An Introduction to Processor Design

Veena.G.S.
IV A

An Introduction to Processor Design

Some of the recent intel processors

- 3xx - Celeron D.
- 4xx - Celeron.
- 5xx - Pentium 4.
- 6xx - Pentium 4.
- 8xx - Pentium D and Pentium Extreme Edition.
- 9xx - Pentium D and Pentium Extreme Edition.
- E1xxx - Celeron Dual-Core.
- **Core i9**. A family of 64-bit x86 CPUs with up to 18 **cores** from **Intel**. ... Designed for high-performance computing and gaming, the 3.3 GHz **i9** chip can be overclocked to 4.5

9th Generation processor list: (some are yet to release) [2019 Intel release]

Intel Core i9 Q3 (2017-present)
Intel Core i9-9900KFC (14th Feb-2019)
Intel® 20XM Processor Extreme Edition (8M Intel® Core™ i9-7980XE Extreme Edition Processor (24.75M Cache, up to 4.20 GHz) i9-7980XE 1,933,533
Intel® Core™ i9-7960X X-series Processor (22M Cache, up to 4.20 GHz) i9-7960X 1,855,467
Intel® Core™ i9-7940X X-series Processor (19.25M Cache, up to 4.30 GHz) i9-7940X 1,803,167
Intel® Core™ i9-7920X X-series Processor (16.50M Cache, up to 4.30 GHz) i9-7920X 1,451,933
Intel® Core™ i9-7900X X-series Processor (13.75M Cache, up to 4.30 GHz) i9-7900X 1,384,900
Intel® Core™ i7-7820X X-series Processor (11M Cache, up to 4.30 GHz) i7-7820X 1,219,200
Intel® Core™ i7-7800X X-series Processor (8.25M Cache, up to 4.00 GHz) i7-7800X 901,833
Intel® Core™ i7-7740X X-series Processor (8M Cache, up to 4.50 GHz) i7-7740X 437,167
Intel® Core™ i5-7640X X-series Processor (6M Cache, up to 4.20 GHz) i5-7640X 406,667
Intel® Core™ i7-6950X Processor Extreme Edition (25M Cache, up to 3.50 GHz) i7-6950X 595,500
Intel® Core™ i7-6900K Processor (20M Cache, up to 3.70 GHz) i7-6900K 509,867
Intel® Core™ i7-6850K Processor (15M Cache, up to 3.80 GHz) i7-6850K 432,600
Intel® Core™ i7-6800K Processor (15M Cache, up to 3.60 GHz) i7-6800K 408,567
Intel® Core™ i7-5960X Processor Extreme Edition (20M Cache, up to 3.50 GHz) i7-5960X 478,000
Intel® Core™ i7-5930K Processor (15M Cache, up to 3.70 GHz) i7-5930K 420,583
Intel® Core™ i7-5820K Processor (15M Cache, up to 3.60 GHz) i7-5820K 396,550
Intel® Core™ i7-4960X Processor Extreme Edition (15M Cache, up to 4.00 GHz) i7-4960X 213,600
Intel® Core™ i7-4940MX Processor Extreme Edition (8M Cache, up to 4.00 GHz) i7-4940MX 251,100
Intel® Core™ i7-4930K Processor (12M Cache, up to 3.90 GHz) i7-4930K 201,733
Intel® Core™ i7-4930MX Processor Extreme Edition (8M Cache, up to 3.90 GHz) i7-4930MX 243,000
Intel® Core™ i7-4820K Processor (10M Cache, up to 3.90 GHz) i7-4820K 148,000
Intel® Core™ i7-3970X Processor Extreme Edition (15M Cache, up to 4.00 GHz) i7-3970X 207,667
Intel® Core™ i7-3960X Processor Extreme Edition (15M Cache, up to 3.90 GHz) i7-3960X 195,800
Intel® Core™ i7-3940XM Processor Extreme Edition (8M Cache, up to 3.90 GHz) i7-3940XM 120,000
Intel® Core™ i7-3930K Processor (12M Cache, up to 3.80 GHz) i7-3930K 189,867
Intel® Core™ i7-3920XM Processor Extreme Edition (8M Cache, up to 3.80 GHz) i7-3920XM 116,000
Intel® Core™ i7-3820 Processor (10M Cache, up to 3.80 GHz) i7-3820 144,00

9th Generation processor list: (some are yet to release) [2019 Intel release] based on their speed.

Intel Core	533 MHz, 667 MHz
Intel Core 2	533 MHz, 667 MHz, 800 MHz, 1066 MHz, 1333 MHz, 1600 MHz
Intel Core i3	1066 MHz, 1600 MHz, 2.5 – 5 GT/s
Intel Core i5	2.5 – 5 GT/s
Intel Core i7	4.8 GT/s, 6.4 GT/s(Giga transfers/second
Intel Core i7(Extreme Edition)	2.5GT/s – 6.4 GT/s
Intel Core i9	8 GT/s

Design of a general-purpose processor

- Design of a general-purpose processor, in common with most engineering endeavors, requires the careful consideration of many trade-offs and compromises

Abstraction is fundamental to understanding complex computers

Abstractions which are employed by computer hardware designers, of which the most important is the logic gate. The design of a simple processor is presented, from the instruction set, through a register transfer level description, down to logic gates.

Processor architecture and organization Advancedment/Progress

Spectacular increase in the performance of processors and an equally spectacular reduction in their cost.

- Most of the improvements have resulted from advances in the technology of electronics, moving from valves (vacuum tubes) to individual transistors, to integrated circuits (ICs) incorporating several bipolar transistors
- and then through generations of IC technology leading to today's very large scale integrated
- (VLSI) circuits delivering millions of field-effect transistors on a single chip.
- transistors get smaller they get cheaper, faster, and consume less power.
- All modern general-purpose computers employ the principles of the *stored-program digital computer*

SOC(Microcontrollers)

- A system-on-a-chip (SoC) is a microchip with all the necessary electronic circuits and parts for a given system, such as a smartphone or wearable computer, on a single integrated circuit (IC). SOCs are built for various applications
- An SoC for a sound-detecting device, for example, might include an audio receiver, an analog-to-digital converter (ADC), a microprocessor, memory, and the input/outputlogic control for a user - all on a single chip.

Processor architecture and organization

- **Computer architecture** describes the user's view of the computer.
- **The instruction set, visible registers, memory management table structures and exception handling model are all part of the architecture.**

Computer organization

Computer organization describes the user-invisible implementation of the architecture.

The pipeline structure, transparent cache, table-walking hardware and translation look-aside buffer are all aspects of the organization.

•

Computer organization

- **Computer organization describes the user-invisible implementation of the architecture**. **The pipeline structure, transparent cache, table-walking hardware (**A translation lookaside buffer (TLB) is a memory cache that is used to reduce the time taken to ... With **hardware** TLB management, the CPU automatically **walks** the page **tables** (using the CR3 register on x86, for instance) **and translation look-aside buffer** are all aspects of the organization.

A general-purpose processor

- A general-purpose processor is **a finite-state automaton that executes instructions held in a memory.**

- The state of the system is defined by the values held in the memory locations together with the values held in certain registers within the processor itself

- Each instruction defines a particular way the total state should change and it also defines which instruction should be executed next

stored-program digital computer.

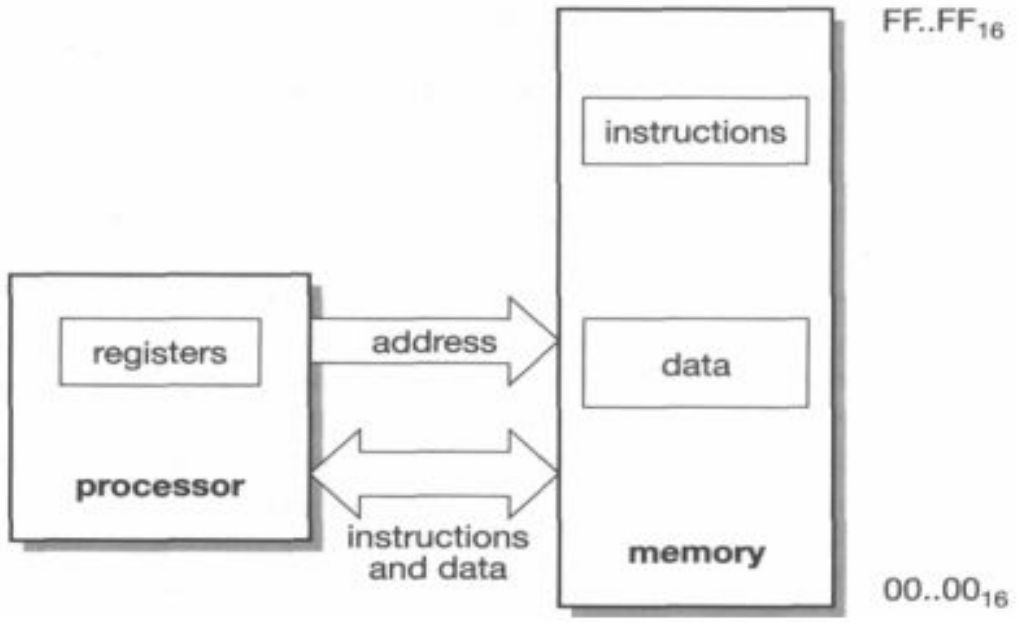


Figure 1.1 The state in a stored-program digital computer.

The **stored-program** digital computer

keeps its instructions and data in the same program memory system, allowing the instructions to be treated as data when necessary.

- This Computer enables the processor itself to generate instructions which it can subsequently execute.
- **Although programs that do this at a fine granularity (self-modifying code) are generally considered bad form these days since they are very difficult to debug**,
- use at a coarser granularity is fundamental to the way most computers operate. Whenever a computer loads in a new program from disk (overwriting an old program) and then executes it the computer is employing this ability to change its own program.

stored-program digital computer.

- In the stored program concept, both the instructions and the data (that the instructions operate on) **are stored in the computer memory itself. Before the introduction of this idea, instructions and data were considered two totally different entities and were thus stored separately in Harvard type of architecture**.
- **Thus instructions like data can be read from the memory and written to the memory by the processor.**
- **The processor then addresses the memory, reads the corresponding instructions, executes them and according to the executed instruction, processes (reads and writes) data as well.**

stored-program digital computer.

- **Computers that store both instructions and data on the same memory are said t**o **be based on the Von Neumann architecture.** Modern desktop computers are still based on the same stored program concept.

Need for Abstraction in hardware design

- Computers are very complex pieces of equipment that operate at very high speeds.
- A modern microprocessor makes use of Abstraction in hardware design be built from several million transistors each of which can switch a hundred million times a second.
- on a desktop PC or workstation and try to imagine how a hundred million transistor switching actions are used in each second of that movement.

Abstraction in hardware design

Abstractions which are employed by computer hardware designers, of which the most important is the logic gate.

The design of a simple processor is presented, from the instruction set, through a register transfer level description, down to logic gates

how a hundred million million transistor switching actions are used in each second of that movement the consequence of a deliberate design decision. None of them is random or uncontrolled

Logic gates abstracted from transistors The point about the gate abstraction is that not only does it greatly simplify the process of designing circuits with great numbers of transistors,

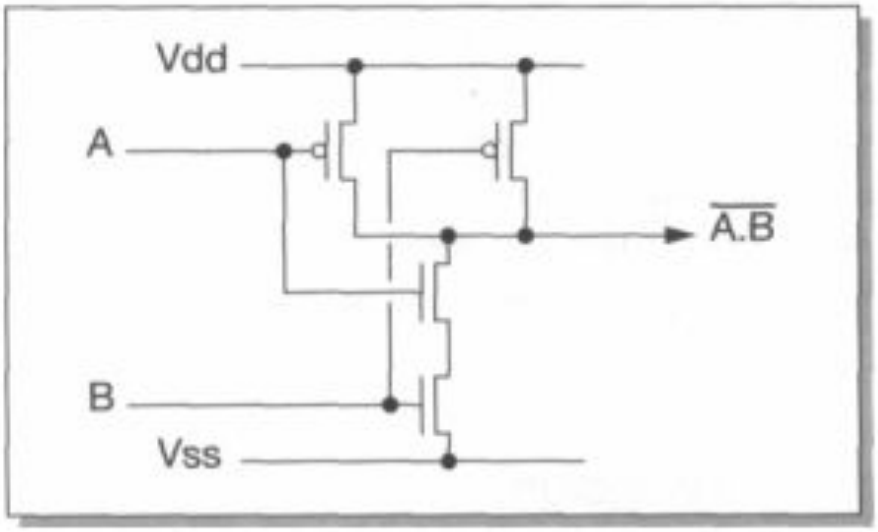


Figure 1.2 The transistor circuit of a static 2-input CMOS NAND gate.

Transistors abstracted as logic gates with symbols

Logic symbol

A	B	Output
0	0	1
0	1	1
1	0	1
1	1	0

Truth table

Abstraction in Transistor

- Yet the gross behaviour of a transistor can be described, without reference to quantum mechanics, as a set of equations that relate the voltages on its terminals to the current that flows though it.
- These equations abstract the essential behaviour of the device from its underlying physics.

Abstraction in logic gates

- **Logic gates** The equations that describe the behaviour of a transistor are still fairly complex.
- When a group of transistors is wired together in a particular structure, such as the CMOS (Complementary Metal Oxide Semiconductor) NAND gate shown in Figure 1.2, the behaviour of the group has a particularly simple description.
- If each of the input wires (A and B) is held at a voltage which is either near to Vdd or near to Vss, the output will will also be near to Vdd or Vss according to the following rules:
- • If A and B are both near to Vdd, the output will be near to Vss.
- • If either A or B (or both) is near to Vss, the output will be near to Vdd.

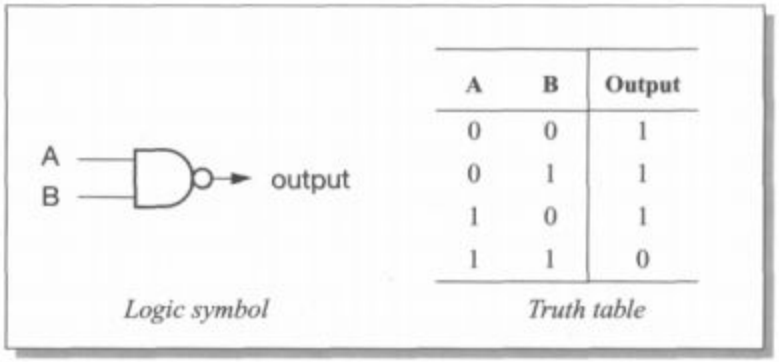
Abstraction in hardware design

- Logic symbol Truth table The gate abstraction Although there is a lot of engineering design involved in turning **four transistors** into a reliable **implementation of this equation**, it can be done with **sufficient reliability** that the logic designer can think almost exclusively in terms of logic gates.
- The concepts that the logic designer works with are illustrated in Figure 1.3, and consist of the following 'views' of the logic gate:

Abstraction in hardware design

- A logic symbol. This is a symbol that represents a NAND gate function in a circuit schematic; there are similar symbols for other logic gates (for instance, removing the bubble from the output leaves an AND gate which generates the opposite output function; further examples are given in 'Appendix: Computer Logic' on page 399).
- • A truth table. This describes the logic function of the gate, and encompasses everything that the logic designer needs to know about the gate for most purposes. The significance here is that it is a lot simpler than four sets of transistor equations. (In this truth table we have represented 'true' by '1' and 'false' by '0', as is common practice when dealing with Boolean variables.)

Abstraction in hardware design(The Gate Level Abstraction)



A	B	Output
0	0	1
0	1	1
1	0	1
1	1	0

Truth table

Figure 1.3 The logic symbol and truth table for a NAND gate.

Levels of abstraction Gate-level design

- **Levels of abstraction Gate-level design removes the need to know that the gate is built from transistors.**
- A logic circuit should have the same logical behaviour whether the gates are implemented using **field-effect transistors (the transistors that are available on a CMOS process), bipolar transistors,** electrical relays, fluid logic or any other form of logic.

Levels of abstraction Gate-level design

- **The implementation technology will affect the performance of the circuit, but it should have no effect on its function**. It is the duty of the transistor-level circuit designer to support the gate abstraction as near perfectly as is possible in order to isolate the logic circuit designer from the need to understand the transistor equations.
- It may appear that this point is being somewhat laboured, particularly to those readers who have worked with logic gates for many years.

Levels of abstraction Gate-level design

- However, the principle that is illustrated in the gate level abstraction is repeated many times at different levels in computer science and is absolutely fundamental to the process which we began considering at the start of this section, which is the management of complexity.

- The process of gathering together a few components at one level to extract their essential joint behaviour and hide all the unnecessary detail at the next level enables us to scale orders of complexity in a few steps.

A typical hierarchy of abstraction at the hardware level might be:

1. transistors;
2. logic gates, memory cells, special circuits;
3. single-bit adders, multiplexers, decoders, flip-flops;
4. word-wide adders, multiplexers, decoders, registers, buses;
5. ALUs (Arithmetic-Logic Units), barrel shifters, register banks, memory blocks;
6processor, cache and memory management organizations;
7. processors, peripheral cells, cache memories, memory management units;
8. integrated system chips;
9. printed circuit boards;
10. mobile telephones, PCs, engine controllers

Gate-level Design

- The next step up from the logic gate is to assemble a library of useful functions each composed of several gates.
- Typical functions are, as listed above, adders, multiplexers, decoders and flip-flops, each 1-bit

Gate Level Design

- Boolean algebra and notation;
- binary numbers;
- binary addition;
- multiplexers;
- clocks;
- sequential circuits;
- latches and flip-flops;
- registers.

MU0 - a simple processor

A simple form of processor can be built from a few basic components:

- **a program counter** (PC) register that is used to hold the address of the current instruction;
- a single register called an **accumulator** (ACC) that holds a data value while it is worked upon;
- **an arithmetic-logic unit** (ALU) that can perform a number of operations on binary operands, such as add, subtract, increment, and so on;
- **an instruction register** (IR) that holds the current instruction while it is executed;
- *instruction decode and control logic* that employs the above components to achieve the desired results from each instruct

MU0 - a simple processor

MU0 is a simple processor with limited set of components allows a restricted set of instructions to be implemented.

- Such a design has been employed at the University of Manchester for many years to illustrate the principles of processor design

- MU0 is a 16-bit machine with a 12-bit address space, so it can address up to 8Kbytes of memory arranged as 4,096 individually addressable 16-bit locations

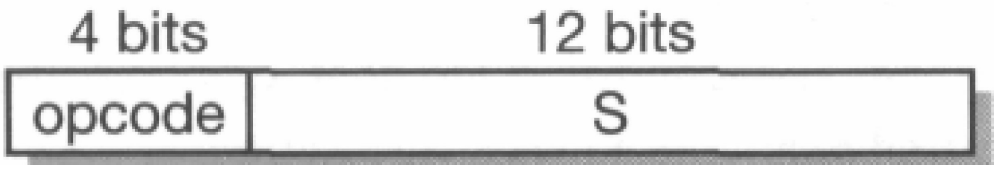
- Instructions are 16 bits long, with a 4-bit operation code (or **opcode)** and a 12-bitaddress field (S)

The MU0 instruction set

An instruction such as 'ACC := ACC + mem16[S]' means 'add the contents of the(16-bit wide) memory location whose address is S to the accumulator

Instruction	Opcode	Effect
LDA S	0000	ACC := mem_{16}[S]
STO S	0001	mem_{16}[S] := ACC
ADD S	0010	ACC := ACC + mem_{16}[S]
SUB S	0011	ACC := ACC – mem_{16}[S]
JMP S	0100	PC := S
JGE S	0101	if ACC ≥ 0 PC := S
JNE S	0110	if ACC ≠ 0 PC := S
STP	0111	stop

MU0 logic design



The MU0 instruction format.

The approach taken here will be to separate the design into two components

- **The datapath.**

All the components carrying, storing or processing many bits in parallel will beconsidered part of the datapath, **including the accumulator, program counter, ALU and instruction register.**

For these components we will use a register transfer level **(RTL)** design style based on registers, multiplexers, and so on.

- **The control logic.**

Everything that does not fit comfortably into the datapath will be considered part of the control logic and will be designed using a **finite state machine (FSM)** approach.

Data Path Design

- There are many ways to connect the basic components needed to implement the MU0 instruction set.
- Where there are choices to be made we need
  a guiding principle to help us make the right choices.
- Here we will follow the principle that the memory will be the limiting factor in our design, and a memory access will always take a clock cycle.
  Hence we will aim for an implementation where:
- **Each instruction takes exactly the number of clock cycles defined by the number of memory accesses it must make**

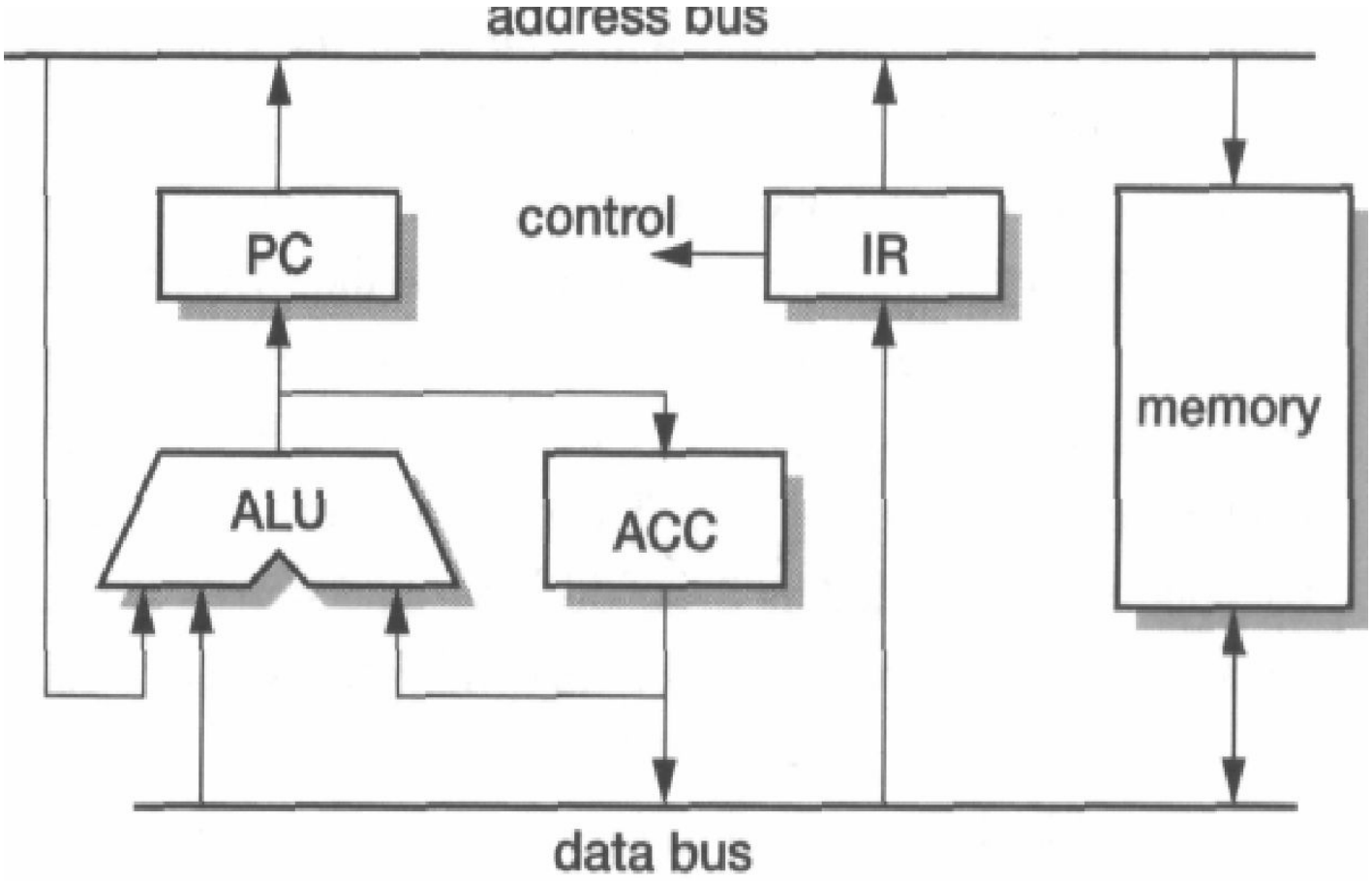
The MU0 instruction set

Instruction	Opcode	Effect
LDA S	0000	ACC := mem_{16}[S]
STO S	0001	mem_{16}[S] := ACC
ADD S	0010	ACC := ACC + mem_{16}[S]
SUB S	0011	ACC := ACC – mem_{16}[S]
JMP S	0100	PC := S
JGE S	0101	if ACC ≥ 0 PC := S
JNE S	0110	if ACC ≠ 0 PC := S
STP	0111	stop

The MU0 instruction set

- Referring back to Table 1.1 we can see that the first four instructions each require two memory accesses (one to fetch the instruction itself and one to fetch or store the operand)
- **whereas the last four instructions can execute in one cycle since they do not require an operand.**

Data path design



MU0 datapath example.

Datapath operation

- The design we will develop **assumes** that each instruction starts when it has arrived in the instruction register.
- Until the instruction is in instruction register we cannot know which instruction we are dealing with.
- **Address in** the instruction register is issued to fetch the next instruction, and in either case the **address is incremented** in the ALU and the **incremented value saved** into the PC.

Datapath operation

Instruction executes in **two stages**, possibly omitting the first of these:

1. **Access the memory operand and perform the desired operation**.

The address in the instruction register is issued and either an operand is read from memory, combined with the accumulator in the ALU and written back into the accumulator, or the accumulator is stored out to memory.

2**. Fetch the next instruction to be executed**.

Either the PC or the address in the instruction register is issued to fetch the next instruction, and in either case the address is incremented in the ALU and the incremented value saved into the PC.

Datapath Operation
Initialization

The processor must start in a known state.
Usually this requires a ***reset* input** to cause it to start executing instructions from a known address.
We will design MU0 to start executing
from address 000_{16}.
There are several ways to achieve this, one of which is to use the **reset signal** to zero the **ALU output** and then clock this into the PC register.

Register Trasfer Level Design

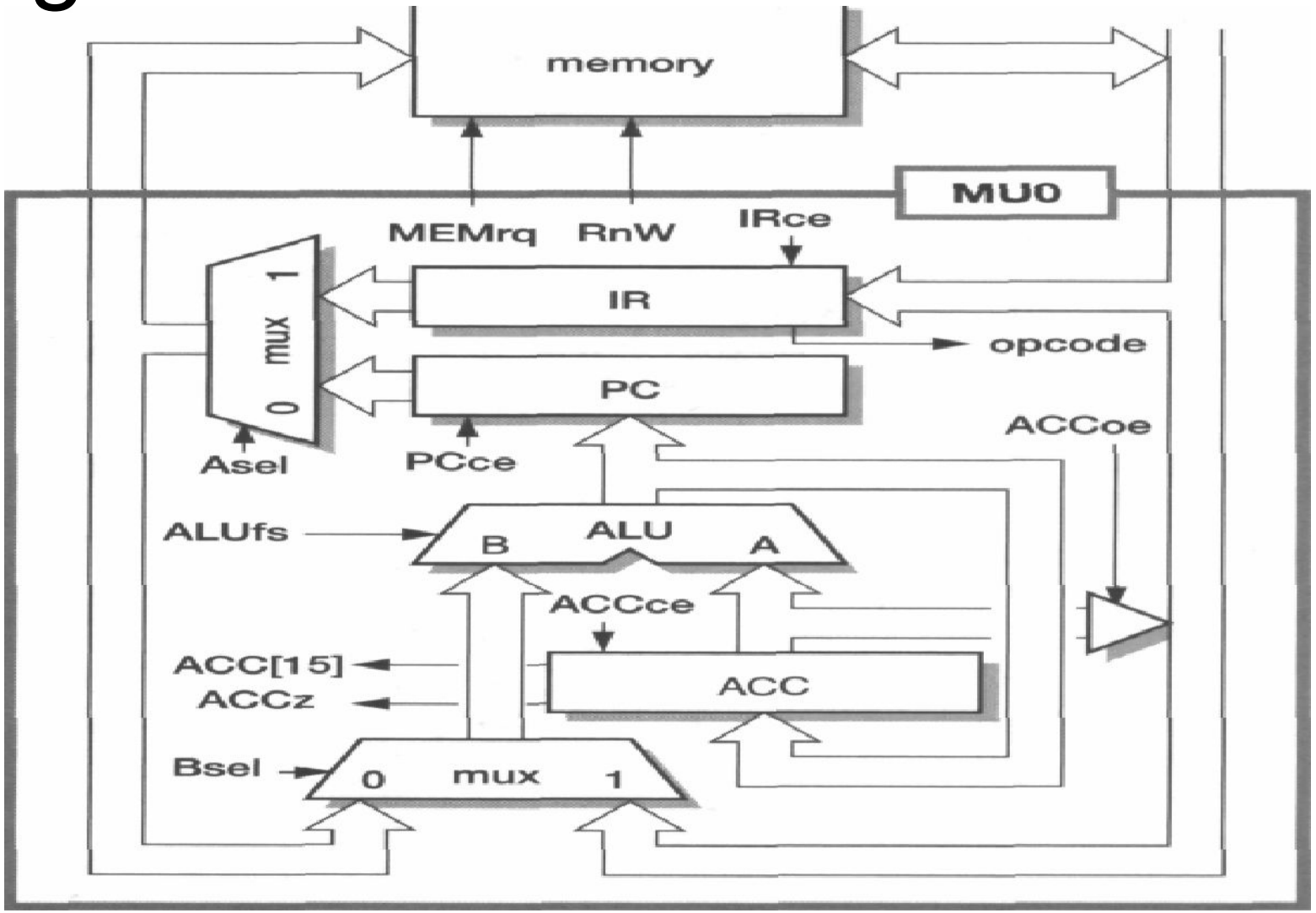
The next step is to determine exactly the control signals that are required to cause the datapath to carry out the full set of operations.
We assume that all the registers change state on the **falling edge of the input clock**, and where necessary have control signals that may be used to prevent them from changing on a particular clock
edge.
The PC, for example, will change at the end of a clock cycle where *PCce* is ' 1' but will not change when *PCce* is '0'.

Register Level Design

A suitable register organization is shown in Figure.
Controls signals are as follows

- This shows enables on all of the registers,
- function select lines to the ALU (the precise number and interpretation to be determined later),
- the select control lines **for two multiplexers**,
- the control for a tri-state driver to send the ACC value to memory and memory request *(MEMrq)* and read/write *(RnW)* control lines.
- The other signals shown are outputs from the datapath to the control logic, including the opcode bits and signals indicating whether ACC is zero or negative which control the respective conditional jump instructions.

MU0 register transfer level organization.



MU0 register transfer level organization.

- The program counter and **instruction register clock** enables *(PCce* and *IRce)* are always the same.
- This makes sense, since whenever a new instruction is being fetched the ALU is computing the next program counter value, and this should be latched too.
- Therefore these control signals may be merged into one.
- Similarly, whenever the accumulator is driving the data bus *(ACCoe* is high)
- the memory should perform a write operation *(RnW* is low), so one of these signals can be generated from the other using an inverter

Control Logic

- The control logic simply has to decode the current instruction
- generate the appropriate levels on the datapath control signals, using the control inputs from the datapath where necessary.
- The implementation requires only two states, **'fetch' and 'execute',** and one bit of state **(Ex/ft)** is therefore sufficient.

Control Logic

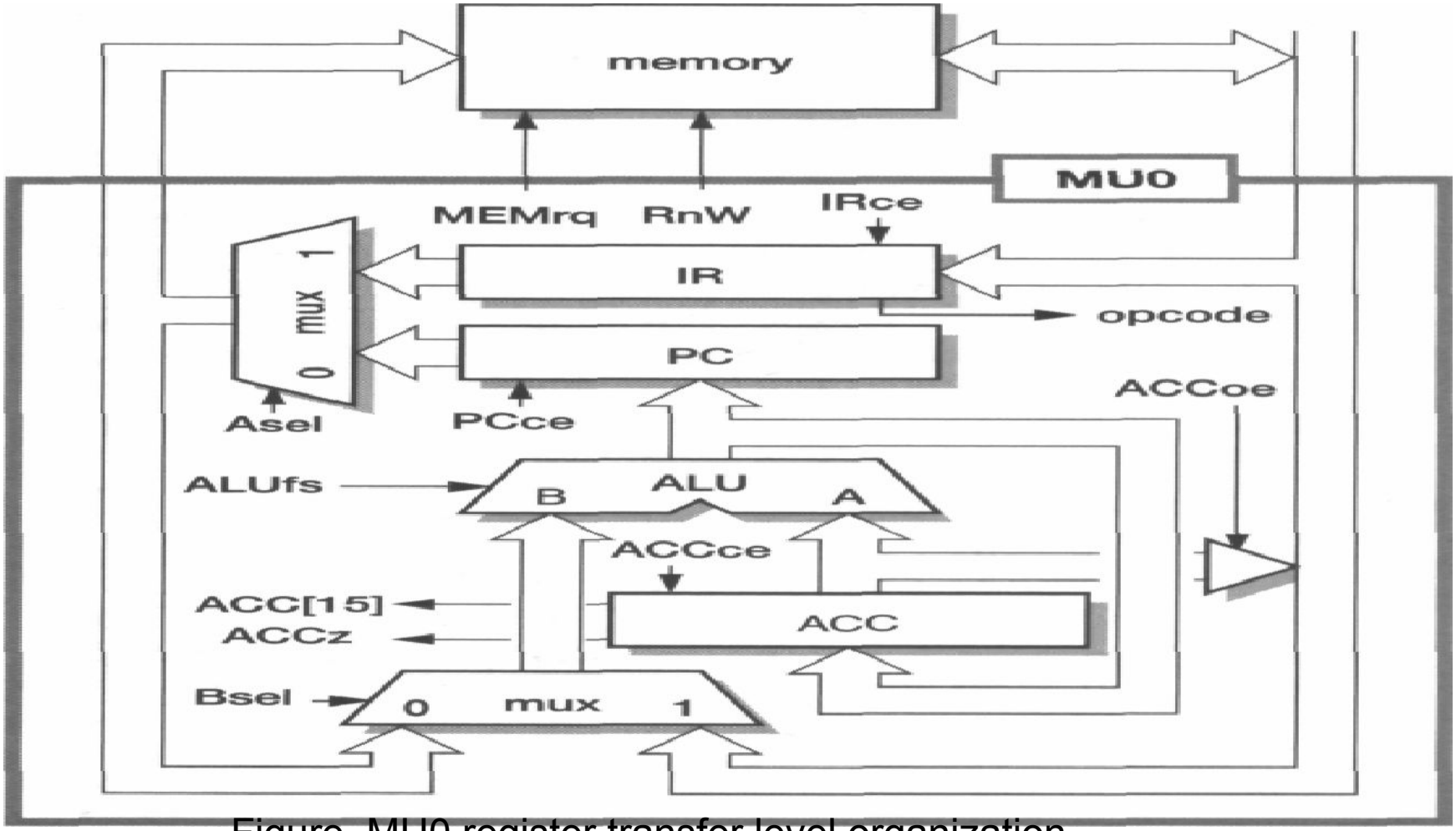


Figure MU0 register transfer level organization.

MU0 control logic.

Inputs						Outputs									
Instruction	Opcode	Reset	Ex/ft	ACCz	ACC15	Asel	Bsel	ACCce	PCce	IRce	ACCoe	ALUfs	MEMrq	RnW	Ex/ft
Reset	xxxx	1	x	x	x	0	0	1	1	1	0	=0	1	1	0
LDA S	0000	0	0	x	x	1	1	1	0	0	0	=B	1	1	1
	0000	0	1	x	x	0	0	0	1	1	0	B+1	1	1	0
STO S	0001	0	0	x	x	1	x	0	0	0	1	x	1	0	1
	0001	0	1	x	x	0	0	0	1	1	0	B+1	1	1	0
ADD S	0010	0	0	x	x	1	1	1	0	0	0	A+B	1	1	1
	0010	0	1	x	x	0	0	0	1	1	0	B+1	1	1	0
SUB S	0011	0	0	x	x	1	1	1	0	0	0	A–B	1	1	1
	0011	0	1	x	x	0	0	0	1	1	0	B+1	1	1	0
JMP S	0100	0	x	x	x	1	0	0	1	1	0	B+1	1	1	0
JGE S	0101	0	x	x	0	1	0	0	1	1	0	B+1	1	1	0
	0101	0	x	x	1	0	0	0	1	1	0	B+1	1	1	0
JNE S	0110	0	x	0	x	1	0	0	1	1	0	B+1	1	1	0
	0110	0	x	1	x	0	0	0	1	1	0	B+1	1	1	0
STP	0111	0	x	x	x	1	x	0	0	0	0	x	0	1	0

Control Logic

- A quick scrutiny of Table reveals a few easy simplifications.
- The program counter and instruction register clock enables ***(PCce* and *IRce)*** are always the same.
- This makes sense, since whenever a new instruction is being fetched the ALU is computing the next program counter value, and this should be latched too.
- Therefore these control signals may be merged into one.
- Similarly, whenever the accumulator is driving the data bus ***(ACCoe* is high**) the memory should perform a **write operation *(Rn W)* is low),** so one of these signals can be generated from the other using an inverter.

Control Logic

- The ALU functions that are required,There are five of them
- *(A + B, A — B, B, B* + 1,0), the last of which is only used while reset is active. Therefore
- the reset signal can control this function directly and the control logic need only
- generate a 2-bit function select code to choose between the other four.
- If the principal ALU inputs are the *A* and *B* operands, all the functions may be produced by augmenting a conventional binary adder:

Control Logic

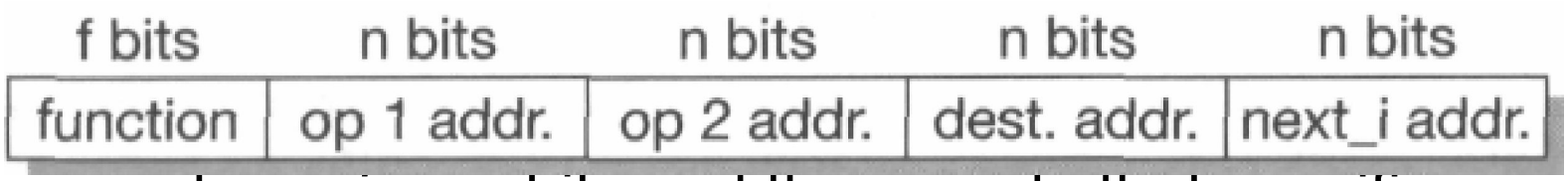
- *A* + *B* is the normal adder output (assuming that the carry-in is zero).
- *A* — *B* may be implemented as *A* + *B* + 1, requiring the *B* inputs to be inverted and the carry-in to be forced to a one.
- *B* is implemented by forcing the *A* inputs and the carry-in to zero.
- *B* + 1 is implemented by forcing *A* to zero and the carry-in to one.

MU0 control logic.

Inputs						Outputs									
Instruction	Opcode	Reset	Ex/ft	ACCz	ACC15	Asel	Bsel	ACCce	PCce	IRce	ACCoe	ALUfs	MEMrq	RnW	Ex/ft
Reset	xxxx	1	x	x	x	0	0	1	1	1	0	=0	1	1	0
LDA S	0000	0	0	x	x	1	1	1	0	0	0	=B	1	1	1
	0000	0	1	x	x	0	0	0	1	1	0	B+1	1	1	0
STO S	0001	0	0	x	x	1	x	0	0	0	1	x	1	0	1
	0001	0	1	x	x	0	0	0	1	1	0	B+1	1	1	0
ADD S	0010	0	0	x	x	1	1	1	0	0	0	A+B	1	1	1
	0010	0	1	x	x	0	0	0	1	1	0	B+1	1	1	0
SUB S	0011	0	0	x	x	1	1	1	0	0	0	A–B	1	1	1
	0011	0	1	x	x	0	0	0	1	1	0	B+1	1	1	0
JMP S	0100	0	x	x	x	1	0	0	1	1	0	B+1	1	1	0
JGE S	0101	0	x	x	0	1	0	0	1	1	0	B+1	1	1	0
	0101	0	x	x	1	0	0	0	1	1	0	B+1	1	1	0
JNE S	0110	0	x	0	x	1	0	0	1	1	0	B+1	1	1	0
	0110	0	x	1	x	0	0	0	1	1	0	B+1	1	1	0
STP	0111	0	x	x	x	1	x	0	0	0	0	x	0	1	0

Instruction set design

- 4-address format:This format requires *4n +f* bits per instruction where each



- operand *requires n* bits and the opcode that specifies 'ADD' requires/fbits.
- instruction requires some bits to differentiate it from instructions other instructions, some bits to specify the operand addresses, some bits to specify where the result should be placed (the **destination),** and some bits to specify the address of the next instruction to be executed

3-address format

- 3-addressThe first way to reduce the number of bits required for each instruction is to make instructions the address of the next instruction implicit (except for branch instructions, whose role is to modify the instruction sequence explicitly).

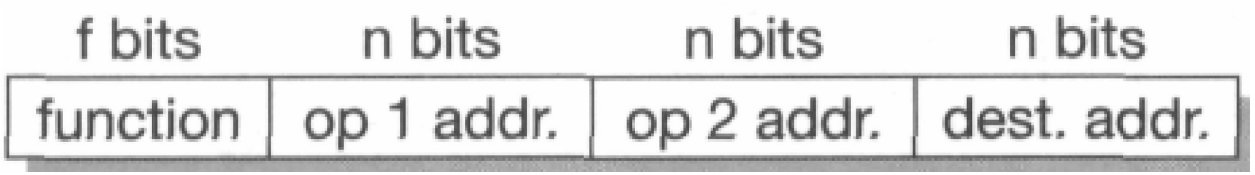
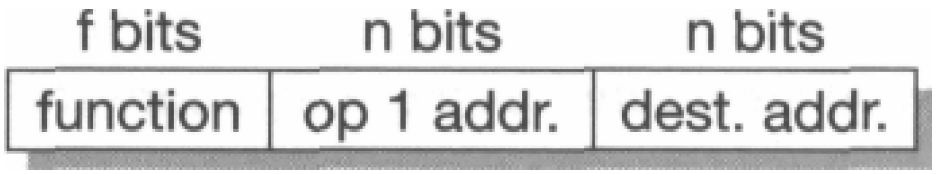


Figure A 3-address instruction format.

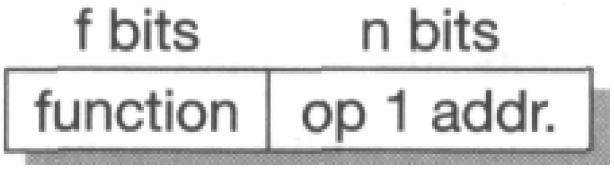
2-Address format

- 2-address: A further saving in the number of bits required to store an instruction can be instructions achieved by making the destination register the same as one of the source registers.



1-Address Instructions

- 1-address instructions
- If the destination register is made implicit it is often called the **accumulator** (see,
- for example, MU0 in the previous section); an instruction need only specify one operand:
- ADD s1 ; accumulator := accumulator + s1



0-address Instructions

- 0-address instructions
- Finally, an architecture may make all operand references implicit by using an evaluation stack.
- The assembly language format is:

ADD ; top_of_stack := top_of_stack +

next_on_stack

Instruction types

- A general-purpose instruction set can be expected to include instructions in the following categories:
- Data processing instructions such as add, subtract and multiply.
- Data movement instructions that copy data from one place in memory to another, or from memory to the processor's registers, and so on.
- Control flow instructions that switch execution from one part of the program to another, possibly depending on data values.
- Special instructions to control the processor's execution state, for instance to switch into a privileged mode to carry out an operating system function.

Orthogonal Instructions

- An instruction set is said to be **orthogonal** if each choice in the building of an instruction is independent of the other choices.
- orthogonal instruction set (architecture):An **instruction set** where all (or most)instructions have the same format an d all registers and addressing modes can be used interchangeably - the choices of **op code**, register, and addressing mode are mutually independent (looselyspeaking, the choices are "**orthogonal**").
- This contrasts with some early **Intel** microprocessors where only certain registers could be used by certain instructions.

Orthogonal Instructions

- an **orthogonal instruction** set is an **instruction** set architecture where all **instruction** types can use all addressing modes. It is "**orthogonal**" in the sense that the **instruction**type and the addressing mode vary independently.Orthogonality in practice · The PDP-11 · The VAX-11 · The MC68000

Addressing modes

1. Immediate addressing: the desired value is presented as a binary value in the instruction.

MOV AL, 35H (move the data 35H into AL register)

2. Absolute addressing: the instruction contains the full binary address of the desired value in memory.

LDA R0,=0x20000000

3. . Indirect addressing: the instruction contains the binary address of a memory location that contains the binary address of the desired value.

```
int temp = *x;
```

Here temp is assigned the value of int type stored at the address contained in X

Addressing Modes

4. Register addressing: the desired value is in a register, and the instruction contains the register number.

```
MOV AX,CX (move the contents of CX register to AX register)
```

5. Register indirect addressing: the instruction contains the number of a register which contains the address of the value in memory.

```
MOV AX, [BX](move the contents of memory location s
addressed by the register BX to the register AX)
```

6. Base plus offset addressing: the instruction specifies a register (the **base)** and a binary offset to be added to the base to form the memory address.

```
MOV AL,[BX+05]
```

ADD AX, [BX+SI]

Addressing modes

7. Base plus index addressing: the instruction specifies a base register and another register (the **index)** which is added to the base to form the memory address.

```
ADD AX, [BX+SI]
```

8. Base plus scaled index addressing: as above, but the index is multiplied by a constant (usually the size of the data item, and usually a power of two) before being

added to the base.

Base + (Index * Scale)

9. Stack addressing: an implicit or specified register (the **stack pointer)** points to an area of memory (the **stack)** where data items are written **(pushed)** or read **(popped)** on a last-in-first-out basis.

Control flow instructions

- Where the program must deviate from the default (normally sequential) instruction sequence, a control flow instruction is used to modify the program counter (PC) explicitly.
- The simplest such instructions are usually called 'branches' or 'jumps'. Since most branches require a relatively short range, a common form is the 'PC-relative' branch.
- A typical assembly language format is:

```
        B
        ..
LABEL   ..
```

Continued..

- Here the assembler works out the displacement which must be added to the value the PC has when the branch is executed in order to force the PC to point to LABEL.
- Conditional branches:Some processors (including MU0) allow the values in the general registers to control whether or not a branch is taken through instructions such as:
- Branch if a particular register is zero (or not zero, or negative, and so on).
- Branch if two specified registers are equal (or not equal).

Subroutine calls

- Sometimes a branch is executed to call a subprogram where the instruction sequence should **return to the calling sequence when the subprogram terminates.**
- Since the subprogram may be called from many different places, a record of the calling address must be kept.
- There are many different ways to achieve this:
- The calling routine could compute a suitable return address and put it in a standard memory location for use by the subprogram as a return address before executing the branch.
- The return address could be pushed onto a stack.
- The return address could be placed in a register.

System Calls

- Another category of control flow instruction is the system call. This is a branch to
- an operating system routine, often associated with a change in the privilege level of the executing program.
- Some functions in the processor, possibly including all the input and output peripherals, are protected from access by user code.
- Therefore a user program that needs to access these functions must make a system call.
- System calls pass through protection barriers in a controlled way.
- A well-designed processor will ensure that it is possible to write a multi-user operating system where one user's program is protected from assaults from other, possibly malicious, users.
- This requires that a malicious user cannot change the system code and, when access to protected functions is required, the system code must make thorough checks that the requested function is authorized.

Exceptions

- The final category of control flow instruction comprises cases where the change in the flow of control is not the primary intent of the programmer but is a consequence of some unexpected (and possibly unwanted) side-effect of the program.
- An attempt to access a memory location may fail, for instance, because a fault is detected in the memory subsystem.
- The program must therefore deviate from its planned course in order to attempt to recover from the problem.
- Try and Catch blocks helps with the solution

Processor design trade-offs

- The art of processor design is to define an instruction set that supports the functions that are useful to the programmer whilst allowing an implementation that is as efficient as possible.
- Preferably, the same instruction set should also allow future, more sophisticated implementations to be equally efficient.

Processor design trade-offs

- The programmer generally wants to express his or her program in as abstract a way as possible, using a high-level language which supports ways of handling concepts that are appropriate to the problem.
- Modern trends towards functional and object-oriented languages move the level of abstraction higher than older imperative languages such as C, and even the older languages were quite a long way removed from typical machine instructions.
- The **semantic gap** between a high-level language construct and a machine instruction is bridged by a **compiler,** which is a (usually complex) computer program that translates a high-level language program into a sequence of machine instructions.
- Therefore the processor designer should define his or her instruction set to be a good compiler target rather than something that the programmer will use directly to solve the problem by hand.

What processors do

- If we want to make a processor go fast, we must first understand what it spends its time doing.
- It is a common misconception that computers spend their time computing, that is, carrying out arithmetic operations on user data.
- In practice they spend very little time 'computing' in this sense.
- Although they do a fair amount of arithmetic, most of this is with addresses in order to locate the relevant data items and program routines.
- Then, having found the user's data, most of the work is in moving it around rather than processing it in any transformational sense.

What processors do

Instruction type	Dynamic usage
Data movement	43%
Control flow	23%
Arithmetic operations	15%
Comparisons	13%
Logical operations	5%
Other	1%

Complex Instruction Set Computers

- A complex **instruction set** computer (**CISC** /ˈsɪsk/) is a computer in which single **instructions** can execute several low-level operations (such as a load from memory, an arithmetic operation, and a memory store) or are capable of multi-step operations or addressing modes within single **instructions**

RISC Organisation

- RISC uses Hard-wired instruction decode logic;
- CISC processors used large microcode
- ROMs to decode their instructions.
- Pipelined execution;
- CISC processors allowed little, if any, overlap between consecutive instructions (though they do now).
- Single-cycle execution; CISC processors typically took many clock cycles to complete a single instruction.

RISC advantages

- A smaller die size.
- A simple processor should require fewer transistors and less silicon area.
- Therefore a whole CPU will fit on a chip at an earlier stage in process technology development, and once the technology has developed beyond the point where either CPU will fit on a chip, a RISC CPU leaves more die area free for performance-enhancing features such as cache memory, memory management functions, floating-point hardware, and so on.

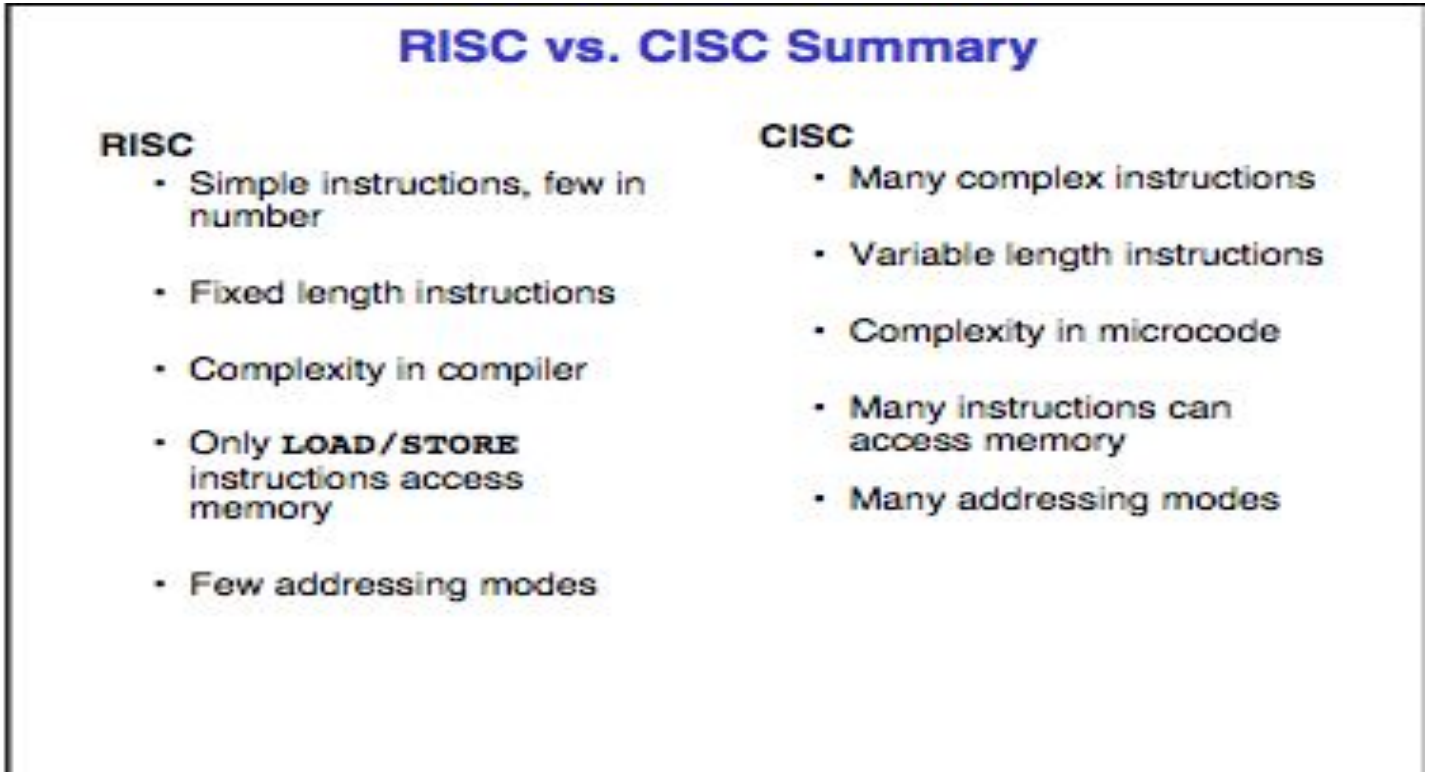
A shorter development time.

- A simple processor should take less design effort and therefore have a lower design cost and be better matched to the process technology when it is launched (since process technology developments need be predicted over a shorter development period).

The Reduced Instruction Set Computer
RISC architecture

- RISC processors had a fixed (32-bit) instruction size with few formats;
- CISC processors typically had variable length instruction sets with many formats.
- RISC architecture was a load-store architecture where instructions that process data operate only on registers and are separate from instructions that access memory;
- CISC processors typically allowed values in memory to be used as operands in data processing instructions.
- In RISC a large register bank of thirty-two 32-bit registers, all of which could be used for any purpose, to allow the load-store architecture to operate efficiently;
- CISC register sets were getting larger, but none was this large and most had different registers for different purposes (for example, the *data and address registers on the* Motorola MC68000).
- These differences greatly simplified the design of the processor and allowed the designers to implement the architecture using organizational features that contributed to the performance of the prototype devices

RISC VS CISC


RISC vs. CISC Summary
RISC
Simple instructions, few in number
Fixed length instructions
Complexity in compiler
Only LOAD/STORE instructions access memory
Few addressing modes
CISC
Many complex instructions
Variable length instructions
Complexity in microcode
Many instructions can access memory
Many addressing modes

RISC VS CISC

Example for RISC vs. CISC

Consider the the program fragments:

CISC

```
mov ax, 10
mov bx, 5
mul bx, ax
```

RISC

```
        mov ax, 0
        mov bx, 10
        mov cx, 5
Begin   add ax, bx
        loop Begin
```

The total clock cycles for the CISC version might be:

(2 movs × 1 cycle) + (1 mul × 30 cycles) = 32 cycles

While the clock cycles for the RISC version is:

(3 movs × 1 cycle) + (5 adds × 1 cycle) + (5 loops × 1 cycle) = 13 cycles

RISC advantages

- A higher performance.
- This is the tricky one! The previous two advantages are easy to accept, but in a world where higher performance had been sought through ever-increasing complexity.

RISC Drawbacks

- With the passage of time, two drawbacks have come to light:
- RISCs generally have poor code density compared with CISCs.
- **code density** - **Definition**. The amount of space that an executable program takes up in memory.
- **Code density** is important in mobile devices that contain a limited amount of memory.
- RISCs don't execute x86 code.

Pipelines

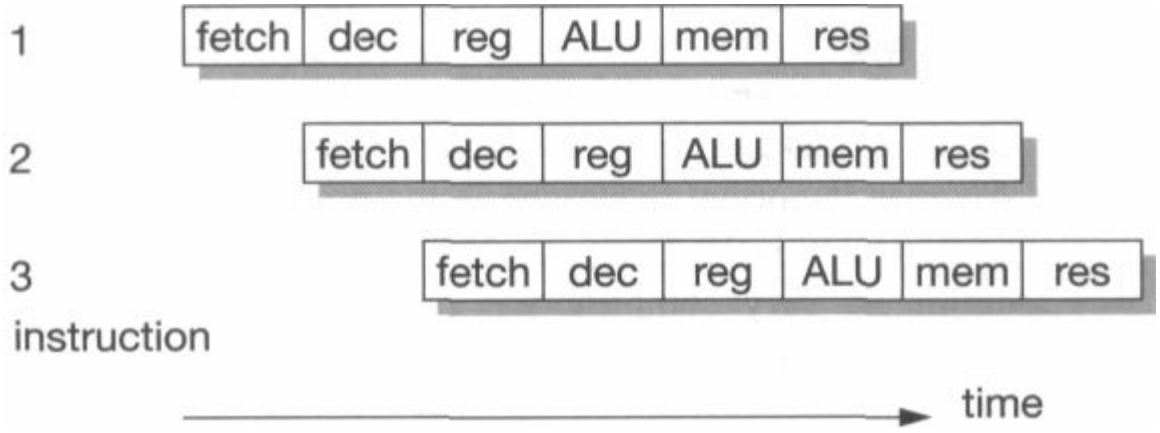
- A processor executes an individual instruction in a sequence of steps. A typical sequence might be:
- 1. Fetch the instruction from memory (fetch).
- 2. Decode it to see what sort of instruction it is (dec).
- 3. Access any operands that may be required from the register bank (reg).
- 4. Combine the operands to form the result or a memory address (ALU).
- 5. Access memory for a data operand, if necessary (mem).
- 6. Write the result back to the register bank (res).

RISC in retrospect

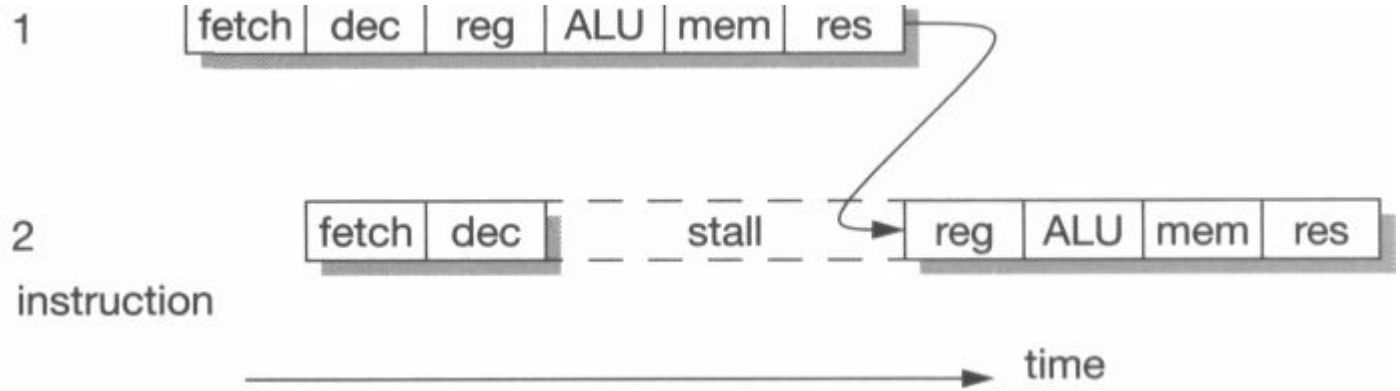
- RISCs achieved their performance through:
- **Pipelining.**
- Pipelining is the simplest form of concurrency to implement in a processor and
- delivers around two to three times speed-up. A simple instruction set greatly simplifies
- the design of the pipeline.
- **A high clock rate** with single-cycle execution.

Pipeline hazards

- It is relatively frequent in typical computer programs that the result from one instruction is used as an operand by the next instruction.
- When this occurs the pipeline operation shown in Figure 1.13 breaks down, since the result of instruction 1 is not available at the time that instruction 2 collects its operands.
- Instruction 2 must therefore stall until the result is available, giving the behaviour shown in Figure 1.14
- This is a **read-after-write pipeline hazard.**



Pipeline hazards

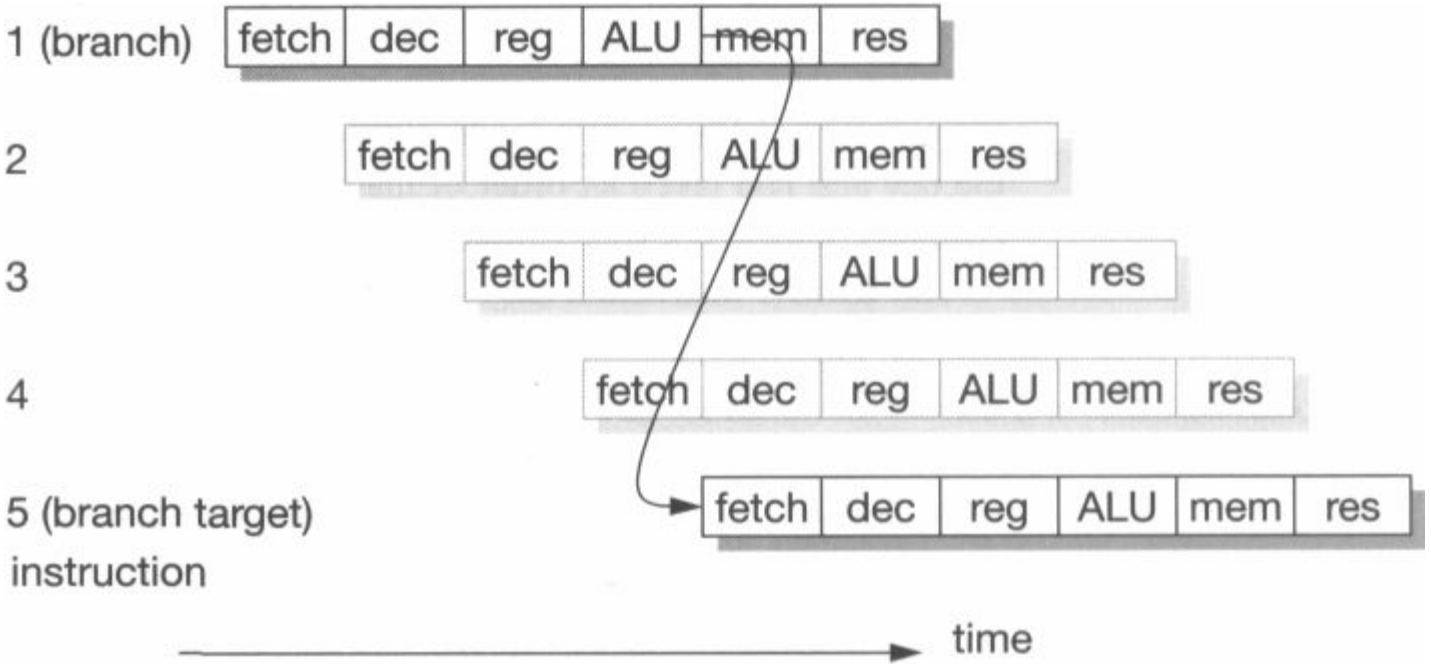


Read-after-write pipeline hazard.

Pipeline hazards(Branch Instructions)

- Branch instructions result in even worse pipeline behaviour since the fetch step of the following instruction is affected by the branch target computation and must therefore be deferred.
- Unfortunately, subsequent fetches will be taking place while the branch is being decoded and before it has been recognized as a branch, so the fetched instructions may have to be discarded

Pipeline hazards(Branch Instructions)



Pipelined branch behaviour.

Operations in ALU

- There are five of them
- *(A* + *B, A* — *B, B, B* + 1,0), the last of which is only used while reset is active.
- Therefore the reset signal can control this function directly and the control logic need only generate a 2-bit function select code to choose between the other four.
- If the principal ALU inputs are the *A* and *B* operands, all the functions may be produced by augmenting
- a conventional binary adder:

• *A* + *B* is the normal adder output (assuming that the carry-in is zero).
• *A* — *B* may be implemented as *A* + *B* + 1, requiring the *B* inputs to be inverted and
the carry-in to be forced to a one.
• *B* is implemented by forcing the *A* inputs and the carry-in to zero.
• *B* + 1 is implemented by forcing *A* to zero and the carry-in to one.

MU0 register transfer level organization

Inputs						Outputs									
	Opcode		Ex/ft	ACC15			Bsel		PCce	ACCoe		MEMrq		Ex/ft	
Instruction		Reset		ACCz		Asel		ACCce		IRce		ALUfs		RnW	
Reset	xxxx	1	x	x	x	0	0	1	1	1	0	= 0	1	1	0
LDA S	0000	0	0	x	x	1	1	1	0	0	0	= B	1	1	1
	0000	0	1	x	x	0	0	0	1	1	0	B+1	1	1	0
STO S	0001	0	0	x	x	1	x	0	0	0	1	x	1	0	1
	0001	0	1	x	x	0	0	0	1	1	0	B+1	1	1	0
ADD S	0010	0	0	x	x	1	1	1	0	0	0	A+B	1	1	1
	0010	0	1	x	x	0	0	0	1	1	0	B+1	1	1	0
SUB S	0011	0	0	x	x	1	1	1	0	0	0	A–B	1	1	1
	0011	0	1	x	x	0	0	0	1	1	0	B+1	1	1	0
JMP S	0100	0	x	x	x	1	0	0	1	1	0	B+1	1	1	0
JGE S	0101	0	x	x	0	1	0	0	1	1	0	B+1	1	1	0
	0101	0	x	x	1	0	0	0	1	1	0	B+1	1	1	0
JNE S	0110	0	x	0	x	1	0	0	1	1	0	B+1	1	1	0
	0110	0	x	1	x	0	0	0	1	1	0	B+1	1	1	0
STP	0111	0	x	x	x	1	x	0	0	0	0	x	0	1	0