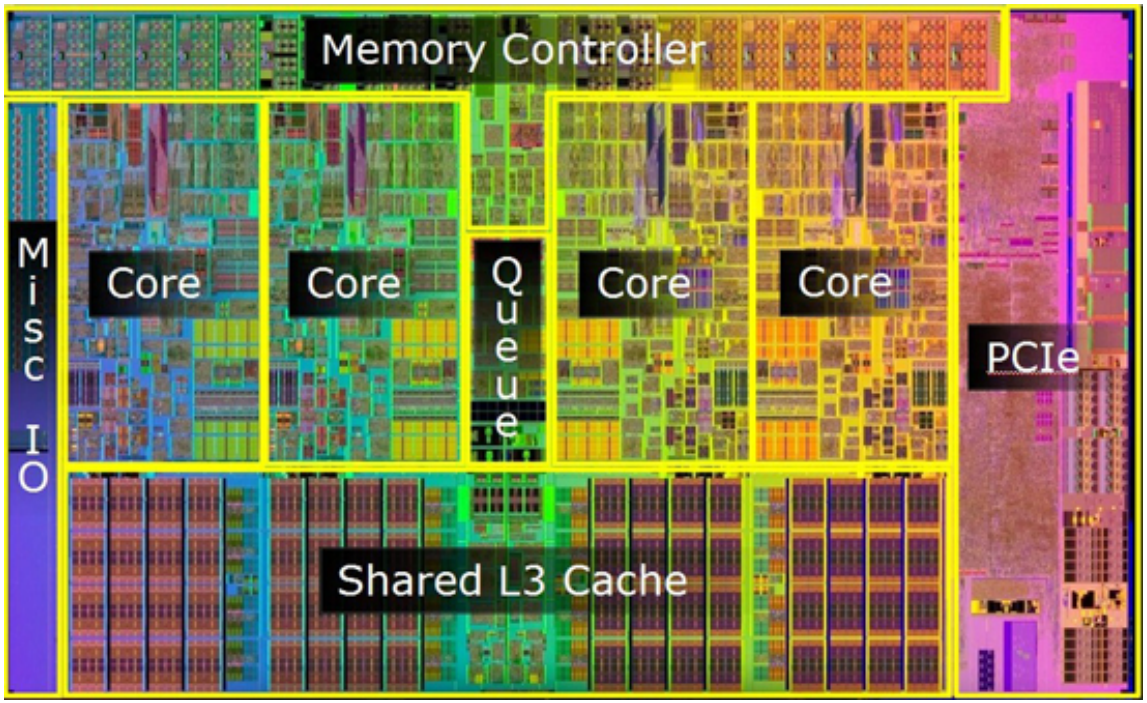
Chip Photo of an Intel Four Core with GPU



http://www.ic-cracker.com/wp-content/uploads/2013/01/break-avr-microcontroller-atmel.jpg

Intel “Lynfield” Architecture 2009

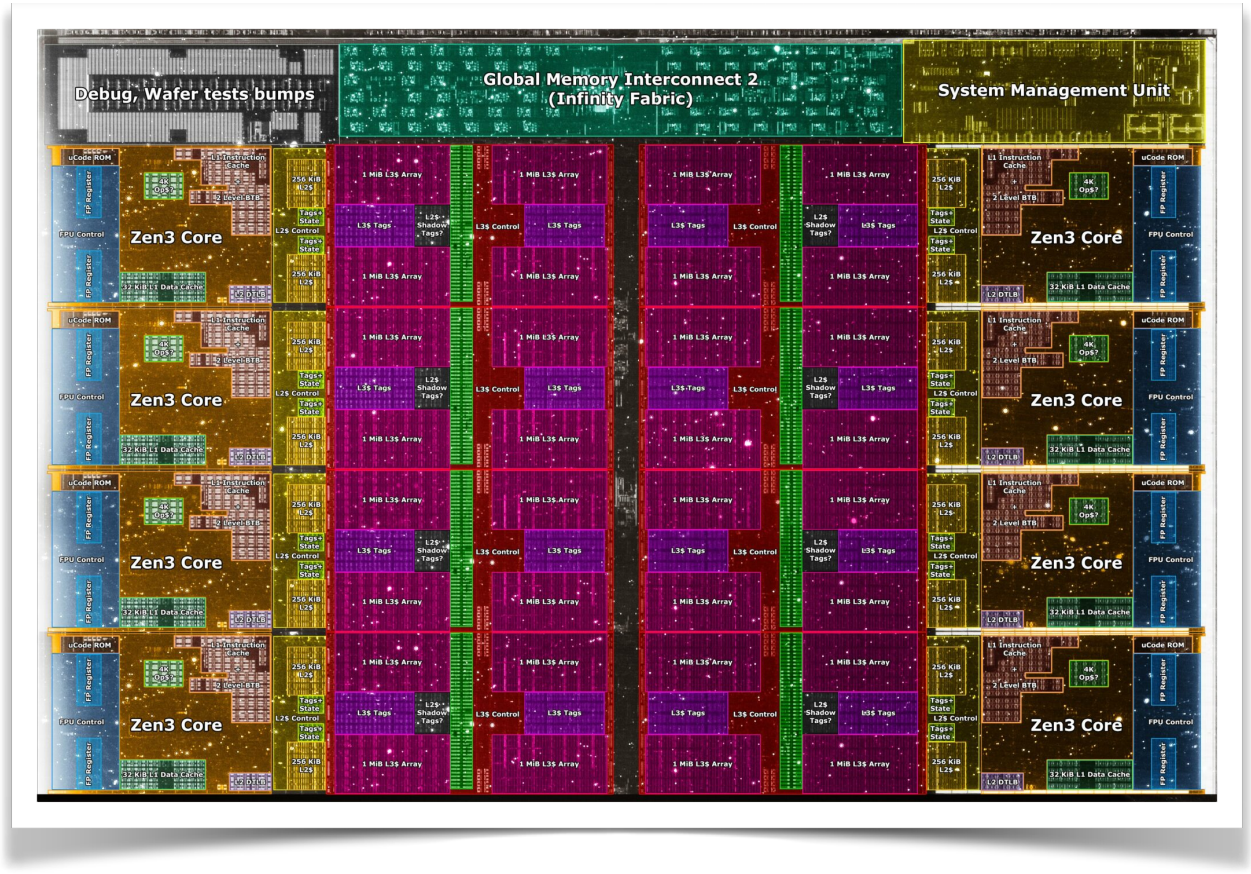
- All three lines of processors use Intel's "Nehalem" architecture, so they have 32KB of L1 cache and 256KB of L2 cache per core, and a large 8MB L3 cache that's shared between the four physical cores.



https://www.pcstats.com/articles/2581/4.html

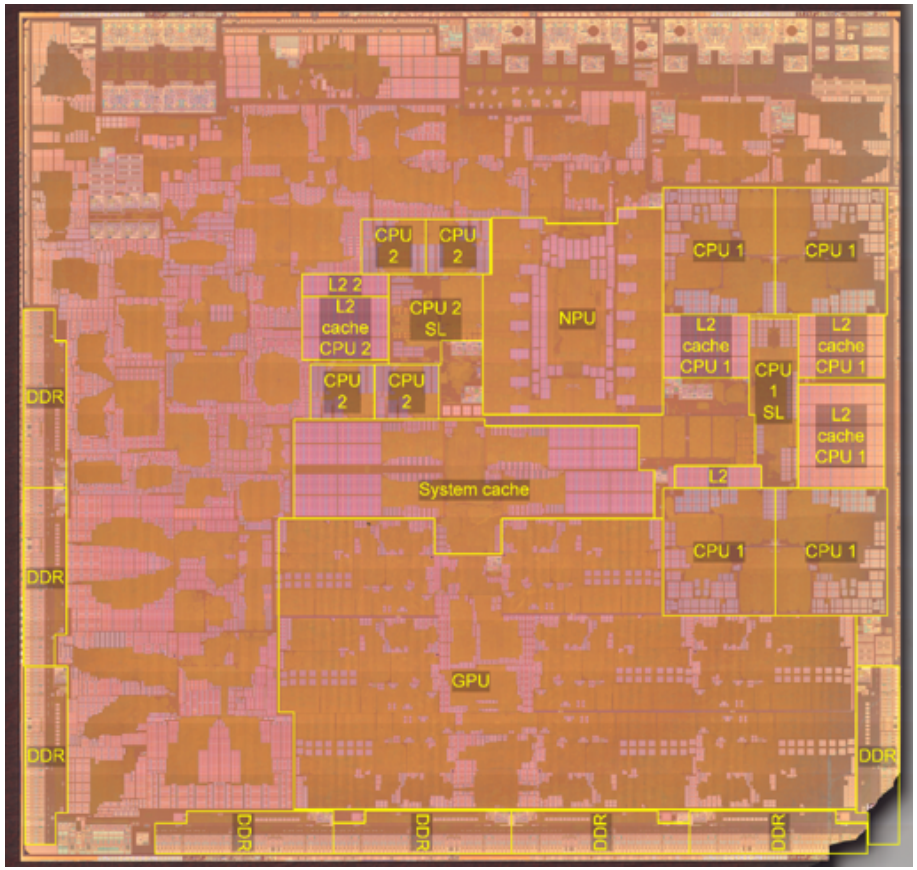
AMD Ryzen 5 5600X 2020

- It looks as if approx. half the chip is dedicated to Level 3 Cache (“L3$”) — in purple in the photo.



https://wccftech.com/amd-ryzen-5000-zen-3-vermeer-undressed-high-res-die-shots-close-ups-pictured-detailed/
Photo @Locuza_

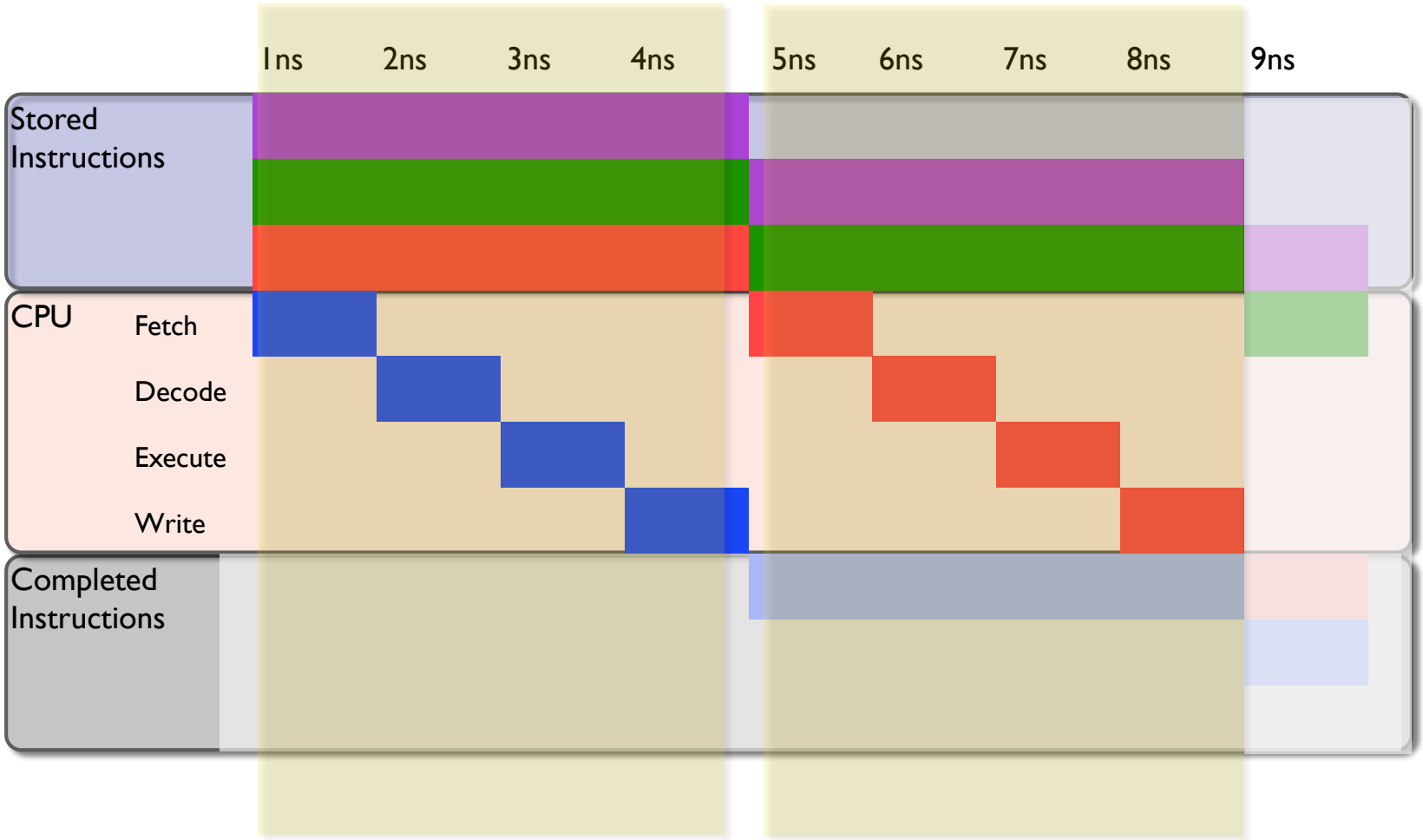
Apple Silicon M1 2020



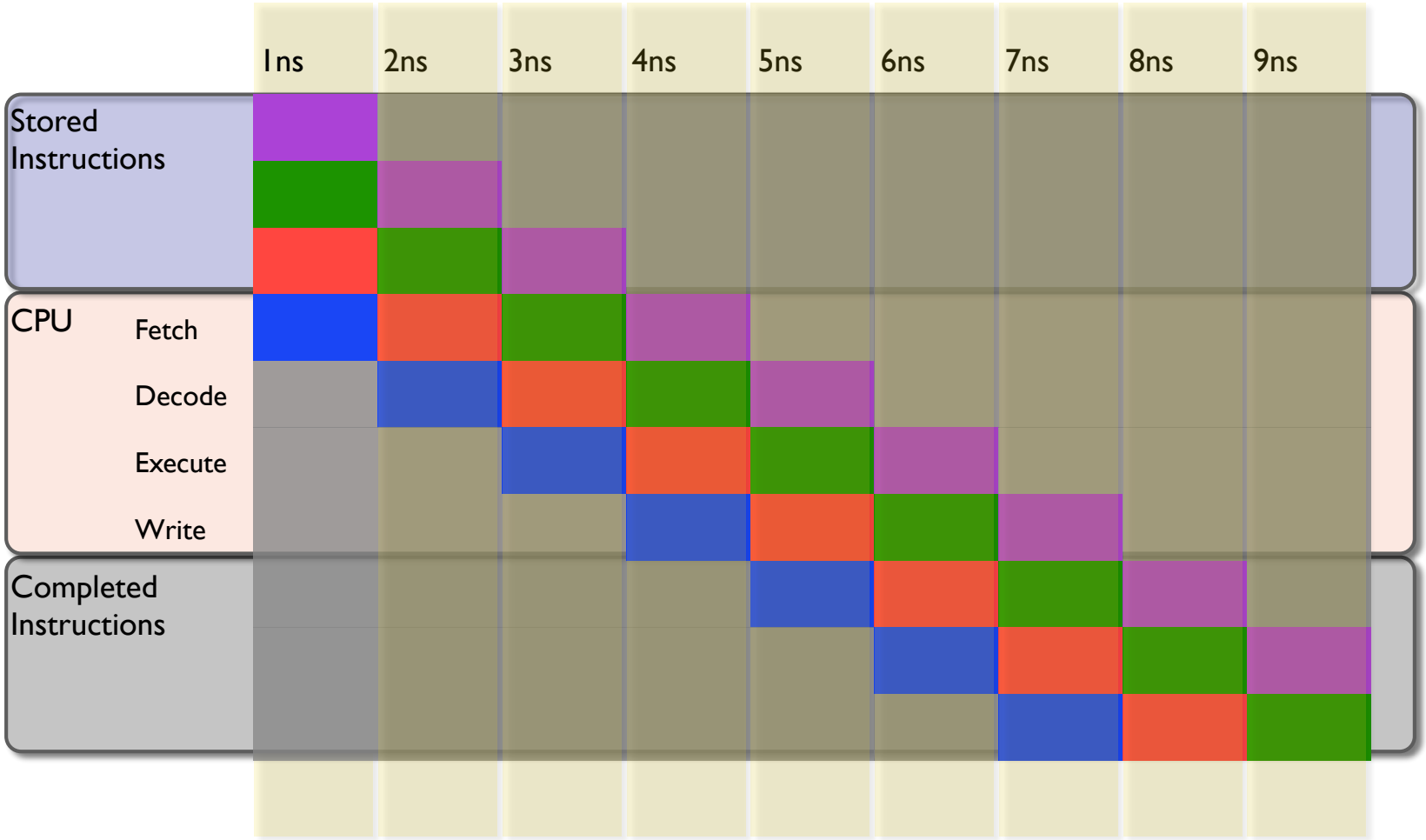
Apple M1 SoC. Image by TechInsights. CPU 1 = FireStorm, CPU 2 = IceStorm.

https://www.extremetech.com/computing/318715-comparison-of-apple-m1-a14-shows-differences-in-soc-design

Single Cycle Processor



Four Stage Pipeline



Figures of Merit

- Throughput – AKA the *Completion Rate*
  - The rate at which instructions are completed.
  - In this example, the completion rate is one per nanosecond.
    - 1,000 Million Instructions Per Second (MIPS)
- Latency – the time it takes for an instruction to execute
  - In this example, each stage takes one nanosecond and there are four stages
    - Thus the latency is 4 nanoseconds.
- Specialised instructions such as Floating Point instructions (not present in this CPU) can take very much longer.

Average vs. Peak Completion Rate

- Say we look at the first 1000 ns of program execution:
  - First instruction isn't completed until the end of the third ns.
  - After 1000 ns, 997 instructions completed, others still "in flight"
- Average completion rate for the first μs = 997 instructions per μs.
- To maximise the completion rate, maximise the pipeline utilisation.

Pipeline

- By breaking the execution of an instruction into relatively distinct pieces (e.g. fetch, decode, execute), and
- By dedicating self-contained units of hardware to performing each piece, and
- By passing each partially completed instruction to the next stage on each clock cycle, keeping every stage's hardware always busy:
- We can make the overall throughput of instructions completed higher.
- The Pentium IV from the mid 2000s has between 20 and 31 stages (different implementations) in the pipeline!

Pipeline Stages, Clock

- Each stage must complete its work in one clock period.
- Thus, by reducing the amount of work each stage must do, (and by increasing the number of stages – making the pipeline "deeper"), the clock period can be shortened, raising the clock frequency, thus raising the overall throughput.

Problems with Pipelines

- Three big problems, one technological, the others "logical" or architectural
  - Power Dissipation
  - Pipeline Stage Size
  - Pipeline Stalls

Power Dissipation

- Power dissipation of semiconductor devices rises with clock rates.
  - More transitions per second -- move current, higher power.
- As semiconductor feature sizes drop, ancillary power losses, especially through current/charge leakage rise dramatically.

Pipeline Stage "Size"

- To increase the number of stages, break instructions into smaller and smaller pieces.
- Each piece occupies one stage in a pipeline, and must complete in one clock period.
- The overall pipeline clock period is governed by the stage taking the longest time.
- Thus, stages that take less time than the longest will not be running at full utilisation.
- So the idea is to even out the activities in stages so they all take the same amount of time.

Pipeline Stalls

- A really big problem for pipelined architectures is when the pipeline "stalls".
- When a pipeline stage cannot complete its work in a clock cycle, it "stalls" the pipeline:
  - All instructions closer to the end will continue, but the all instructions in stages behind the stalled stage are stopped.
  - As the instructions preceding the stalled instruction leave the pipeline, "bubbles" of idle stages fill the processor.
  - When the stalled stage resumes, the bubbles will be flushed gradually from the pipeline...

Kinds of Stalls

- Memory Access
- Control Hazards
- Data Hazards
- Structural Hazards

Stalls (1) – Memory Access

- Memory access is usually very much longer than the clock period of the processor (more later on).
- Modern computers have a memory hierarchy, which must be managed, from small very fast memory (e.g. processor registers) down through caches to main memory.
- Access to a memory location not in cache can cause a very long pipeline stall.

Stalls (2) – Branching

- If a branch or jump is taken, then the instructions that were fetched sequentially must be discarded.
  - The pipeline must be flushed and refilled.
- The deeper the pipeline,
  - the more chance of a branch instruction being in the pipeline
  - the more that must be discarded
- This is a Control Hazard

Stalls (3) – Data Dependencies

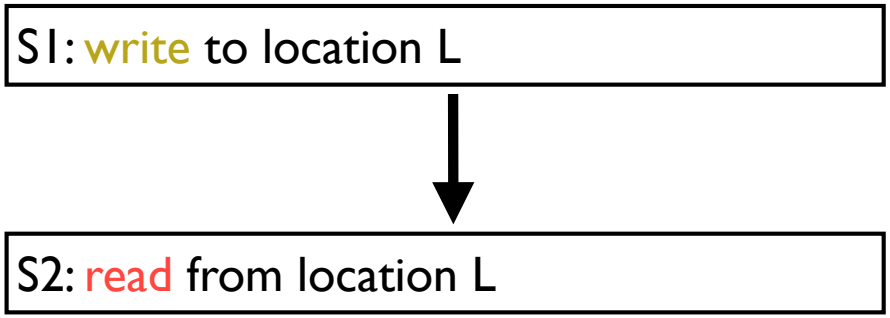
- Imagine two instructions:

  add A,B,C ; C ← A+B
  add C,D,D ; D ← C+D

- In a pipelined processor, the second *add* must wait until the first *add* instruction has completed its write operation, introducing a bubble.

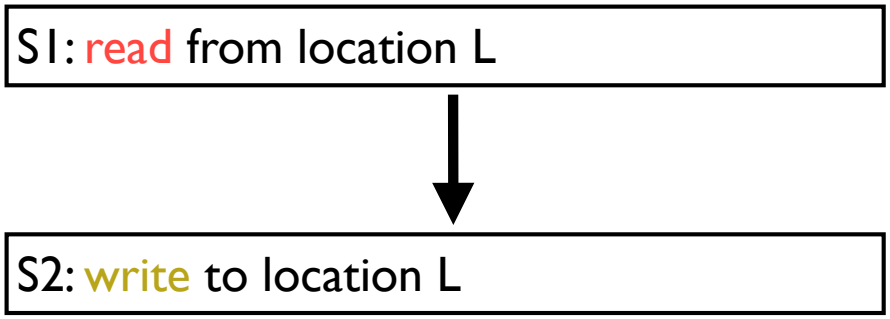
1 – Flow Dependence

- Flow Dependence: S2 is flow dependent on S1 because S2 reads a location S1 writes to.
  - It must be written to (S1) before it's read from (S2)



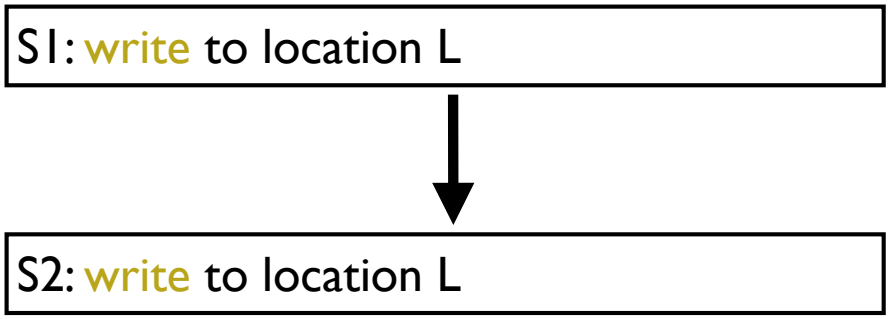
2 – Antidependence

- Antidependence: S2 is antidependent on S1 because S2 writes to a location S1 reads from.
  - It must be read from (S1) before it can be written to (S2)

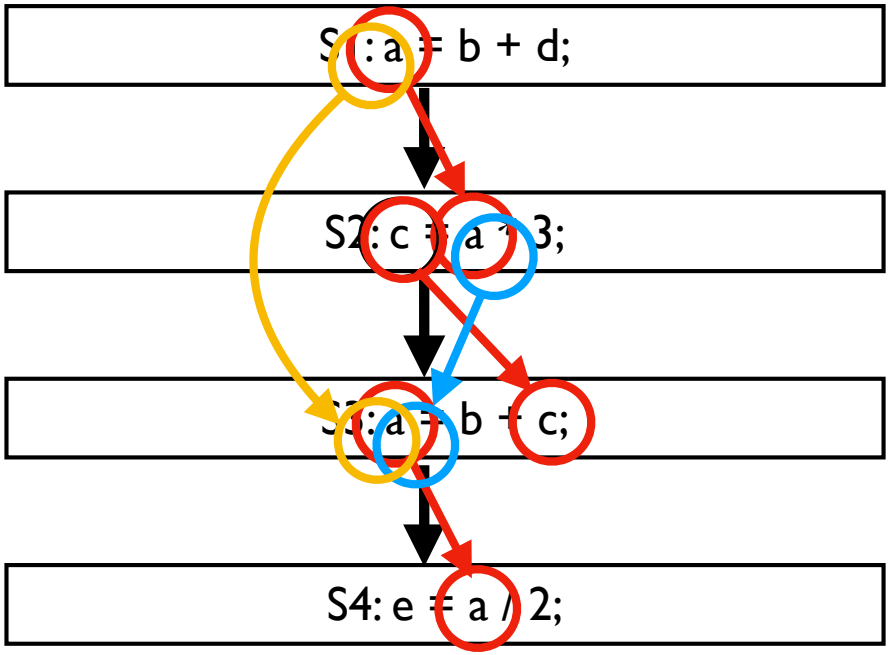


3 – Output Dependence

- Output dependence: S2 is output dependent on S1 because S2 writes to a location S1 writes to.
  - The value of L is affected by the order of S1 and S2.



Example



Flow Dependence Relation

Anti Dependence Relation

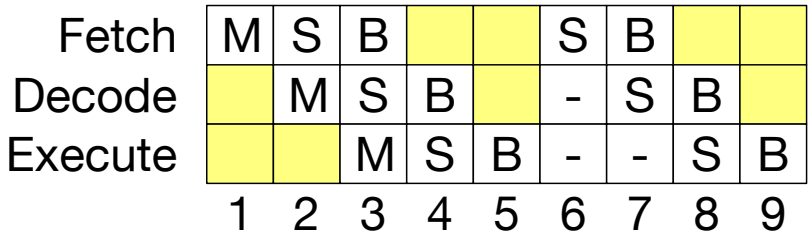
Output Dependence

Stalls (4) – Structural Hazard

- A structural hazard occurs where the processor doesn't have the hardware resources to allow two things to be performed independently.
- For example, suppose a processor had just one 32-bit adder which was used for integer arithmetic but also for address calculation.
  - Then, a stage calling for an address calculation couldn't proceed while another stage calling for an integer calculation was going on at the same time.

Hypothetical Pipeline Calculation

- Consider the sequence of instructions below in an imaginary pipelined CPU with three stages: fetch, decode and execute, each of which takes 1 nanosecond.
- Assume no cache is needed, as memory accesses are extremely fast and do not delay any stage of the pipeline, and assume that the CPU has no branch prediction logic.
- How long will it take to execute the sequence of instructions, from the time the first instruction is fetched until the last instruction is complete?




```
M =       mov r0,#2
S =  l1   subs r0,#1
B =       bne l1
```

Lab 5

- Write a program in which you represent three matrices A, B and C of signed 4-byte integers.
  - A is a 1 x 3 matrix
  - B is a 3 x 2 matrix
  - C is a 1 x 2 matrix
- Calculate C as the product of A and B.
- Figure out how to represent the matrices in memory.
- Represent them all the same way — no funny stuff!
- Give them suitable sample values.
- Perform the multiplication.
- Check the results.
- Don't worry about overflow, etc. in this lab only!

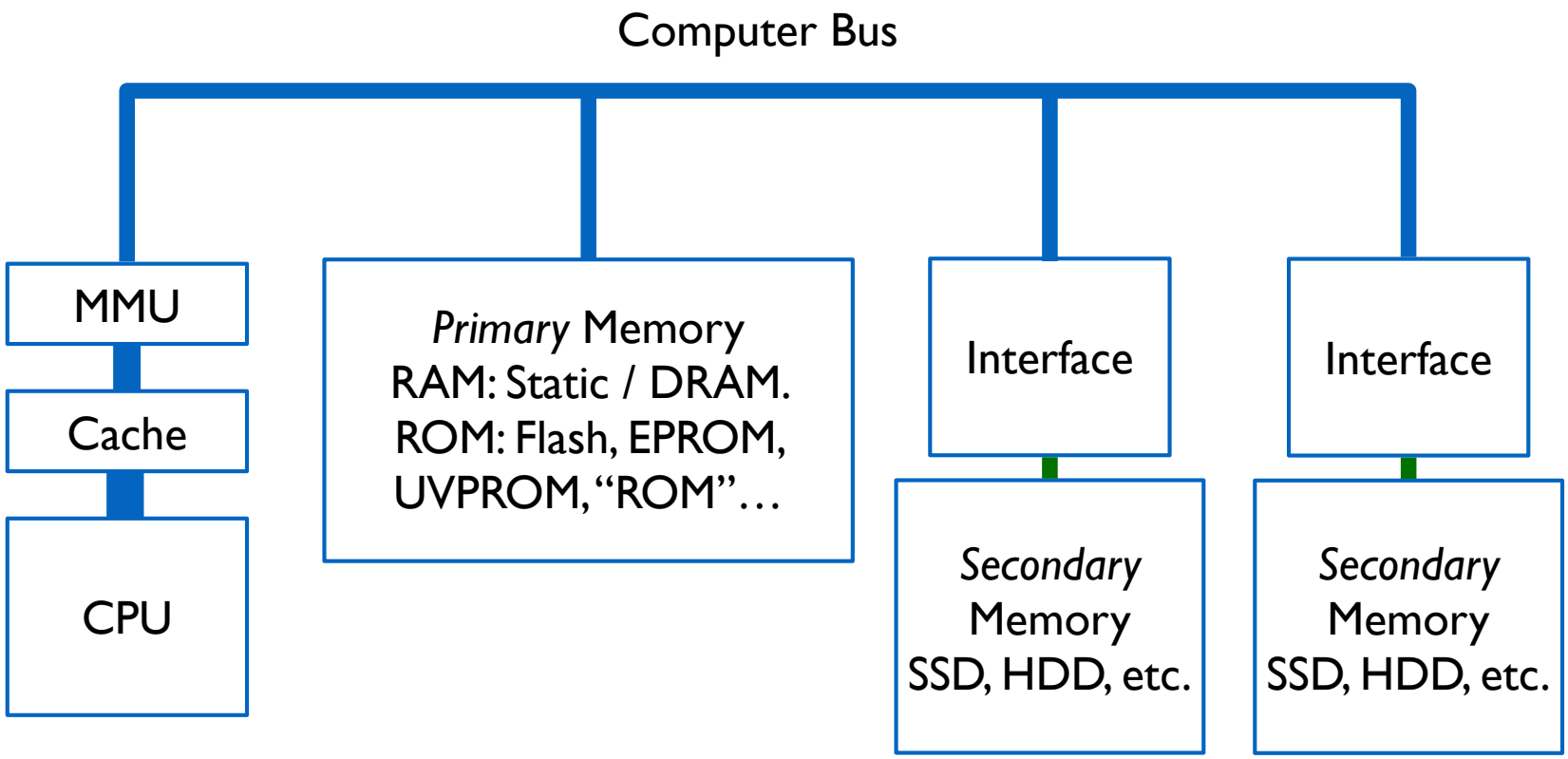
$$\begin{bmatrix} 1 & 4 & 6 \end{bmatrix} A \quad \times \quad \begin{bmatrix} 2 & 3 \\ 5 & 8 \\ 7 & 9 \end{bmatrix} B \quad = \quad \begin{bmatrix} 64 & 89 \end{bmatrix} C$$

Sample multiplication from: https://www.mathwarehouse.com/algebra/matrix/multiply-matrix.php

Wait, there's more…

- Huge increase in the number of transistors that could be integrated onto a chip.
- What to do?
- One obvious possibility is to add more ALUs

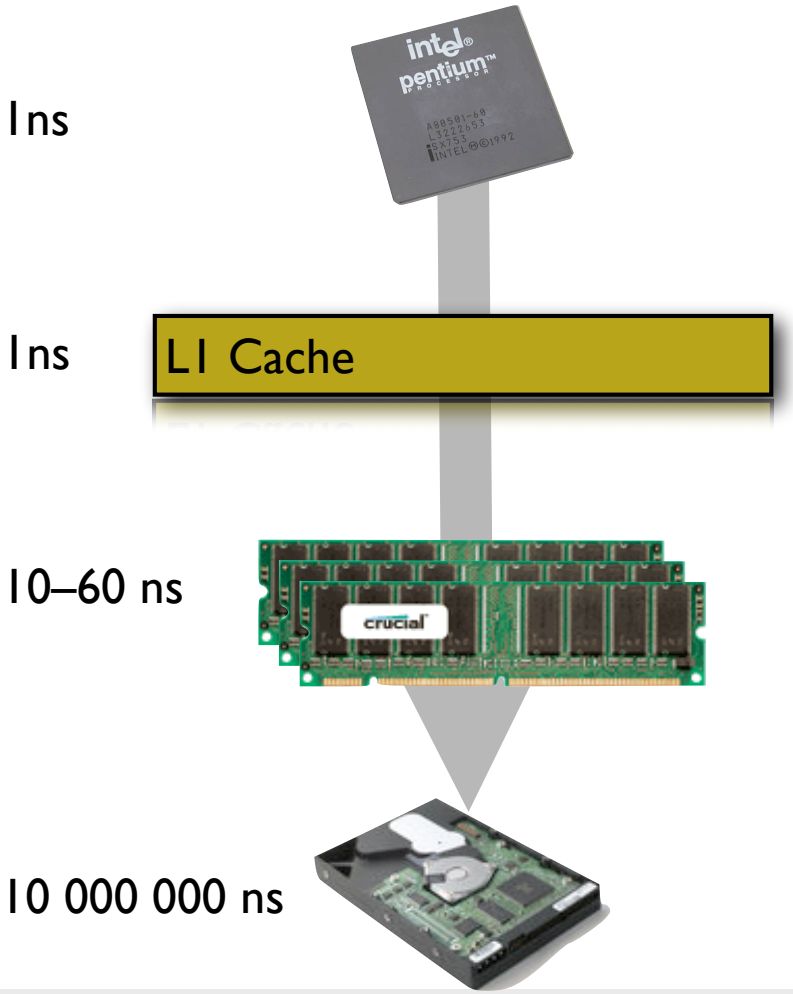
Slightly More Realistic Von Neumann Layout



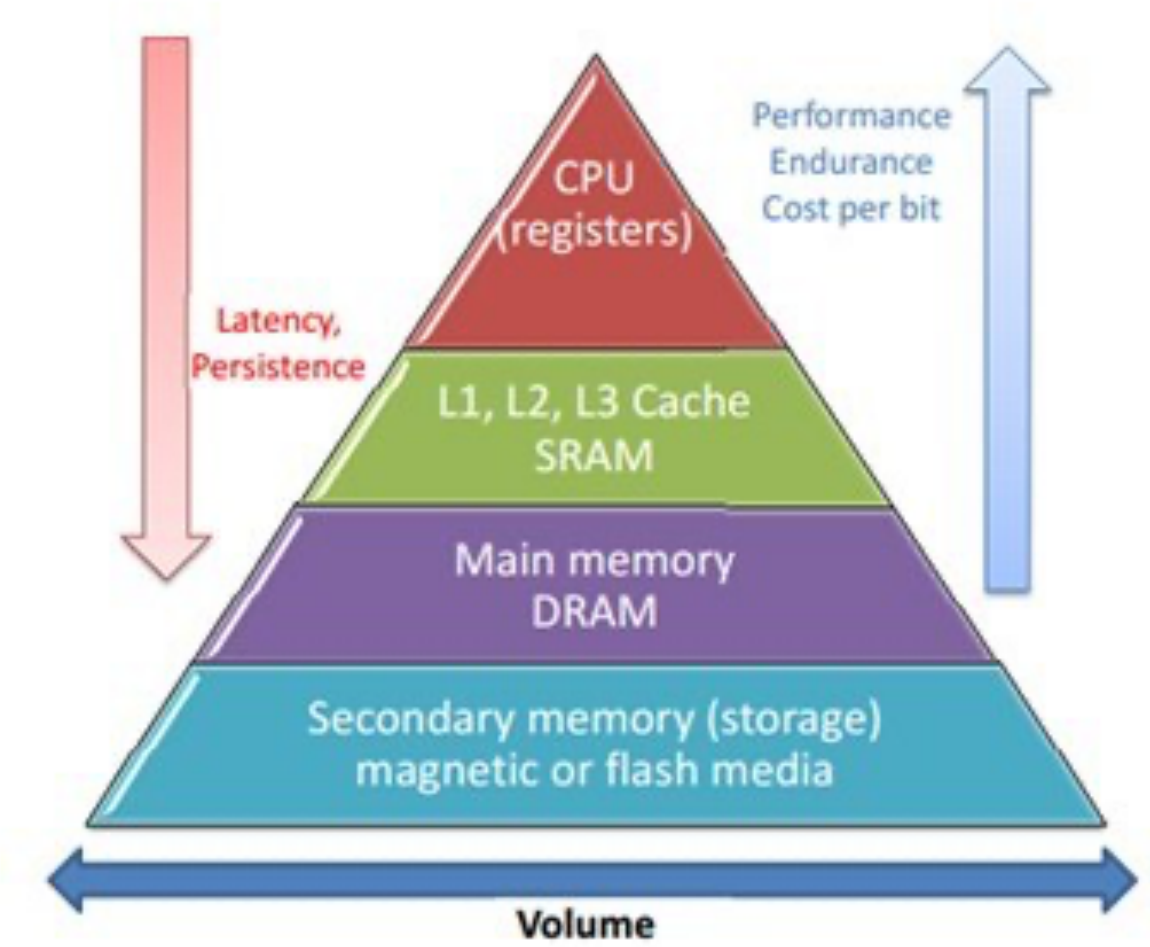
Typical for a Desktop / Laptop / Smartphone

MMU = Memory Management Unit, SSD = Solid State Drive, HDD = Hard Disk Drive, etc. could include USB Memory Sticks, Tape Drives, more.

Memory Hierarchy



Memory Hierarchy Pyramid



source: https://www.researchgate.net/publication/319529366_Emerging_NVM_A_Survey_on_Architectural_Integration_and_Research_Challenges

Registers

- Registers are implemented using very fast memory.
- Registers are referenced by name, and thus controllable and managed by the program, thus by the programmer or the compiler.
- Volatile.

Primary Memory — "Memory"

- Primary Memory is memory that may be referenced by addresses that form part of a machine code instruction — i.e. addressable by the CPU.
  - Access to any random address always takes the same time as to any other location.
  - Can be ROM (Read-Only Memory) or RAM (can be read from and written to).
    - ROM is Readable but not Writable or alterable. It is non-volatile. Attempting to write to ROM will typically silently fail but could, in principle, be made to cause a run-time exception.
    - RAM is both Readable and Writable. RAM is usually volatile — remove power and it loses its contents.
- Primary Memory is often called "Main Memory" or sometimes just "Memory".

Secondary Memory — "Storage"

- Secondary Memory is usually some kind of storage that is accessible via a peripheral interface, e.g. Hard Disk Drive, SSD, USB Stick, Floppy Disk, Network Storage device.
  - Secondary memory, by definition, may not be referenced by addresses forming part of a machine code instruction.
  - If you want to access something in secondary memory, it must be copied in to primary memory first.
  - Secondary Memory is usually non-volatile, and thus often called "Storage".

Main Memory — RAM

- RAM stands for *Random Access Memory*, but it is always understood to be *Read-Write Random Access Memory*, i.e. it can be read from or written to.
- Main Memory typically includes using some form of DRAM and/or SRAM, and is addressable either directly by a program or indirectly via memory management.
  - DRAM = Dynamic RAM — each bit is stored in a gated capacitor, which leaks over time.
    - Before it's too late, its value is periodically read and the capacitor is recharged ("refreshed").
    - DRAM is relatively compact, relatively cheap and volatile.
  - SRAM = Static RAM — each bit is stored in a flip-flop.
    - Static RAM is very fast, but takes up more space than DRAM, can be very low power when not accessed.
    - Expensive, volatile.
- In either case, the actual location to be accessed is known and can be issued to the memory system.

Main Memory — ROM

- Main Memory almost invariably includes some kind of ROM – *Read Only* Random-Access *Memory*. It is typically used to hold programs and data needed to start up ("boot") a computer system.
  In embedded systems (e.g. the LPC2138) it may hold all the software.

- Many different ROM technologies. Here are a few:

  - Flash ROM. The contents are normally read-only and non-volatile. However, in a special programming mode the contents can be rewritten ("flashed"). Convenient. Used on the LPC2138.

  - UVPROM. The contents are normally read-only and non-volatile. However, the contents can be erased by exposure to very short-wave and dangerous Ultra Violet radiation. Convenient but probably outmoded.

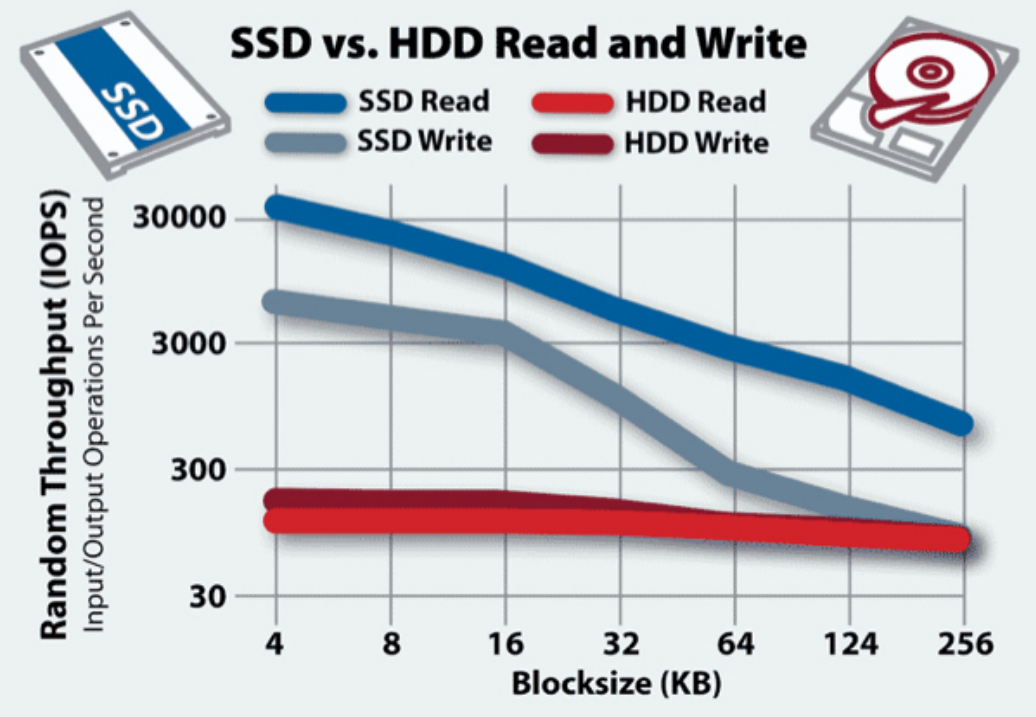
  - Mask ROM. The contents are imprinted during manufacture and can never be altered. Cheap.

- As primary memory, the actual location to be accessed is known and can be issued to the memory system.

Hard Disk Drives and Solid State Drives

- Hard Disk Drives (HDDs) and Solid State Drives (SSDs) can be viewed as very large but slow holders of numbered bytes of data.
  - HDDS are very slow and are not random access devices.
  - SSDs can be very fast but are not random access devices.
- Almost invariably, file systems are built on top of the "bare" drives to organise them. (File systems are usually provided as part of an Operating System.)
- But, as with registers and main memory, the location to which access it to be made is known to the program or programmer.
- HDDs and SSDs are non-volatile.

SSD vs Hard Disk



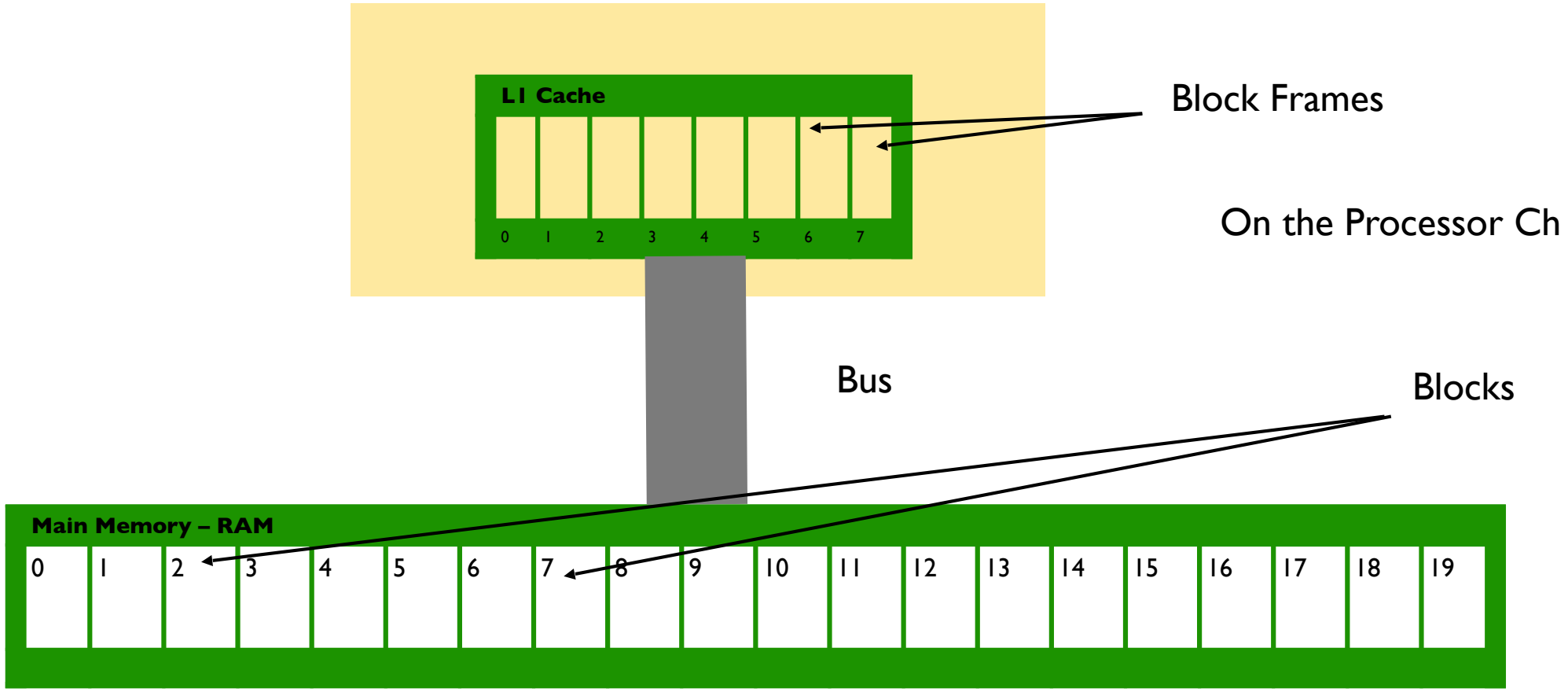
Note: Small (4kB — 64kB) Blocksize Performance is extremely important for Virtual Memory implementation

https://www.enterprisestorageforum.com/storage-hardware/ssd-vs-hdd-speed.html (2019)

Caches

- Caches are un-named and un-numbered memory stores, managed by hardware in response to run time conditions.
- Caches usually (but not always!) contain duplicates of the contents of memory locations.
- When the processor wishes to access those memory locations…
  - If they are duplicated in cache (a cache "hit"), the cache supplies the item, speeding up access.
  - If not, (a cache "miss"), when the item is finally referenced in memory, it may be copied to a cache for future reference.

Cache Organisation



From "Inside The Machine" by Jon Stokes

Cache Organisation

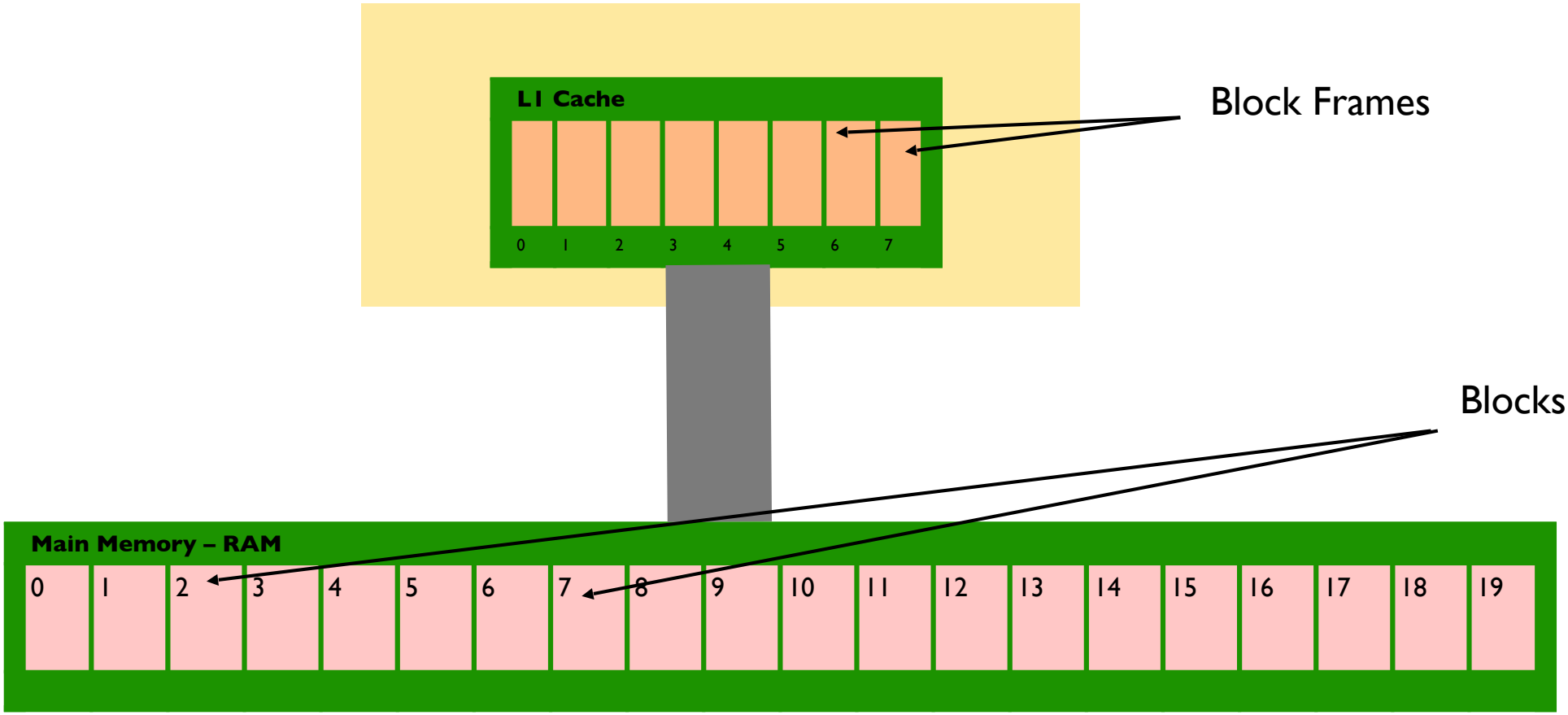
tag	flags	cached memory contents
<address>	<management flags>	<e.g. 64 bytes from the address>

Organisation of a Hypothetical Cache Line

- A cache is organised as a (small) number of block frames, each capable of holding a chunk of contiguous main memory locations in a structure called a "cache line" or a "cache block".
- Associated with each block frame cache is a cache line with Tag RAM containing information about where it's from in main memory — effectively the address of the cached memory contents. There will also be flags used to manage the cache line.
- When an access is made, the Tag RAM contents indicates whether a block contains a cached copy of the location.
- Tag RAM must be very very very fast, thus is very expensive in silicon, complexity and energy consumption.

Fully Associative

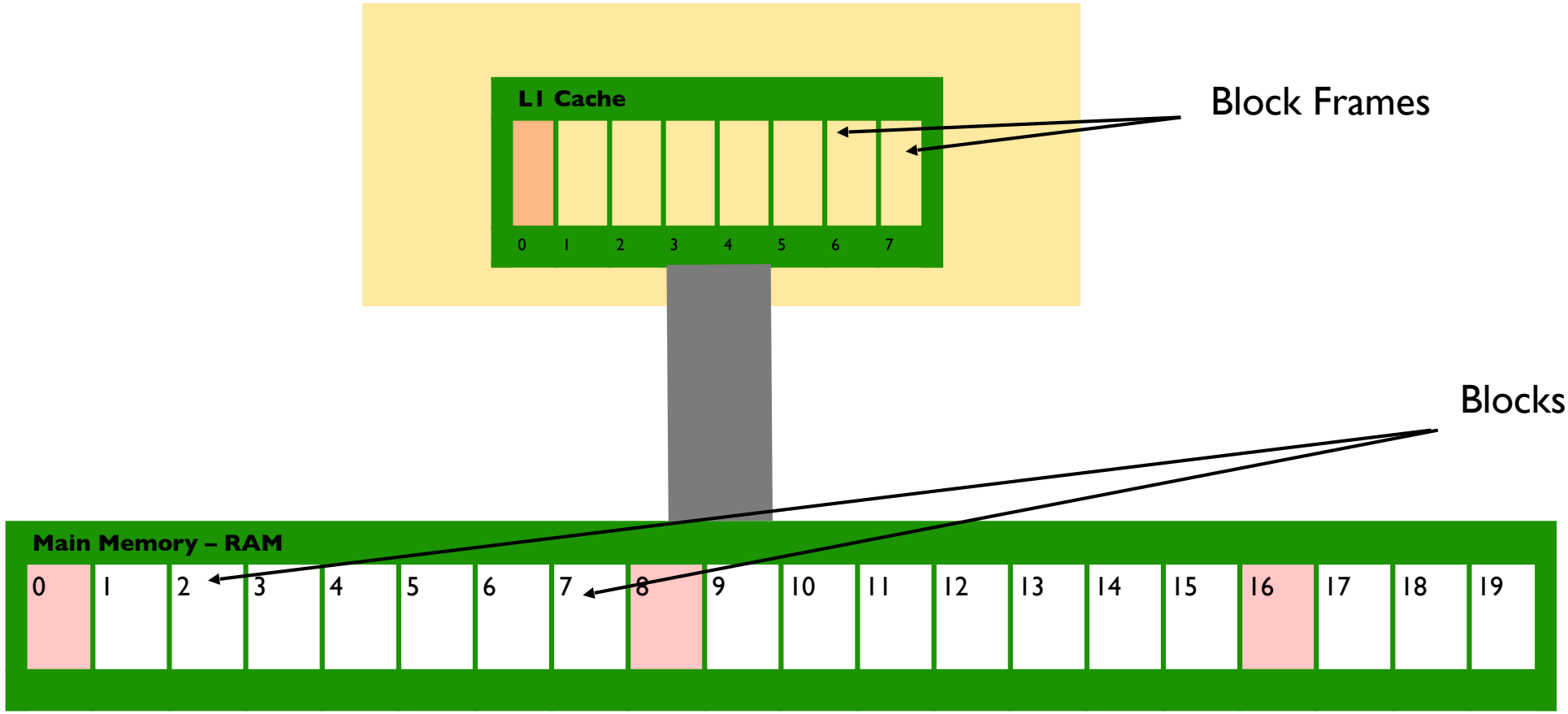
Any Block Frame to any Block



Fully Associative Mapping

- Due to problems building very fast Tag RAM, there are different kinds of Cache Management.
- Fully Associative Mapping:
  - Any RAM block can be mapped into – ("associated with") any block frame in cache.
  - Simple to understand.
  - Problem is that to find a reference, every single Tag RAM must be searched.
    - Can be slow and/or complex for larger caches,
    - Scales badly.

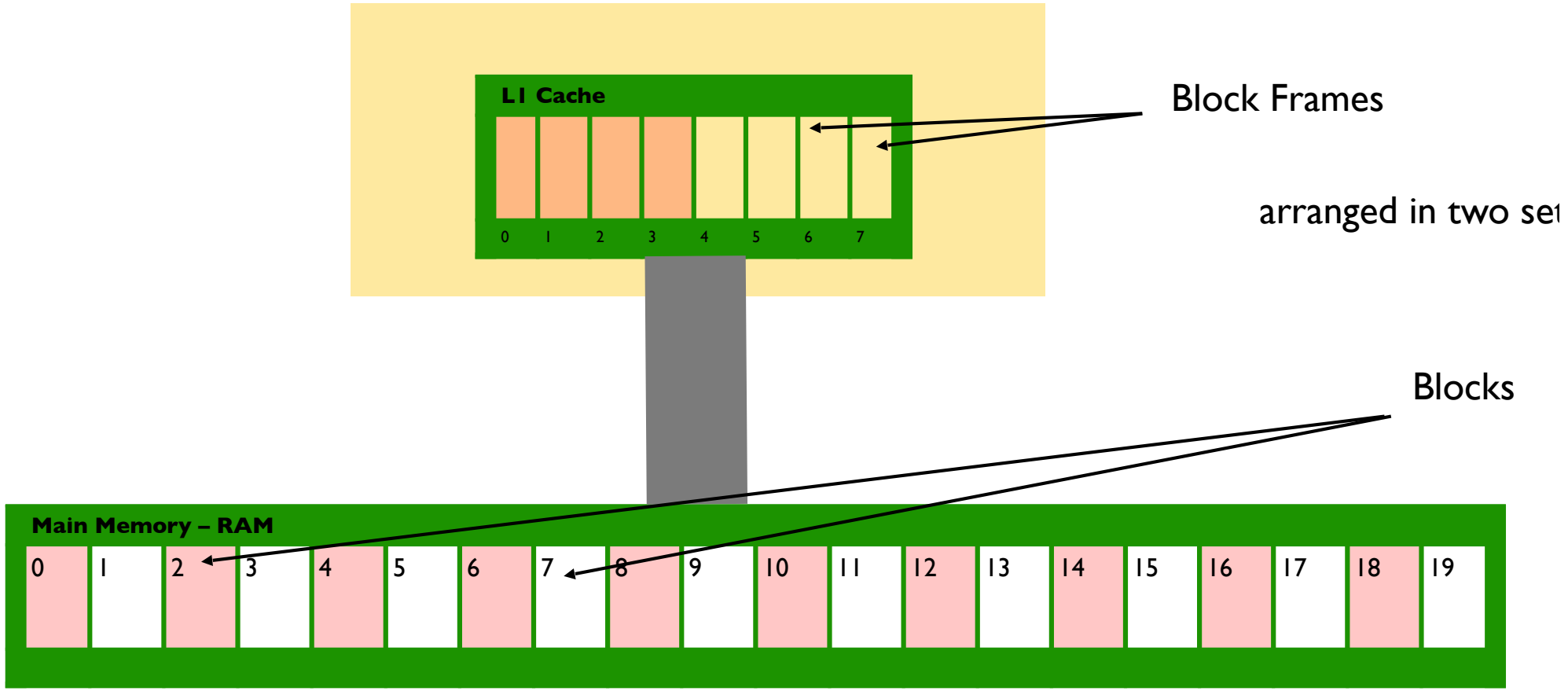
Direct Mapping



Direct Mapping

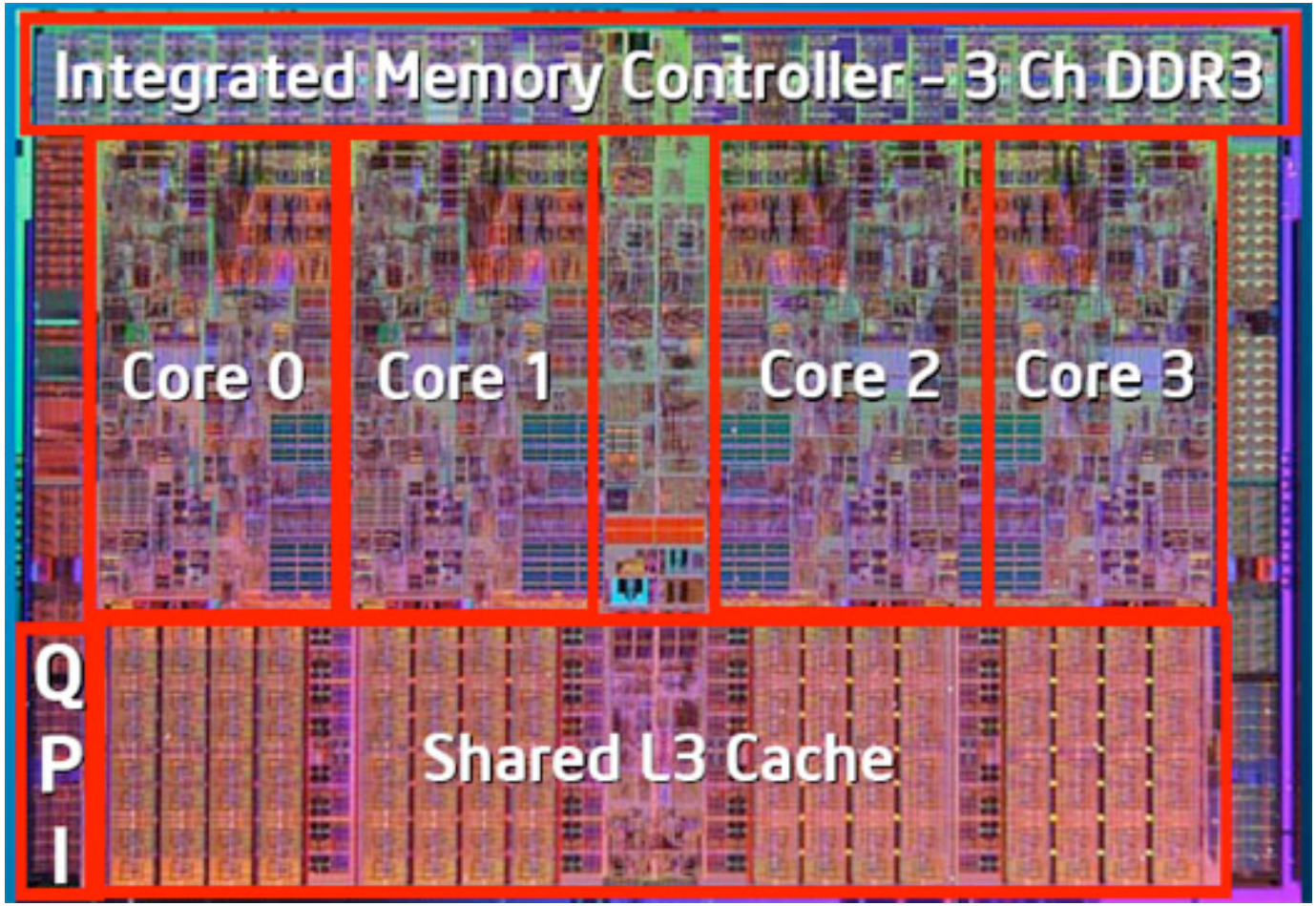
- Each block frame (in cache) can only cache a subset of memory blocks.
  - Faster,
  - Restrictive – some pathological cases.

Four-way Set Associative



Blocks and block frames are divided into sets, and f

Cache is a big deal…



Intel Nehalem (around 2008)

Cache Content Management

- Caches are invariably smaller than main memories, therefore there will be pressure on space in them.
- As programs execute over time, caches will fill up and new references will have to be accommodated, requiring existing cache entries to be evicted. This requires some kind of *Cache Replacement/Eviction Policy*.
- Ideally a cache replacement policy has the minimum effect on overall performance.
- Temporal locality of reference is exploited in a Least-Recently-Used (LRU) policy:
  - Due of temporal locality, an LRU policy, or a variation, is intended to predict the cache entry least likely to be used again soon.

Spatial Locality – Block Size

- The smaller the block size, the finer-grained the coverage of the working sets of the processes, and the less "wasted" memory references – i.e. fetches of locations that are part of the block but that will never be referenced.
- But smaller block sizes imply larger numbers of cache blocks, increasing the complexity of the caches.

Write Policies

- Two ways to handle a write in the presence of cache:
  - Write-Through: all copies of the data are updated straight way
    - Simple to conceptualize, good for multiple-bus-master systems.
    - Slower, higher bus bandwidth requirements
  - Write-Back: the write is only sent to the cache. When the cache entry is evicted, or when the system isn't busy, any pending changes are written back to main memory.
    - Lower memory bandwidth, but problems with multiprocessor, multi-bus-master systems.

Cache Misses

- A cache miss is when a reference is made that the cache can't satisfy.
- Three kinds of reasons for a cache miss:
  - *Compulsory Miss* – the reference was never stored in cache.
  - *Collision Miss* – a reference was stored in cache but was evicted because another a reference to another block in the block frame's set was made.
  - *Capacity Miss* – a reference was stored in cache was was evicted because there wasn't enough space in the cache.

Hypothetical Cache Calculation

- Consider the sequence of instructions below are executed on an an imaginary non-pipelined CPU with a cache. Assume instructions and data are initially uncached.
  - An uncached instruction normally takes 10 nanoseconds to run. A cached instruction normally takes 1 nanosecond.
  - However, if the instruction is a load (LDR) instruction, the instruction will take an extra 10 nanoseconds if the location to be loaded from is uncached, and no extra time if the location is cached.
  - Similarly, if the instruction is a memory store (STR) instruction, the instruction will always take an extra 10 nanoseconds (i.e. writethrough caching is enabled) to write the data to the location.

		I	Dr	Dw	I	Dr	Dw
	`movs r0,#2`	10	-	-			
`l1`	`ldr r2,[r1]`	10	10	-	1	0	-
	`add r2,#1`	10	-	-	1	-	-
	`str r2,[r1]`	10	-	10	1	-	10
	`subs r0,#1`	10	-	-	1	-	-
	`bne l1`	10	-	-	1	-	-
	Instructions	First Run			Loop Once		

I = Instruction Fetch
Dr = Data Read
0 means the data is in cache
Dw = Data Write
Numbers are nanoseconds

Review Questions

- Why should software developers care about cache?
- What is write-through / write-back?
- Describe and contrast Full Associative / Direct Mapped and N-Way Set-Associative cache organisations.
- Caches are an important and scarce resource. Describe how they are managed for maximum benefit.
- Write a short for loop in ARM assembly (not more than five instructions) and illustrate how long it might take in the presence of an initially cold cache.

Simple Style Pointers

Single Entry and Exit points for Subroutines.

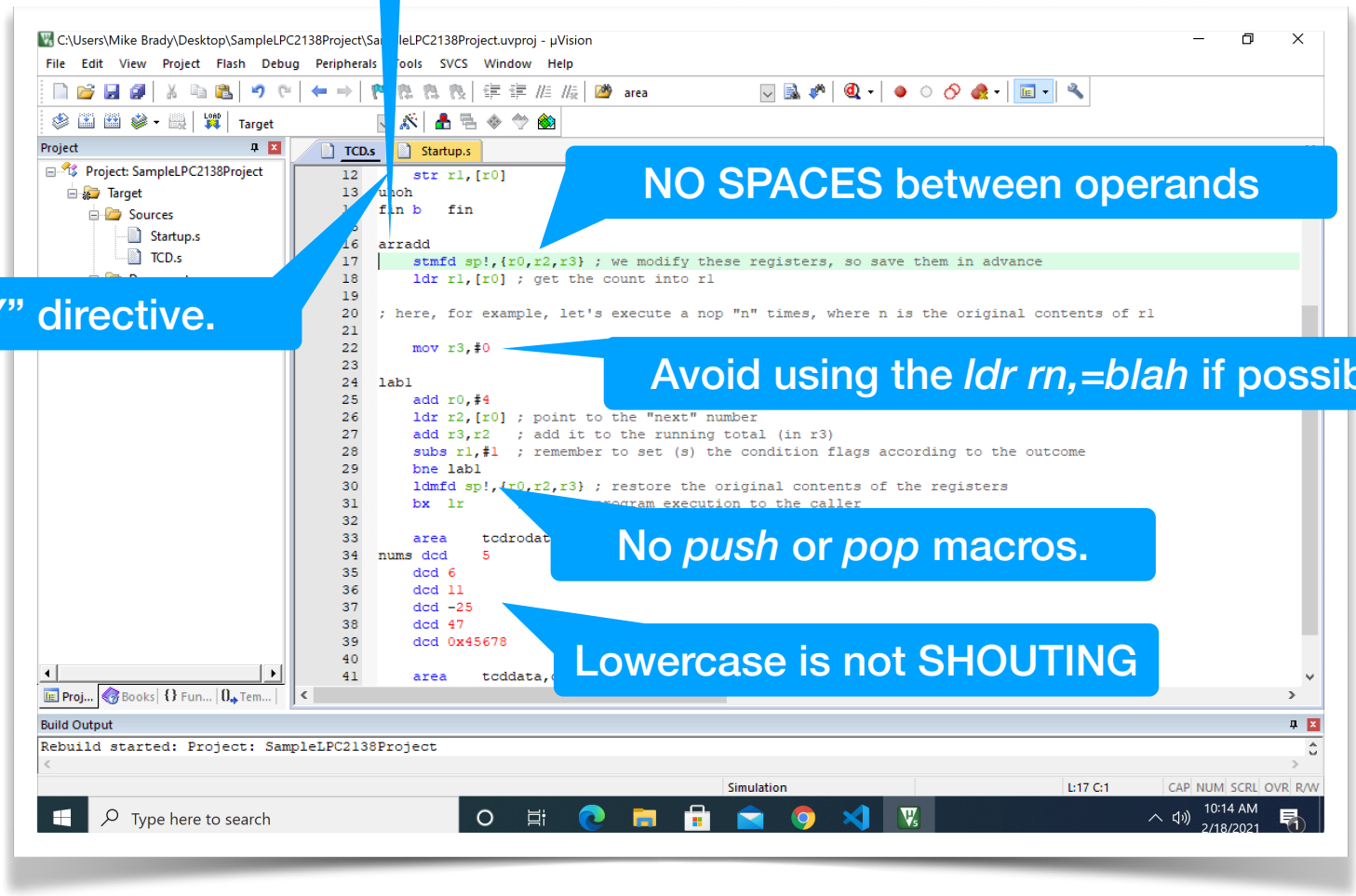
NO SPACES between operands

No “ENTRY” directive.

Avoid using the *ldr rn,=blah* if possible

No *push* or *pop* macros.

Lowercase is not SHOUTING



Input/Output

- So, how can a thing as abstract as program, running on a computer, interact with the "outside world"?
  - How can it interact with parts of the outside world, both to *sense* them and to *affect* them?
- Imagine a program running in a Von Neumann machine. It's a pretty abstract idea.
  - The mathematical/logical idea of an algorithm is converted into a sequence of physical actions taken by devices — in this case the physics is electronics. Nothing much seems to happen on the outside. Energy is consumed by the electronic devices.
  - The concept of time passing is just about there — one instruction follows another in time, but exactly how long things take is not part of the "model".
  - The running program is really pretty cut off from things in the outside world (the world outside the CPU/memory/bus).

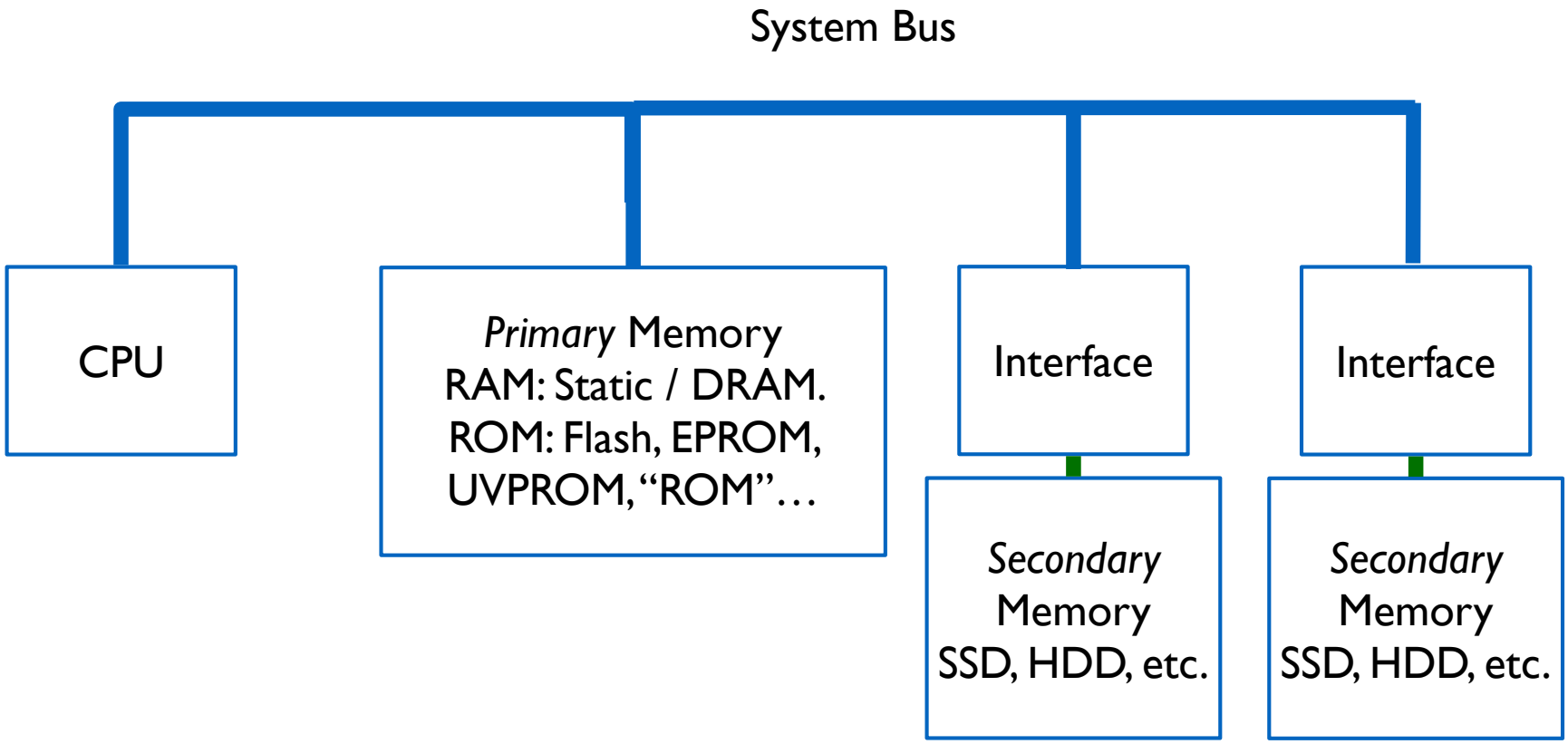
Solution (1) — Output

- The abstract ideas we think of as bits — binary digits — have a physical manifestation in the physical (electronic) device. Thus, as physical effects, they can be used as electronic signals to control electronic devices. But this can work both ways — certain (electronic) physical effects can be interpreted as binary digits.
  - Thus, a bit pattern, as well as representing, say, a number, could also represent a pattern of electronic signals that affect or command the operation of a device.
  - This (binary digit(s) → physical manifestation) mapping from the abstract program world to the physical one is the basis for enabling a program to effect (i.e. to do) something in the outside world.

Solution (2) — Input

- Going in the opposite direction, physical phenomena can be interpreted as binary digits.
  - A group of suitable electronic signals can be interpreted as a group of binary digits. (This already happens in DRAM, where the presence or absence of a charge in a DRAM call is taken as a '1' or a '0'.)
  - This (physical phenomenon → binary digit(s)) mapping from the physical world to the program's abstract world is the basis for enabling a program to detect something in the outside world.

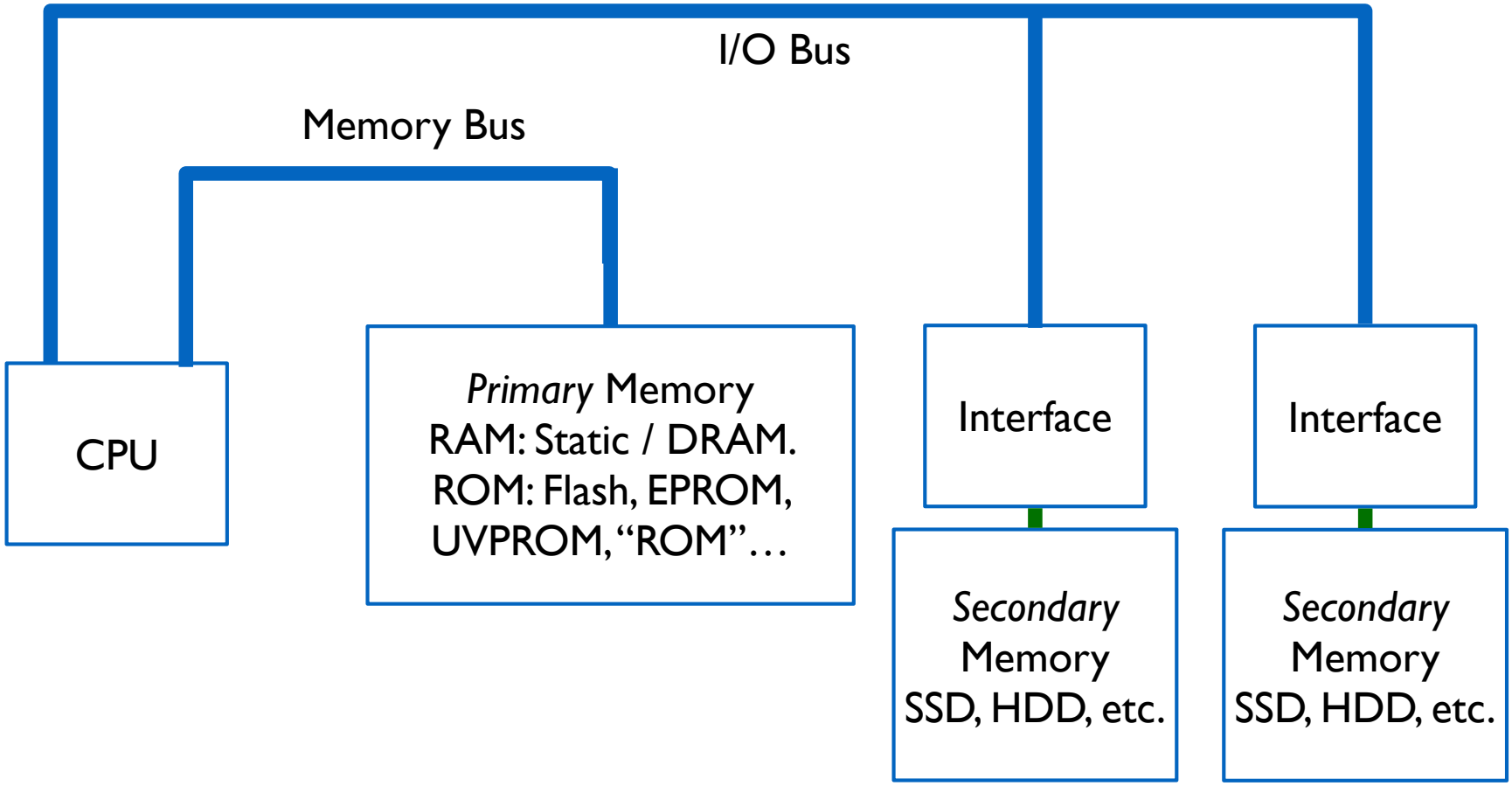
Computer with an Memory Mapped I/O



Output & Input Interfaces

- Specialised input circuitry for connecting signals emanating from electronic devices to binary digits, and
- Specialised output circuitry for connecting binary digits to electronic signals for controlling some circuitry in the outside world are called interfaces.
- However — just as with memory, where you need to know where bits are to be found or stored — so with interfaces, you need to know, in the "inner world" of the Von Neumann computer, where these interface bits are to be found.
- Two common approaches:
  - An Input/Output Bus
  - Memory Mapping

Computer with an Input/Output Bus



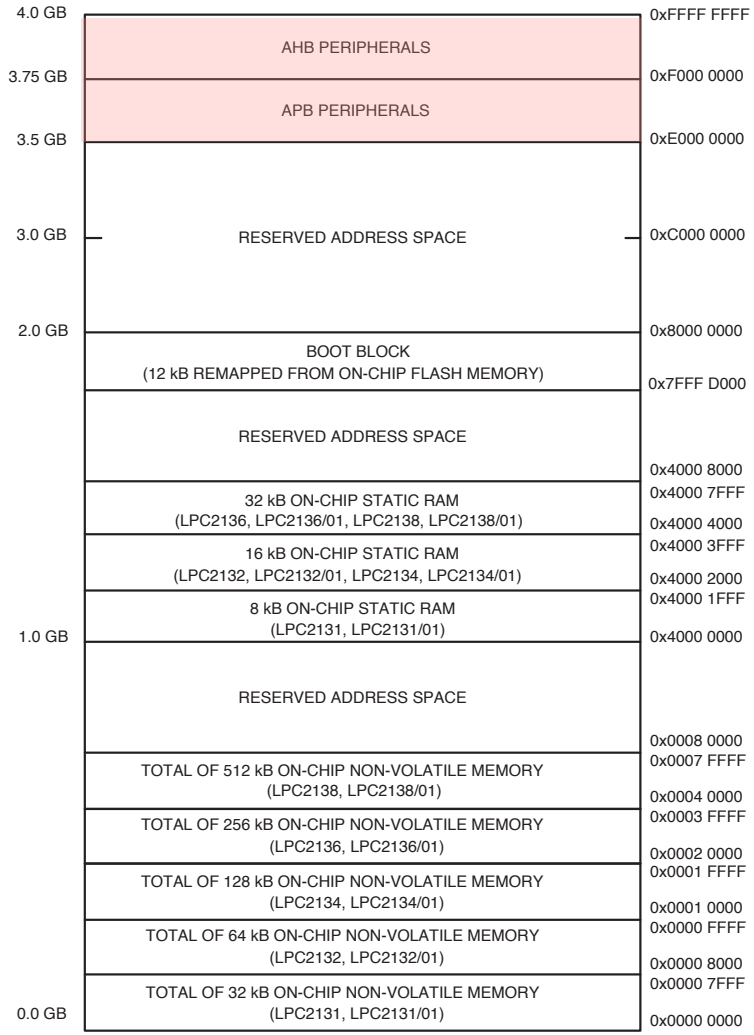
I/O Bus

- An I/O Bus has some advantages
  - The number of interfaces (aka "I/O Devices") on an I/O bus is likely to be small, so a small number of bits (aka "I/O Device Addresses") needed to specify the interface needed in an I/O transaction.
  - Both the I/O Bus and the Memory Bus can be busy simultaneously, potentially improving performance.
- Disadvantages
  - Extra instructions needed to use the I/O bus — "I/O instructions"
  - Somewhat more difficult — conceptually at least — to organise Direct memory Access.

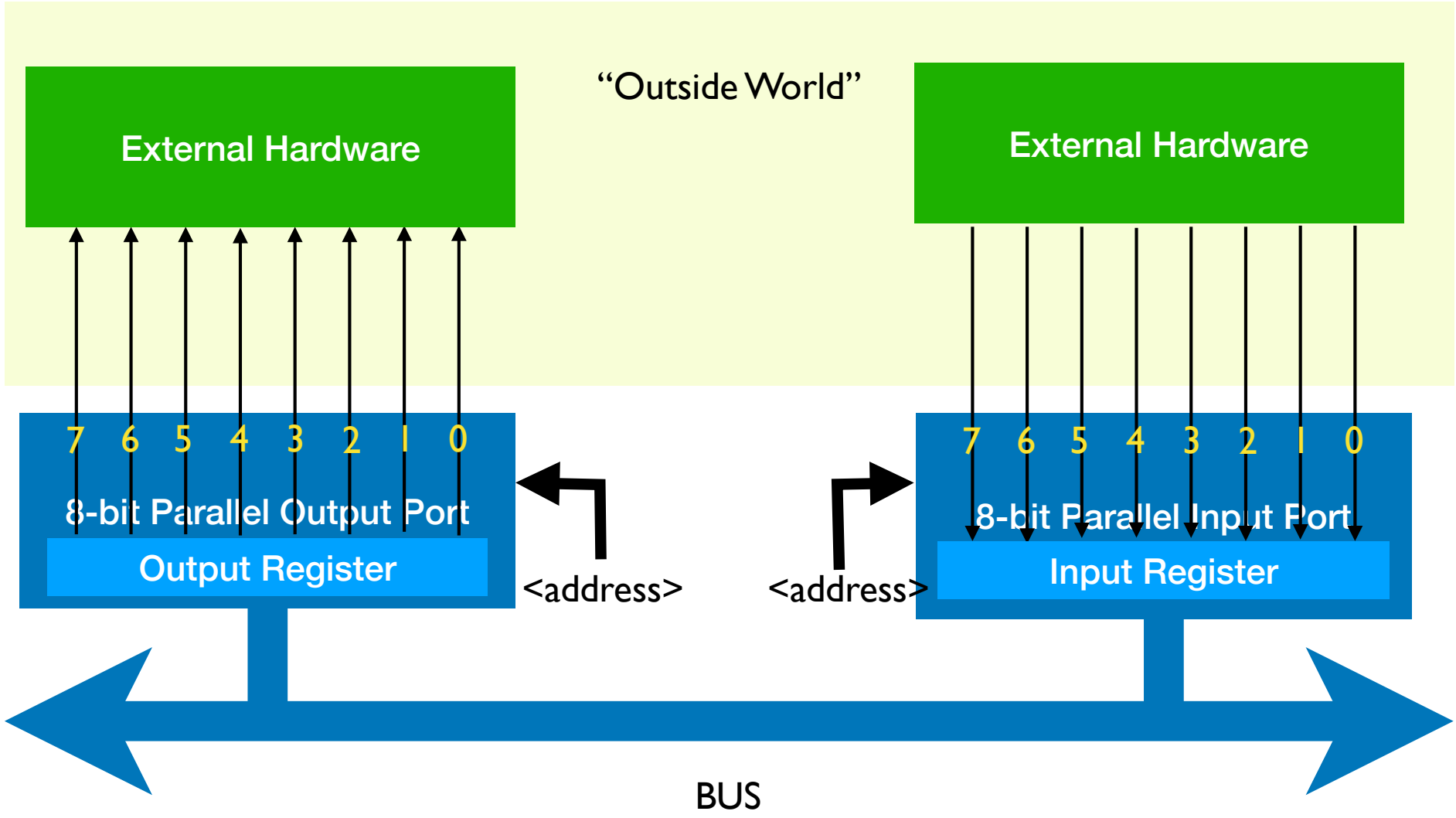
Memory Mapped I/O

- I/O Devices are given locations in what would normally be thought of as the memory space of the computer.
  - That is, I/O devices are mapped to (small numbers of) memory locations — hence the term "Memory Mapped"
- Thus the address space of a computer can now be populated by:
  - RAM
  - ROM
  - I/O Devices

I/O Device in the LPC2138 Address Space



Example of Two Memory-Mapped Ports



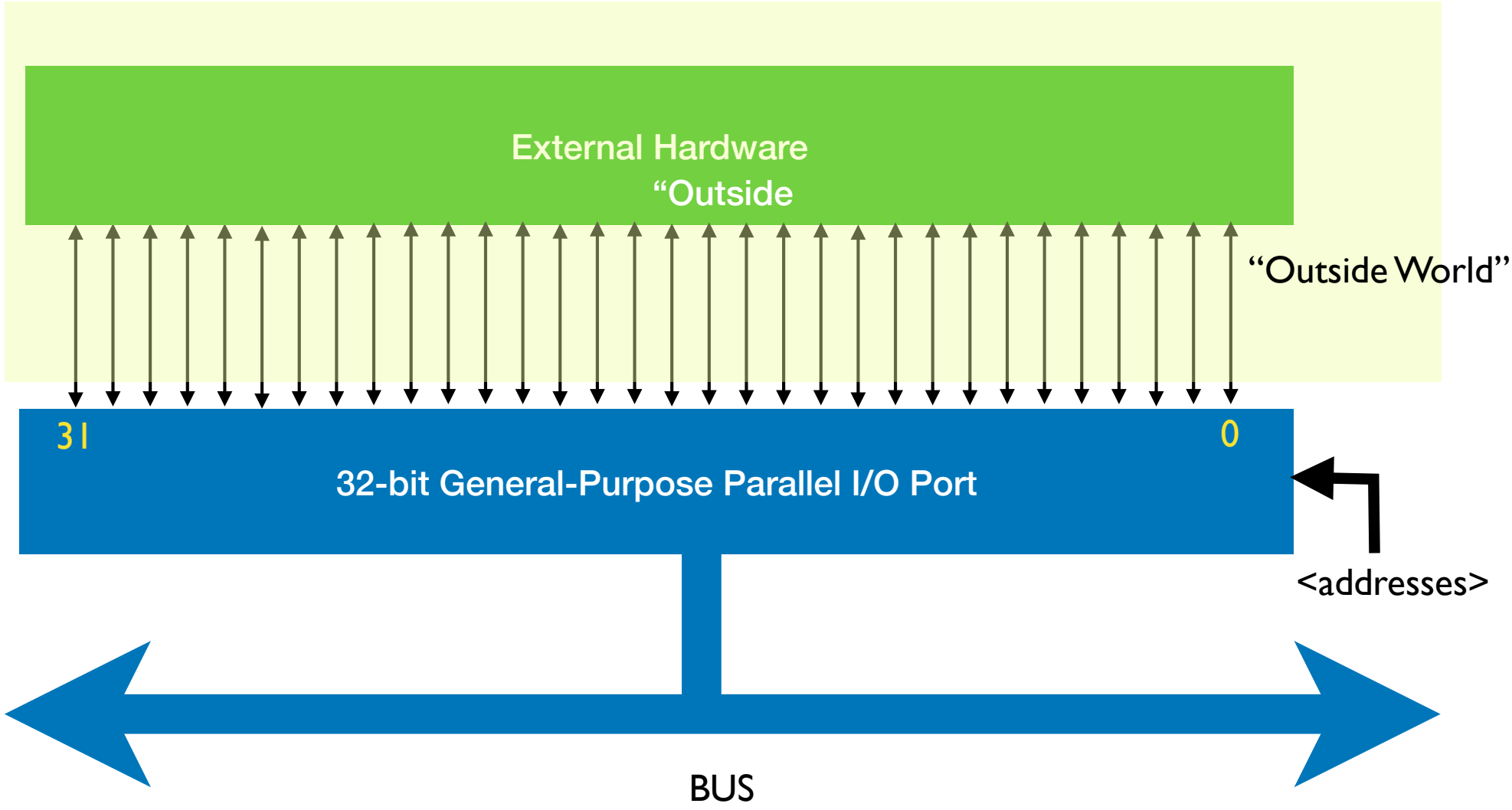
The external hardware is part of the "Outside World"

Using These Ports

- Input
  - Information coming from external hardware is processed electronically and made available as binary data in the Input Register (which will have some address in memory)

- Output
  - Data placed in the Output Register (which will have some address in memory) will be processed by the electronics of the output port and made available to external hardware.

GPIO


External Hardware
"Outside
"Outside World"
31
0
32-bit General-Purpose Parallel I/O Port
<addresses>
BUS

GPIO — Some Details

- The ARM GPIO* is a 32-bit parallel port.
  - Instead of an 8-bit Output Register and an 8-bit Input Register, the GPIO has four 32-bit registers.
- Each bit is thought of as being connected to one electrical "pin" on the chip. So sometimes the bits are called pins.
- Each bit can be an Input or an Output.
  - The direction of the each bit is determined by a corresponding bit in a 32-bit "Data Direction Register"
- The value of each bit can be read or written in a 32-bit "Pin Register".
- Output bits can be set (to 1) by setting the corresponding bit in the "Set Register"
- Output bits can be cleared (to 0) by setting the corresponding bit in the "Clear Register"

Not sure where the acronym came from, possibly from ARM.

Recursion Karma

Recursive Factorial in C

```c
//  Created by Mike Brady on 12/03/2021.
//
#include <stdio.h>
#include <inttypes.h>

uint64_t fact(uint32_t n) {
  uint64_t response;
  if (n == 0)
    response = 1;
  else
    response = n * fact(n-1);
  return response;
}


int main(int argc, const char * argv[]) {
  uint32_t s = 31;
  printf("The factorial of %u is %" PRIu64 ".\n", s, fact(s));
//  printf("The factorial of %u is 0x%" PRIx64 ".\n", s, fact(s)); //hex
  return 0;
}
```

Input/Output Programming Techniques

- Two general approaches (apart from trivial situations such as in the Sample Project)
  - *Polling* — this is where the CPU continually checks ("polls") for the opportunity or ability to do I/O.
    - The responsibility for detecting the I/O opportunity rests with the CPU, so it must devote resources to detecting the opportunity.
    - Conceptually simple, but resource intensive and wasteful and potentially awkward and inflexible.
    - Widely used.
  - *Interrupt Driven I/O* — where the interface hardware detects an opportunity to do I/O and signals the CPU
    - The responsibility for detecting the I/O opportunity rests with the interface, so the CPU can continue to do something else, or to sleep.
    - Much more flexible to implement, more efficient use of the CPU, potentially faster.
    - Requires more CPU infrastructure, organisation and signalling.
    - Very widely used.

Let's do some Polled I/O on µVision

- Write a tiny piece of program to poll Bit 24 of GPIO1 until it becomes zero.

Polling Advantages

- Easy to understand
- Easy to implement
- No special extra hardware facilities needed
- Thus widely used in, e.g. low-level diagnostic tools.

Polling Disadvantages

- Very hard to incorporate in a system that is doing other things.
  - If you are doing something else, e.g. scientific computation, you must periodically do some polling, so you need to add the polling code to the application.
  - You may not even know in advance what that application is.
  - How do you deal with a situation where the hardware is modified, e.g. an extra device that also need polling?
- Inefficient: CPU must poll frequently so as not to miss anything.
  - Higher energy usage
  - CPU's computational power wasted

Lab 7

- Take a 32 bit unsigned binary number in memory and display each digit of its value in decimal on GPIO1 Pins 3, 2, 1 and 0.
- Each number should appear for about a second, starting with the most significant.
- The last digit should be followed by 1,1,1,1 to signify the end of the display.
- The display should then start over again.
- Note — the ARM7 doesn't have a divide instruction.
- Make sure you program's calculations are done efficiently.
- 0x00000025 → 37 → 0b0000 0000 0000 0000 0000 0000 0010 0101

Advanced Issues

- How does the system respond to events?
  - Power up
  - External events
    - How quickly?
  - Errors
- How does the system protect programs from other programs?
- Can the CPU's instruction set be extended in any way?

Advanced Issues — Framework for Solutions

- Exceptions
  - Unforeseen or unpredictable occurrences, including signals from outside, error conditions, unimplemented instructions, and more, are bundled together as "exceptions" and have special hardware to handle them.
- Modes
  - The CPU has a number of different categories of operation, each one backed up by hardware-implemented "modes". Exceptions can cause the CPU to change mode.
- Register Banks
  - Different modes have private copies of certain important registers.

Exceptions

- The processor hardware does more than just execute instructions.
  - It monitors for events and situations that are:
    - unusual, or
    - unanticipated.
- These are collectively known as exceptions, so we can talk about:
  - exception monitoring hardware and also about
  - *exception handling*, both by the processor itself and by software ("exception handlers").

Exceptions – Details

- Exceptions arise whenever the normal flow of a program has to be halted temporarily, for example to service an interrupt from a peripheral.
  - (So, an "interrupt request" is a type of exception.)
- Before an exception can be handled, the current processor state must be preserved so that the original program can resume when the handler routine has finished.
  - RESET is different. (Why?)

Modes

- The processor has different *modes* of operation.
- Exceptions generally cause the processor to jump into another mode.
- Certain instructions can also cause a change in mode.

Mode	Description
User	Unprivileged, normal.
FIQ	Fast Interrupt
IRQ	Interrupt
Supervisor	Privileged
Abort	Memory access violation
Undef	Handles undefined instructions
System	Privileged, but no *register bank* switch

When an Exception occurs:

- Instruction execution stops.
- Depending on the cause of the exception:
  - The *mode* of the CPU will be changed.
  - A minimal part of the CPU's state is usually preserved:
    - The PC* is copied to the new mode's LR and the CPSR is copied to the new mode's SPSR.
  - Some bits (e.g. the I and F bits) in the CPSR may be changed.
- Instruction execution will resume at a particular location.
  - That location is called the exception's "vector", and is part of the *Exception Vector Table* in low memory.

**Due to pipelining, the PC may not have exactly the address of the next instruction — it may need to be adjusted for later use*

Exception Vector Table

- An Exception Vector Table entry is four bytes long, long enough for one instruction!
- Except for FIQ, it's got to be a branch or jump type instruction.
- FIQ is at the end of the table, so its program can flow sequentially from its Exception Vector Table entry — faster!

Address	Exception	Mode on Entry	Notes	Priority
0x00000000	Reset	Supervisor	CPSR.I and CPSR.F both set	1 (Highest)
0x00000004	Undefined Instruction	Undefined		6
0x00000008	Software Interrupt	Supervisor		6
0x0000000C	Abort (Prefetch)	Abort	Caused by a memory error	5
0x00000010	Abort (Data)	Abort	Caused by a memory error	2
0x00000014	Reserved	_		
0x00000018	IRQ	IRQ	CPSR.I set	4
0x0000001C	FIQ	FIQ	CPSR.F set	3

"CPSR.I" means the I bit of the Current Processor Status Register

Register Banks

USER/SYSTEM
R0
R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
R11
R12
R13 (SP)
R14 (LR)
R15 (PC)
CPSR

SUPERVISOR
R0
R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
R11
R12
R13_SVC
R14_SVC
R15
CPSR
SPSR_SVC

ABORT
R0
R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
R11
R12
R13_ABT
R14_ABT
R15
CPSR
SPSR_ABT

UNDEFINED
R0
R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
R11
R12
R13_UNDEF
R14_UNDEF
R15
CPSR
SPSR_UNDEF

IRQ
R0
R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
R11
R12
R13_IRQ
R14_IRQ
R15
CPSR
SPSR_IRQ

FIQ
R0
R1
R2
R3
R4
R5
R6
R7
R8_FIQ
R9_FIQ
R10_FIQ
R11_FIQ
R12_FIQ
R13_FIQ
R14_FIQ
R15
CPSR
SPSR_FIQ

The Exception Handler

- The exception vector normally branches to a piece of code called the *Exception Handler*.
- The Exception Handler deals appropriately with the cause of the exception.
  - If the cause of the exception is an interrupt or fast interrupt request, the exception handler must deal the request so that the request is removed. (Why?)
- Usually, the exception handler ends by returning program execution to the program that was stopped (i.e. the program that was "excepted").
- To do this properly, the exception handler must restore the *entire* context of the excepted program completely – it must be totally "well behaved"!
  - This includes the state of the CPSR when the exception occurred.
- Thus, an exception handler usually must save every register it uses and restore them prior to exit.

When an Exception ends normally:

- The PC is restored from the mode's Link Register, possibly adjusted slightly.*
- The CPSR is restored from the current mode's SPSR.
  - This will also restore the interrupt flags to what they were on entry.

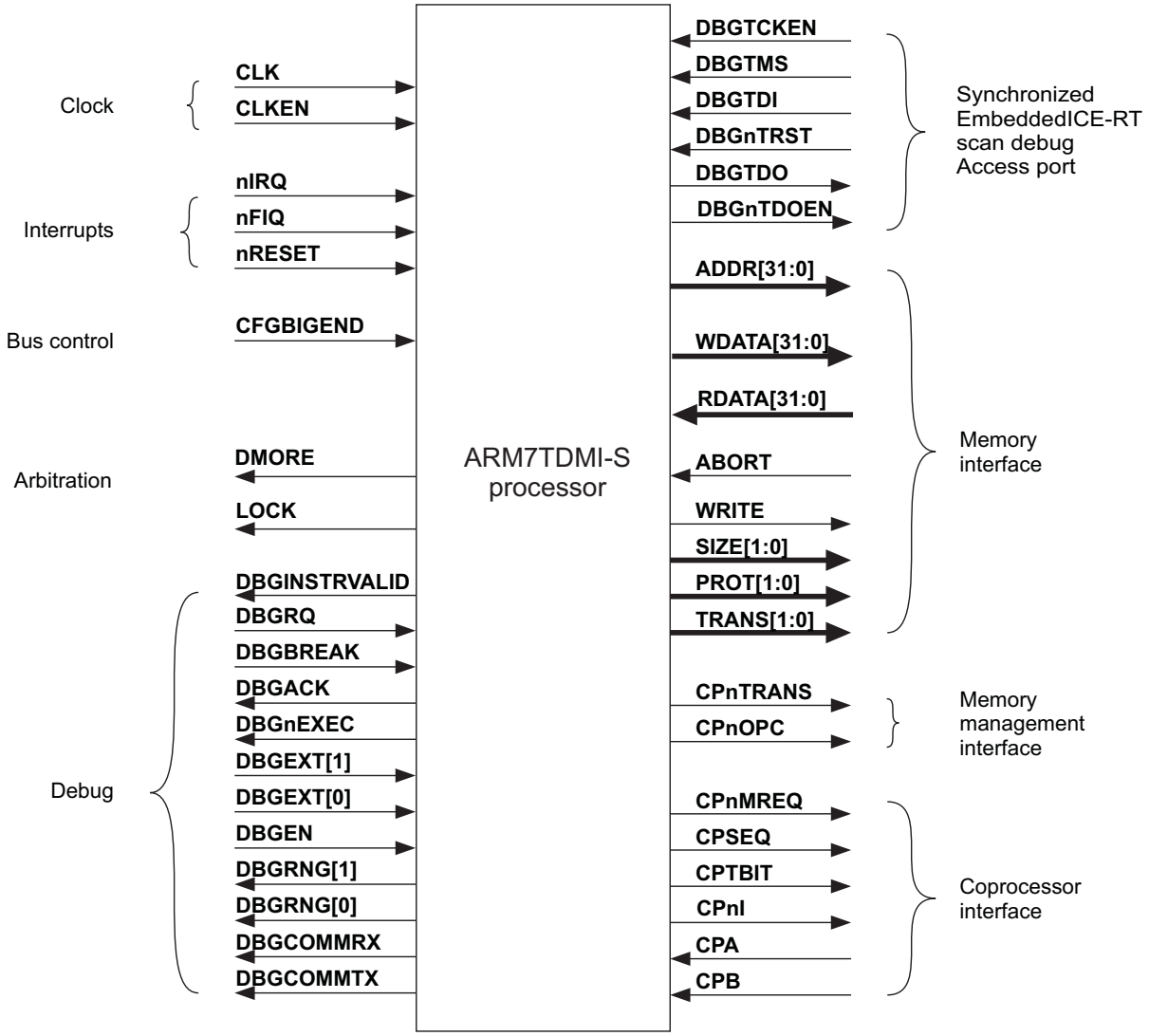
Modes and "Privilege"

- Different modes have different levels of abilities and facilities (aka "privilege").
- The ARM has two levels of privilege.
- The user mode has the lower level of privilege, it is an *unprivileged mode*.
- All other modes have the higher level — they are *privileged modes*.
  - Some instructions are "privileged" and some resources are "protected" and can only be executed or accessed with the CPU in a privileged mode.
  - Generally, instructions or accesses that could change the overall state of the machine can only be used or accessed in a privileged mode:
    - An attempt to change mode from the user mode by changing the mode bits in the CPSR is ignored.

Privilege Enforcement Outside the CPU

- So, for example, one could ensure that access to peripheral interface addresses was protected, i.e. that user-level programs could not access it. (Why?)
- With an extra device called a Memory Management Unit (MMU), one could:
  - Specify how addresses generated by a user program are mapped to real addresses.
  - Only allow user mode accesses to a certain range of real addresses.
  - … and thus prevent user mode access from straying outside a permitted set of real addresses, offering *hardware-level memory protection* for separate programs ("processes") running in the same machine.
  - MMUs can also help implement Virtual Memory, typically used by operating systems to simulate the availability of very large amounts of memory to processes.
- BTW, the LPC2138 does not have an MMU and does not enforce any mode-based memory segregation.

The Processor Core



Privilege Enforcement Outside the CPU

- As well as Address and Data, R/W' etc., the ARM bus carries two "PROT" lines.
- Thus, hardware connected to the bus can tell the mode (it's really the level of privilege) of the access attempt:

PROT[1:0]	Mode	Opcode or data
00	User	Opcode
01	User	Data
10	Privileged	Opcode
11	Privileged	Data

Register Banks

Modes						
	Privileged modes					
		Exception modes				
User	**System**	**Supervisor**	**Abort**	**Undefined**	**Interrupt**	**Fast interrupt**
R0	R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7	R7
R8	R8	R8	R8	R8	R8	R8_fiq
R9	R9	R9	R9	R9	R9	R9_fiq
R10	R10	R10	R10	R10	R10	R10_fiq
R11	R11	R11	R11	R11	R11	R11_fiq
R12	R12	R12	R12	R12	R12	R12_fiq
R13	R13	R13_svc	R13_abt	R13_und	R13_irq	R13_fiq
R14	R14	R14_svc	R14_abt	R14_und	R14_irq	R14_fiq
PC	PC	PC	PC	PC	PC	PC
CPSR	CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
		SPSR_svc	SPSR_abt	SPSR_und	SPSR_irq	SPSR_fiq

indicates that the normal register used by User or System mode has been replaced by an alternative register specific to the exception mode

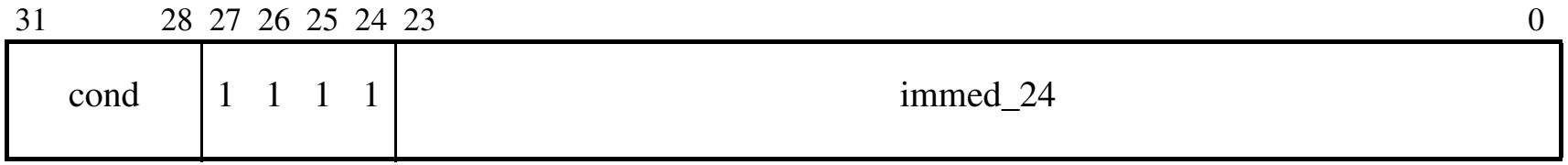
Source: Arm Document ARM DDI 0100I, pp A2-5

Exceptions / Modes / Register Banks

- Together used to provide advanced features:
  - Ability to respond to the outside world
    - Interrupts / Fast Interrupts
  - Ability to provide security
    - Supervisor/System Mode vs User Mode
  - Ability to provide limited error detection
    - Memory access violation
    - etc.
  - Ability to provide OS Support
    - [Parts of] the OS can run in Supervisor Mode.
    - User-level programs can ask to get to the supervisor mode via SWI (SoftWare Interrupt request) instruction.

SWI Instruction

```
SWI{<cond>}  <immed_24>
```



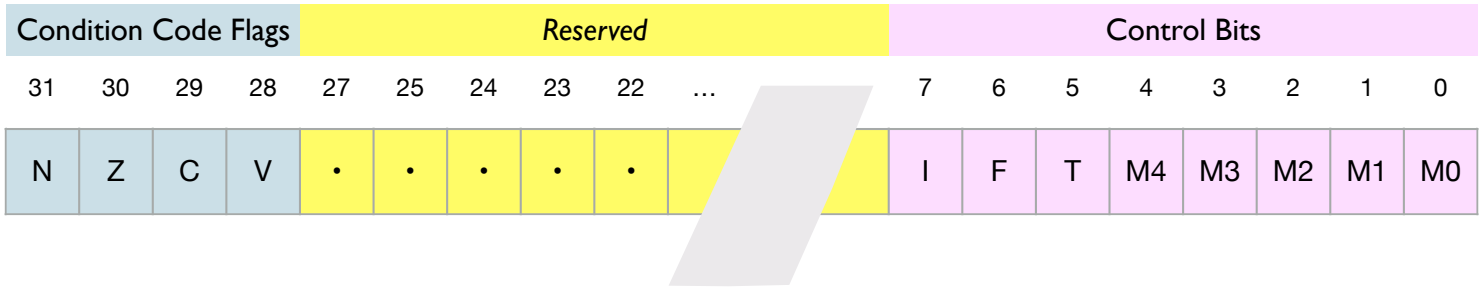
- Aka "SVC" (probably from "SuperVisor Call", a possible use of the SWI instruction.)
- Usage: SVC <24-bit operand>, e.g. SVC 0xABCD89

Source: ARM DDI 0100I, pp A3-29

SWI Instruction

- The SWI (aka SVC) instruction causes a Software Interrupt exception,
  - The CPU changes to the Supervisor mode
  - The SWI Exception Handler runs
  - The least significant 24 bits of the instruction can be accessed as arguments.
- Probably designed for helping to implement *system calls* to an OS, where the CPU moves to a supervisor mode and the 24 bits of the instruction used as a selector.
- The SWI is an example of a slightly paradoxical *elective* exception — an exception that is requested.
  - Useful because it changes the mode of the CPU in a controlled way.
  - The Undefined Instruction exception could also be viewed/used as an elective exception.

Current Program Status Register (CPSR)



- Conditions typically change quickly, possibly after every instruction.
- Program Status changes relatively slowly.
- M4–M0 represent the Processor's Mode
- T is the Thumb Instruction Enable Bit
- I & F are the Interrupt or Fast interrupt disable bits

Mode Bits M4 – M0 in the CPSR

M4–M0	Mode
10000	User
10001	FIQ
10010	IRQ
10011	Supervisor
10111	Abort
11011	Undefined Instruction
11111	System

Processor Mode

Note: a user mode program can not change the values of the Mode Bits

“Normal” Return from an Exception

- The full context of the “excepted” program must be restored
  - All registers must be perfectly restored (except for elective exceptions).
  - The PC must be restored (though it’s a little bit complicated*)
  - The CPSR must also be restored.
    - This also restores the state of the program, including its mode, I & F bits, etc.

*Exception Return Address 'Adjustment'

- To understand how exception return address need to be adjusted, consider what happens when the exception occurs:
  - If the source of the exception is from program execution — SWI, Undefined Instruction, Instruction Prefetch Abort — then Exception Handling occurs 'smoothly'.
  - If the cause is a side-effect of instruction execution, then that instruction will have to be re-run if/ when the cause is fixed.
  - If the cause has an external source (e.g. IRQ/FIQ), then the current instruction is allowed to complete and exception handling starts in place of the next instruction, which must be re-run when/if the exception returns to this code. That is, the next instruction is "usurped" by the externally source exception, so it must be re-run.
- Thus, depending on the type of the exception, the value of the PC may have to be adjusted before returning program execution to the program that was excepted. See next slide.

*Exception Exit Instruction

- Instruction to restore appropriate PC value at the end of an exception.

Cause of Exception	Return Instruction
SWI	MOVS PC,R14
UDEF	MOVS PC,R14
FIQ	SUBS PC,R14,#4
IRQ	SUBS PC,R14,#4
PABT (Prefetch)	SUBS PC,R14,#4
DABT (Data)	SUBS PC,R14,#8
RESET	Nothing – but why?

- Note — the fact that the destination register is the PC and the "S" suffix is included means that the CPSR is restored from the mode's SPSR.

Lab 8

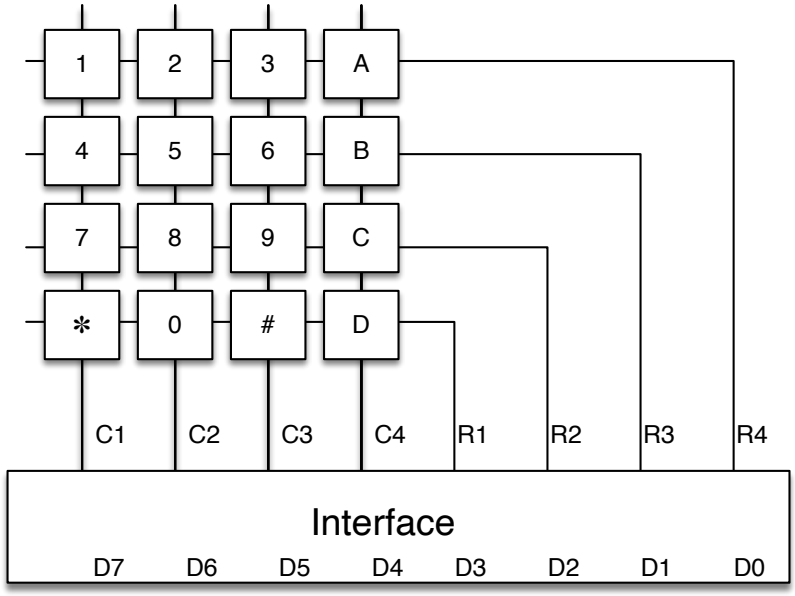
- Write an SWI exception handler to "extend" the instruction set as follows:
  - Taking the 24 bits of immediate operand as three separate bytes, representing:
    - Start Register (Bits 23 to 16)
    - End Register (Bits 15 to 8)
    - Operand (Bits 7 to 0)
  - The emulator should multiply all the registers from the Start Register up to the End Register by the Operand, taken as an unsigned 8-bit number. The result should be fitted into the same register.
    - If the Start Register is after the End Register, do nothing.
    - Registers can range from R0 to R5, inclusive.

Lab 9

- Sample Solution of Assignment 1

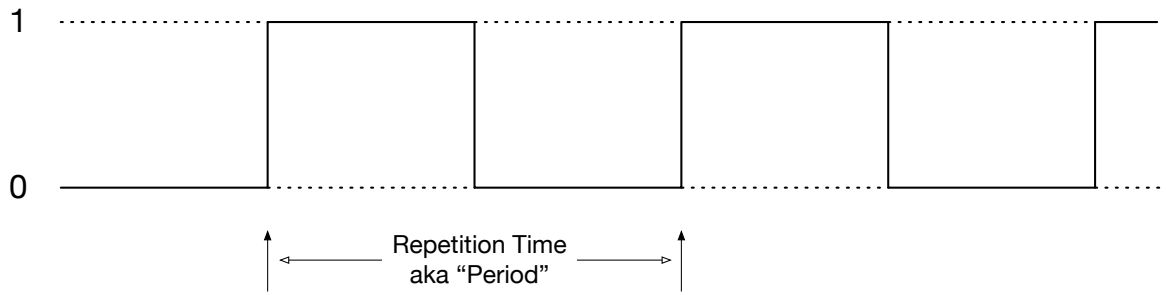
Tutorial

- Assume this interface is at 0xE8D6048A.
- Write a subroutine to return the ASCII code of the key in R0.



The Timer Peripheral

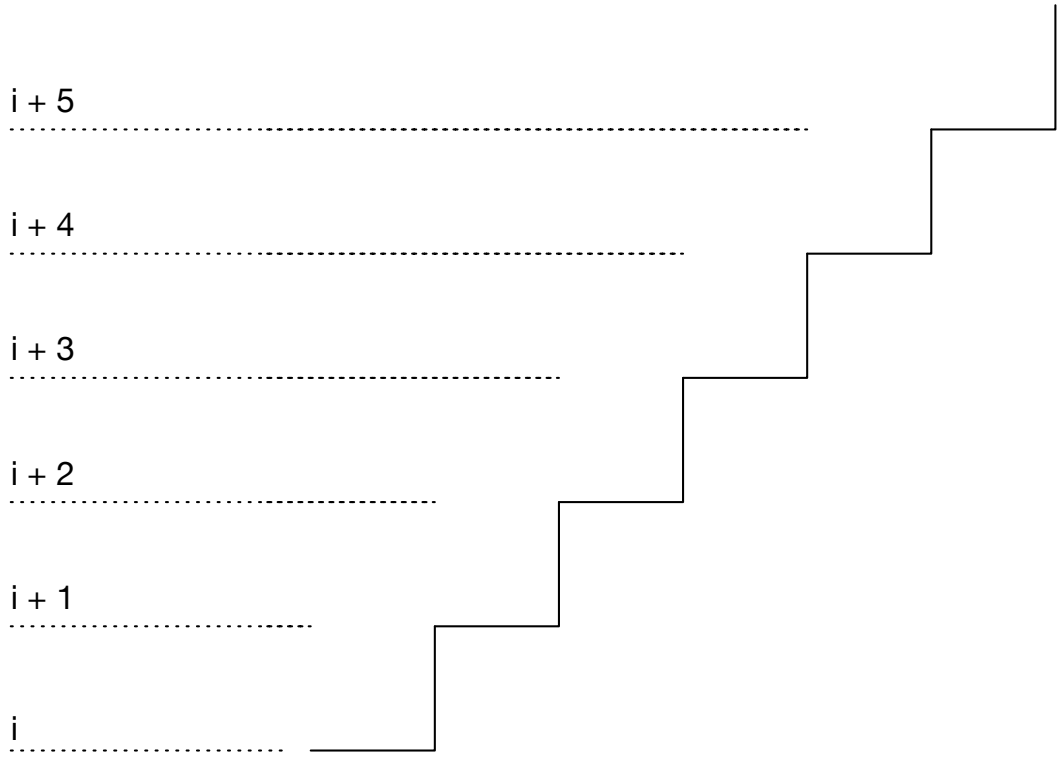
- Hard or impossible to time things accurately using instruction loops.
- One solution is to use a "clock" — an external oscillator that has a highly accurate frequency or period.
  - E.g. typical a crystal controlled clock will be accurate to about 10 parts per million.



- The LPC2138 has two clock peripheral devices, T0 and T1, that can be set up like this.
- We can set a clock's period in "ticks" of the Arm Peripheral Bus clock.
- The APB clock runs at 14.745600 MHz, thus 14,745,600 APB ticks per second.

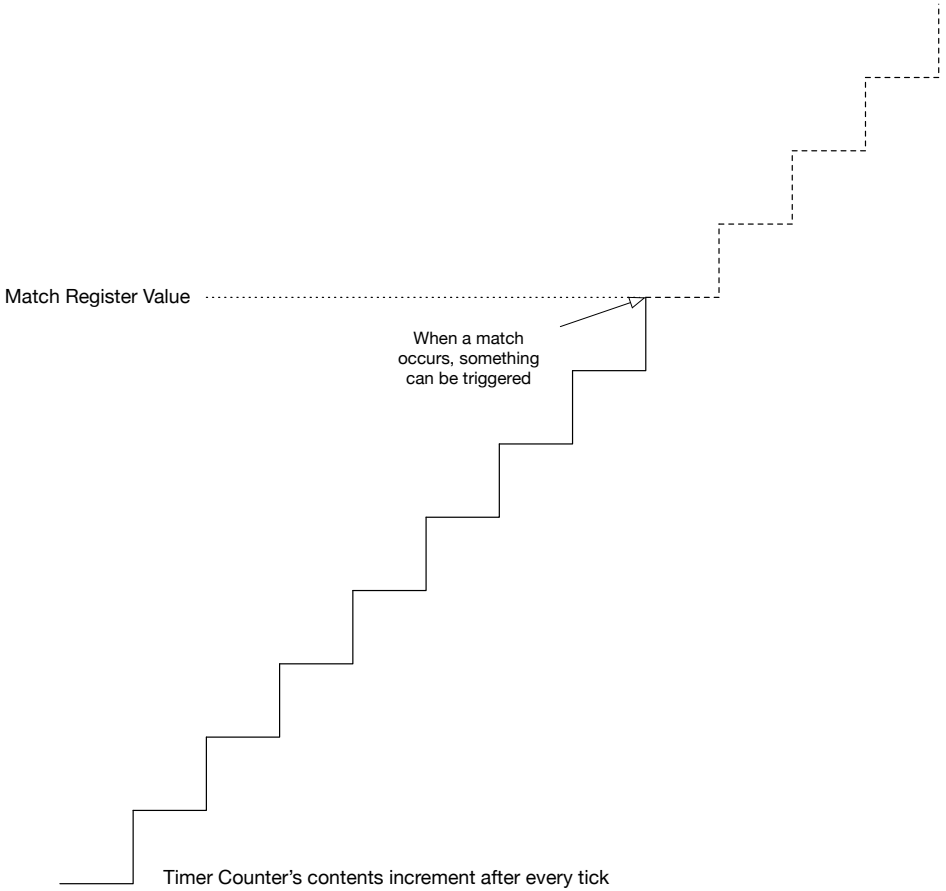
The Timer Peripheral

- Each timer has a built-in Timer Register (TR).
- Every time a period elapsed (i.e. for every "tick"), the TR is incremented.
- Thus the TR's value rises, a bit like a staircase.



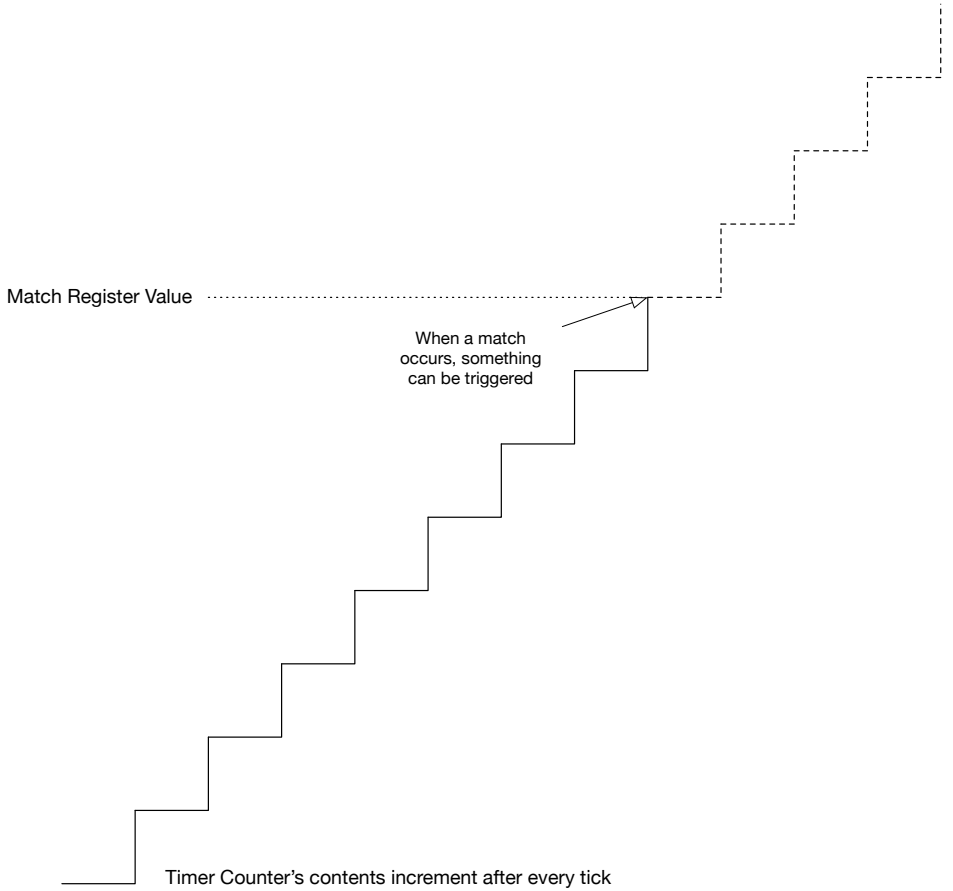
The Timer Peripheral

- The computer could set the period, and then could read the Timer Register (TR) to determine how many ticks had elapsed, thereby "tell the time".
- Each timer also has four Match Registers MR0 to MR3
  - Give a MR a value and when the TR reaches that value, something can be done.



The Timer Peripheral

- Suppose we wanted a match every 625 µs, i.e. 1,600 times per second.
- Ticks per event should be 14745600/1600.
- We need to "program" Timer0 to reset the TC and generate an interrupt. See later…

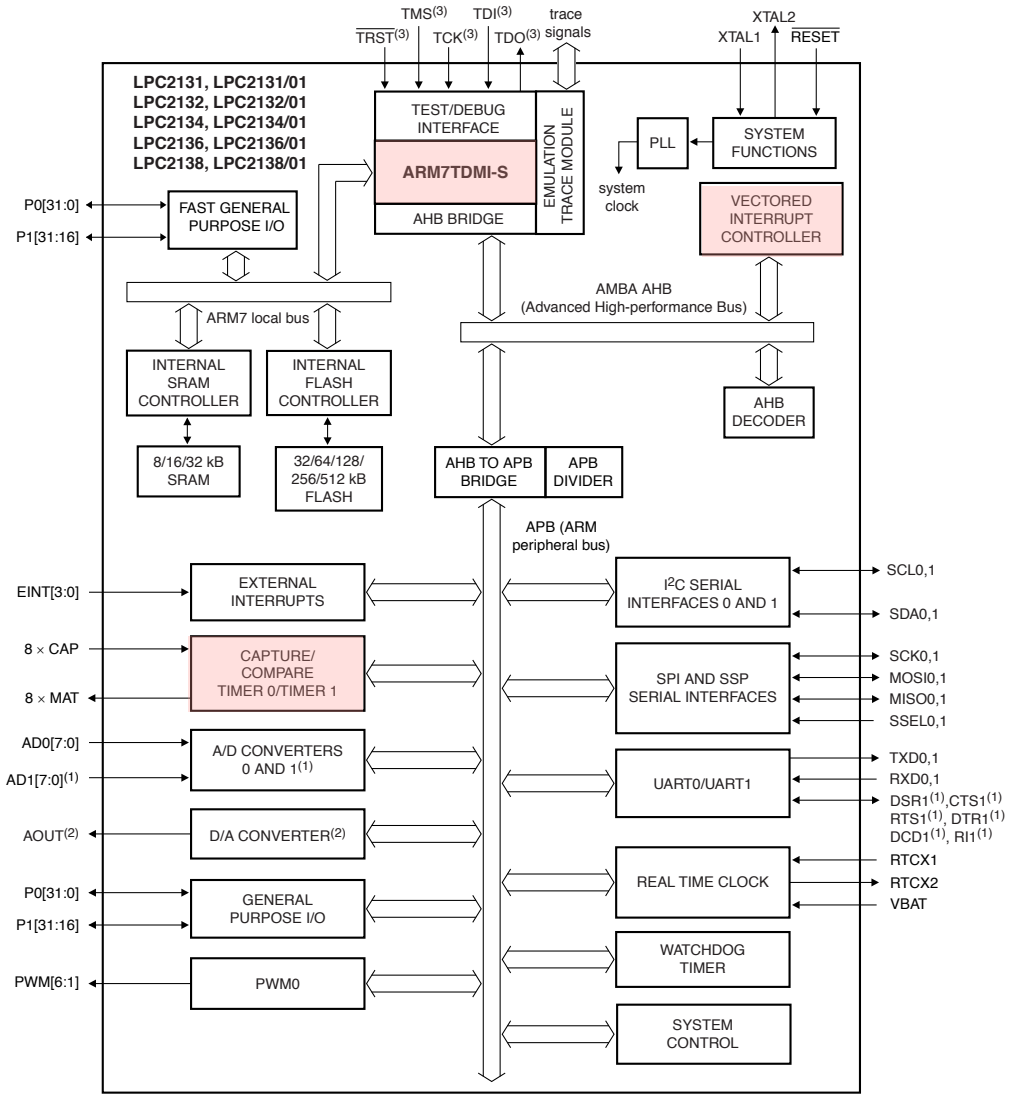


ARM Interrupt Handling

- The ARM7TDMI has just one IRQ line for peripherals to request interrupts.
  - (It also has one FIQ for "fast" i.e. high-priority interrupt requests.)
- So, what happens if you have lots of IRQ* sources?
  - When an IRQ is made, you must determine where the interrupt request originated to decide how to handle the request, i.e. what handler to run.
  - If more than one device has a request pending, how to you organise priority?
  - If this is done in software, essentially by checking every device that could have caused the IRQ, this can take a long time.
  - That is, selecting the correct code to run can take a [very] long time at a critical period — the latency time between when an IRQ is request and an effective response is made.

**we'll just talk here about IRQs but most of this refers also to FIQs*

The LPC2138 Block Diagram



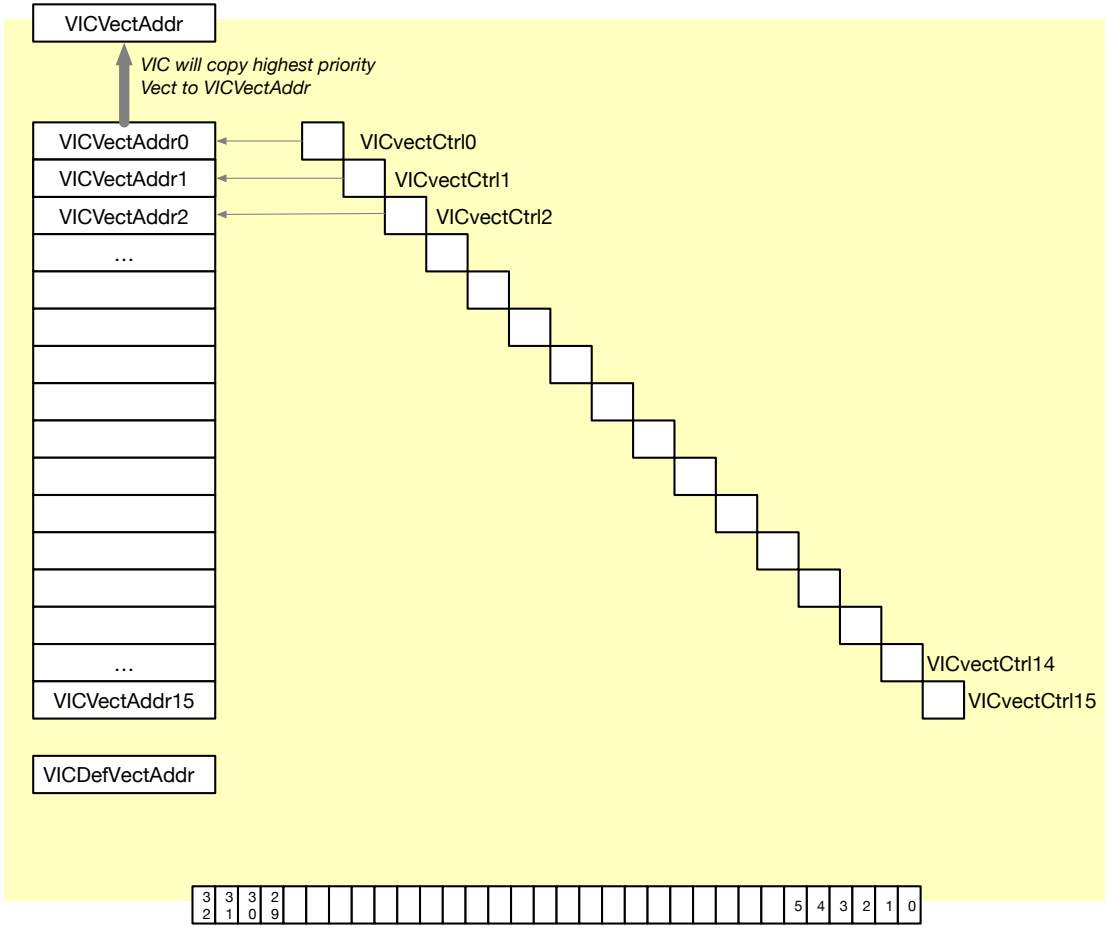
Source: User Manual UM10120 Page 8

Vectored Interrupt Controller

- To solve the problem or responding quickly to interrupt requests, the LPC2138 has a very unusual peripheral, called a *Vectored Interrupt Controller* (VIC).
- The VIC receives all interrupt requests, prioritises them in hardware, makes an IRQ (or FIQ) as appropriate, and then, crucially, makes a "vector" available for the CPU to use to jump to the appropriate handler directly.

The Vectored Interrupt Controller

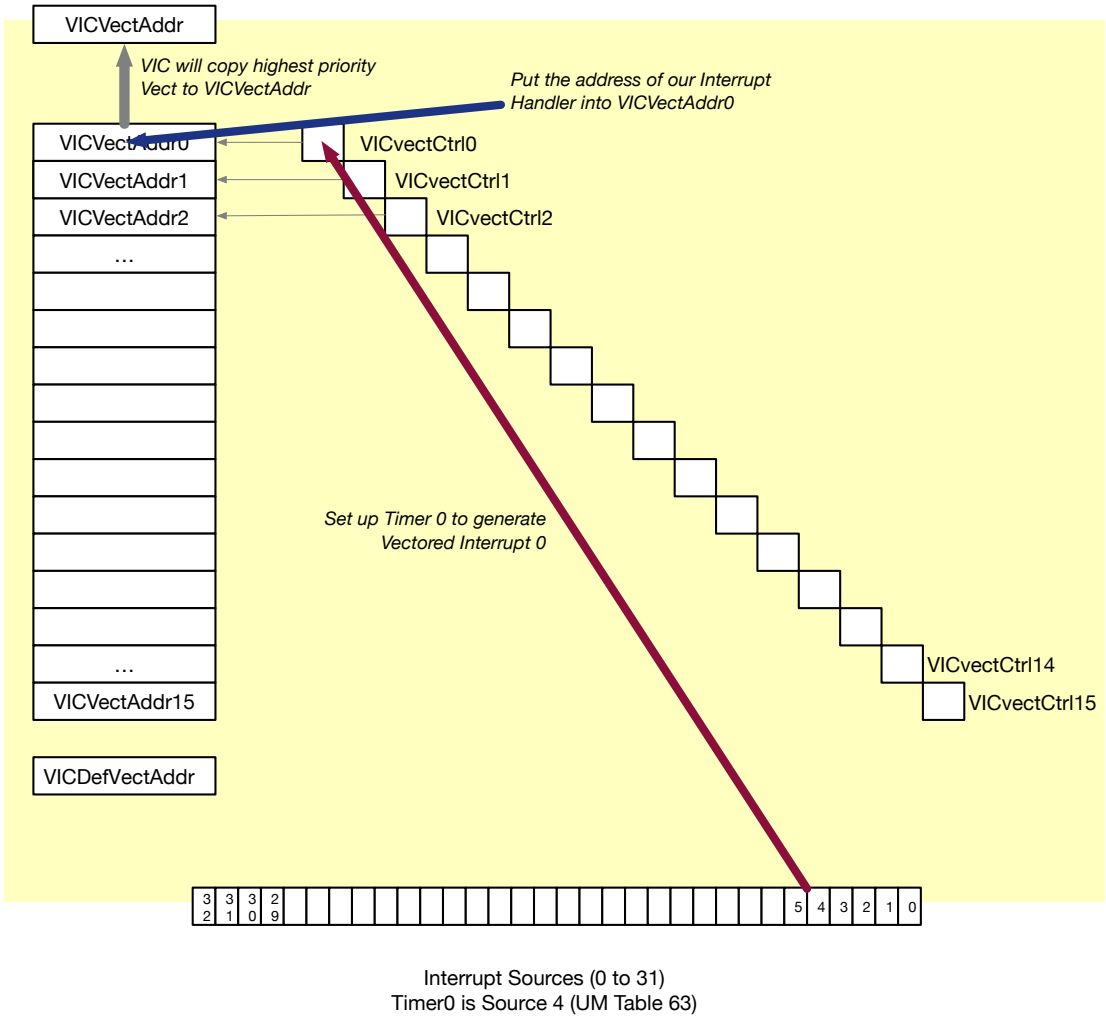
- 32 possible sources of interrupts (not all used on LPC2138).
- Each can be mapped to 16 Vectored Interrupts.
  - Lowest Number → Highest Priority
- Can be non-vectored too.
- For each Vector X:
  - Use VICvectCtrlX to specify source.
  - Use VICVectAddrX to specify handler address.



Interrupt Sources (0 to 31)
Timer0 is Source 4 (UM Table 63)

The Vectored Interrupt Controller

- During setup, write to the relevant VICvectCtrl*N* register to set the source,
- Write the address of the interrupt handler to the relevant VICVectAddr*N* register.
- When an interrupt occurs, the VIC will copy the contents of the highest priority VICVectAddr*N* register to VICVectAddr where it can be read by the CPU interrupt handler.



Interrupt Sources (0 to 31)
Timer0 is Source 4 (UM Table 63)

Some Code – Useful EQUates…

```
415  ; Definitions  -- references to 'UM' are to the User Manual, UM10120
416
417  ; Timer Stuff -- UM, Table 173
418
419  T0   equ 0xE0004000      ; Timer 0 Base Address
420  T1   equ 0xE0008000
421
422  IR   equ 0           ; Add this to a timer's base address to get actual register
423  TCR equ 4
424  MCR equ 0x14
425  MR0 equ 0x18
426
427  TimerCommandReset   equ 2
428  TimerCommandRun equ 1
429  TimerModeResetAndInterrupt  equ 3
430  TimerResetTimer0Interrupt   equ 1
431  TimerResetAllInterrupts equ 0xFF
432
433  ; VIC Stuff -- UM, Table 41
434  VIC equ 0xFFFFF000      ; VIC Base Address
435  IntEnable   equ 0x10
436  VectAddr    equ 0x30
437  VectAddr0   equ 0x100
438  VectCtrl0   equ 0x200
439
440  Timer0ChannelNumber equ 4   ; UM, Table 63
441  Timer0Mask  equ 1<<Timer0ChannelNumber  ; UM, Table 63
442  IRQslot_en  equ 5       ; UM, Table 58
```

Initialise the VIC…

```
444 ; Initialise the VIC
445     ldr r0,=VIC          ; looking at you, VIC!
446
447     ldr r1,=irqhan
448     str r1,[r0,#VectAddr0]   ; associate our interrupt handler with Vectored Inte
449
450     mov r1,#Timer0ChannelNumber+(1<<IRQslot_en)
451     str r1,[r0,#VectCtrl0]   ; make Timer 0 interrupts the source of Vectored Int
452
453     mov r1,#Timer0Mask
454     str r1,[r0,#IntEnable]   ; enable Timer 0 interrupts to be recognised by the
455
456     mov r1,#0
457     str r1,[r0,#VectAddr]        ; remove any pending interrupt (may not be neede
```

Initialise Timer0

```
459  ; Initialise Timer 0
460      ldr r0,=T0            ; looking at you, Timer 0!
461
462      mov r1,#TimerCommandReset
463      str r1,[r0,#TCR]
464
465      mov r1,#TimerResetAllInterrupts
466      str r1,[r0,#IR]
467
468      ldr r1,=(14745600/1600)-1     ; 625 us = 1/1600 second
469      str r1,[r0,#MR0]
470
471      mov r1,#TimerModeResetAndInterrupt
472      str r1,[r0,#MCR]
473
474      mov r1,#TimerCommandRun
475      str r1,[r0,#TCR]
476
477  ;from here, initialisation is finished, so it should be the main body of the mai
```

Very basic Interrupt Handler

```
483  irqhan  sub lr,lr,#4
484      stmfd   sp!,{r0-r1,lr}  ; the lr will be restored to the pc
485
486  ;this is the body of the interrupt handler
487
488  ;here you'd put the unique part of your interrupt handler
489  ;all the other stuff is "housekeeping" to save registers and acknowledge interr
490
491
492  ;this is where we stop the timer from making the interrupt request to the VIC
493  ;i.e. we 'acknowledge' the interrupt
494      ldr r0,=T0
495      mov r1,#TimerResetTimer0Interrupt
496      str r1,[r0,#IR]     ; remove MR0 interrupt request from timer
497
498  ;here we stop the VIC from making the interrupt request to the CPU:
499      ldr r0,=VIC
500      mov r1,#0
501      str r1,[r0,#VectAddr]   ; reset VIC
502
503      ldmfd   sp!,{r0-r1,pc}^ ; return from interrupt, restoring pc from lr
504                  ; and also restoring the CPSR
```

Lab 10

- Build a main program and interrupt handler to time how long a "button" is pressed.
- The timer must be accurate to within 1 millisecond.

Thread Scheduler

- Design* a simpler "scheduler" to enable two separate programs (effectively two separate threads) to share the computer, each getting approximately half the CPU.
- The scheduler should be built around an interrupt handler that is called periodically.
- However, instead of returning to the interrupted program,
  - We arrange for it to "return" to the alternate program.
- Thus, it "returns" to each program half the time.
- So, each program is alternately running and "sleeping"

Thread Scheduler

- How do we do this?
- Why would we do this?
- No special constraints on the programs
  - No registers to avoid!
  - No extra code to write!
  - Thus, developers and their toolchains don't have to be specialised to accommodate the scheduler.

Key Concept – Program *Context*

- The *context* of as running program is everything it uses or accesses:
  - Memory Locations
  - Registers
  - Peripherals and the State of the Outside World
- To stop and start a programme one must *save* the context upon stopping a program…
  - …and *restore* the context upon resuming the program
- This saving/restoring should be completely* invisible to the program

*almost completely!

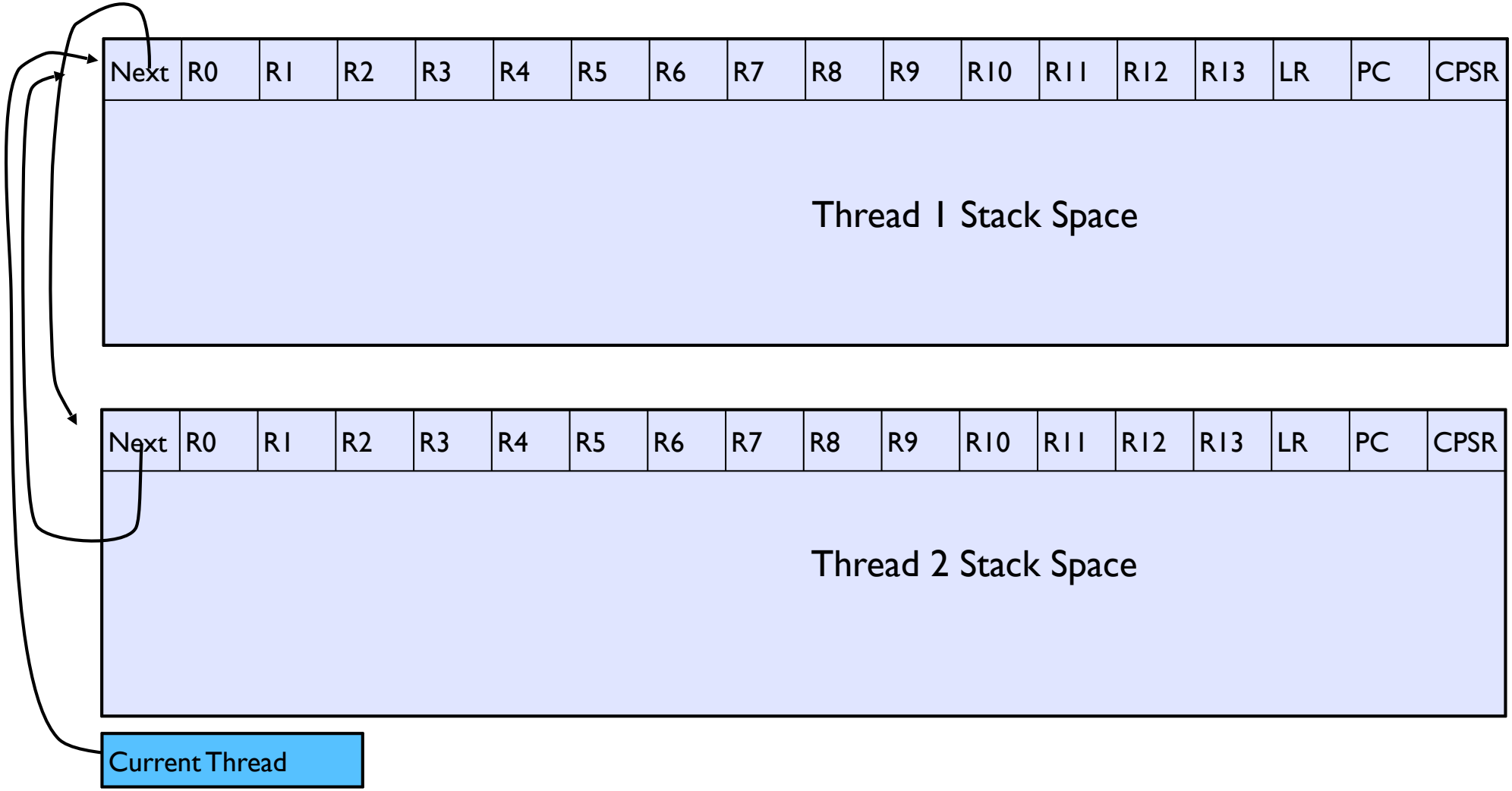
Preserving Context

- Peripherals: Can't really preserve the context of Peripherals or the Outside World
  - Solution: agree not to affect them while the program is not running. Or, be prevented from accessing them completely and have an Operating system control them instead.
- Memory: In general, too "expensive" to preserve memory.
  - Solution: agree not to interfere with a thread's memory when not in use. Or, be prevented from accessing memory assigned to other programs, using protection and/or memory management units (MMUs).
- Registers:
  - Must save and restore registers completely faithfully.

General Solution

- Give each thread its own (small amount of) private memory space.
  - This space can be used for two things:
    - The thread can have its own stack in that space
    - The scheduler can store the thread's register context in there.
  - The thread can use the rest of RAM on the understanding that other threads won't interfere with allocations.
- More generally, for example, a unix pthread is similarly given its own stack space
  - It continues to have access to the rest of memory (the 'heap') for malloc'ed memory.

Two-Thread Scheduler Scheme



Subroutine Standards

- Subroutines are frequently used to implement:
  - Functions
  - Methods
  - Procedures

    in high level languages like C, C++.
- There is an ARM standard for passing parameters and using registers.
  - This is a *software convention* developed by ARM Ltd. The hardware of the CPU does not require it.
  - It makes it possible for subroutines / functions written in different languages to interwork.

PCS Register Convention

Register	Synonym	Special	Role
r15		PC	Program Counter.
r14		LR	Link Register.
r13		SP	Stack Pointer.
r12		IP	Intra-Procedure-call scratch register.
r11	v8	FP	Frame Pointer or Variable-register 8.
r10	v7		Variable-register 7.
r9	v6	SB TR	Platform register. (“Role defined by the platform standard.”)
r8	v5		Variable-register 5.
r7	v4		Variable-register 4.
r6	v3		Variable-register 3.
r5	v2		Variable-register 2.
r4	v1		Variable-register 1.
r3	a4		Argument / scratch register 4.
r2	a3		Argument / scratch register 3.
r1	a2		Argument / result / scratch register 2.
r0	a1		Argument / result / scratch register 1.

ARM DOC ID IHI0042I_2019Q4, P18
Remember, it’s a software convention, not a hardware requirement.