Lecture 2

The CPU, Instruction Fetch & Execute

In Lecture 1 we learnt that the separation of data from control helped simplify the definition and design of sequential circuits, particularly when there were many registers involved only with storage.

We saw that the main memory, as well as holding plain data, could hold the program data — the ordered list of instructions that specify what you want to machine to do.

We speculated that when an instruction is read from memory, it could be passed immediately to the control part of the machine in order to change the effective transfer function of the data section.

In this lecture we develop the detailed organization of the CPU to support that idea.

Rather than introduce individual components separately and then stick them together, we will dive in at the deep end by revealing a "Bog Standard Architecture" for the CPU.

2.1 A Bog Standard Architecture

The CPU contains

- a number of registers, some of which fall on the address side, others on the data side;
- an arithmetic logic unit;
- the control section or control unit;
- connections to the memory (a large unit of storage) by two buses, the uni-directional address bus and the bi-directional data bus; and
- internal buses or data pathways which allow the output of one register to connect to the input of another.

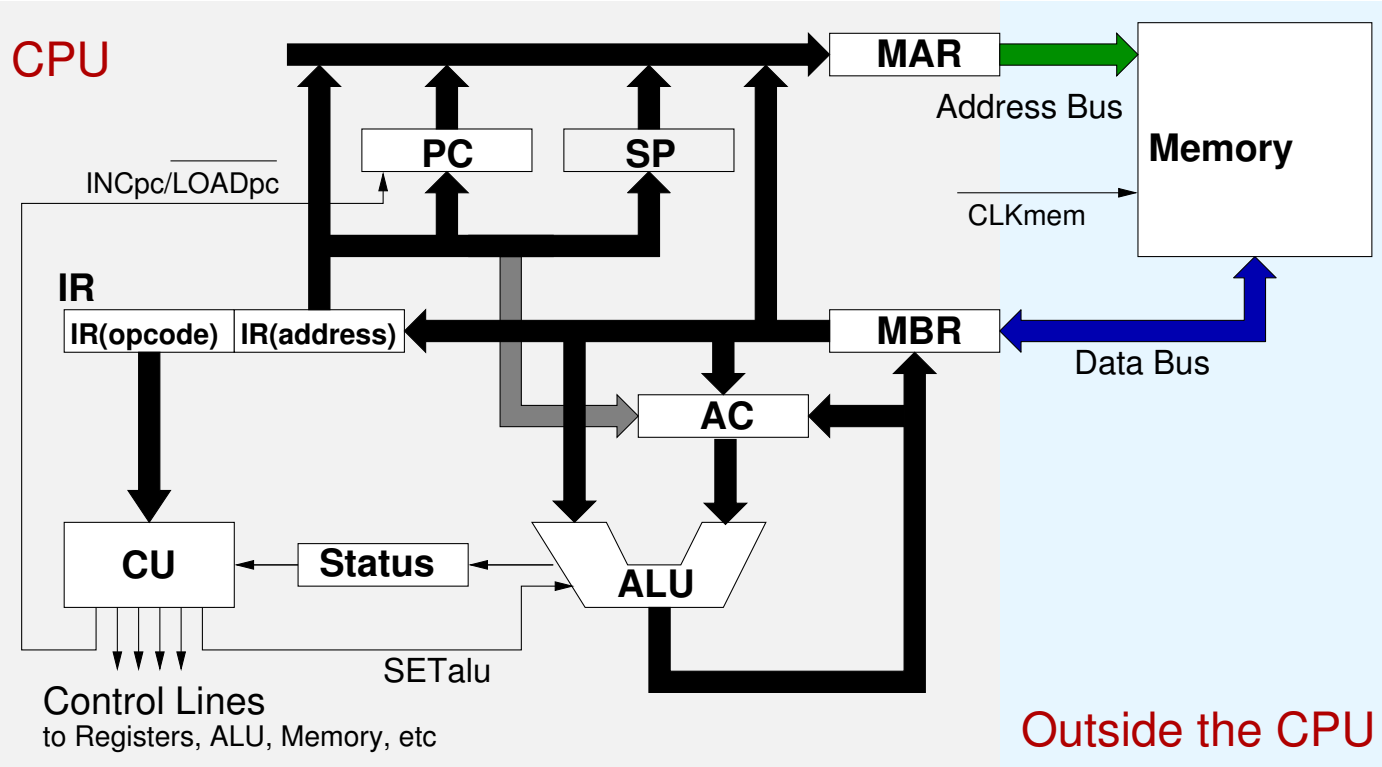


Figure 2.1: Our Bog Standard Architecture

2.1.1 CPU Registers

☆ **MAR** The **Memory Address Register** is used to store the address to access memory.

☆ **MBR** The **Memory Buffer Register** stores information that is being sent to, or received from, the memory along the bidirectional data bus.

☆ **AC** The **Accumulator** is used to store data that is being worked on by the ALU, and is the key register in the data section of the cpu. Notice that the memory can't access the AC directly. The MBR is an intermediary.

☆ **PC** Connected to the internal address bus, the **Program Counter** holds the address in memory of the next program instruction. Notice that it doesn't connect directly to the memory, but must go via the the MAR. (The PC is special in that it is both a register and a counter. More later.)

☆ **IR** When memory is read, the data first goes to the MBR. If the data is an instruction it gets moved to the **Instruction Register**. The IR has two parts:

1. **IR** (opcode) The most significant bits of the instruction make up the opcode. This the genuine *instruction* part of the instruction, that tells the cpu what to

do. The instruction in **IR** (opcode) gets decoded and executed by the control unit, CU.

2. **IR** (address) The least significant bits of the instruction are actually *data*. They get moved to **IR** (address). As the name suggests they usually form all or part of an address for later use in the MAR. (However, in immediate addressing they are sent to the AC.)

☆ **SP** The **Stack Pointer** is connected to the internal address bus and is used to hold the address of a special chunk of main memory used for temporary storage during program execution.

☆ **All Registers** are edge-triggered D-types — we will use falling-edge-triggered devices.

For all their fancy names, the registers comprise nothing more than a row of D-type latches which share a common clock input providing temporary storage on the CPU. In our design (and I hope I've been consistent) they are falling edge-triggered (hence the circle on the clock input). Because these registers output onto buses they have tri-state buffers are connected to a single input OE, for "Output Enable", as shown in Fig. 2.2

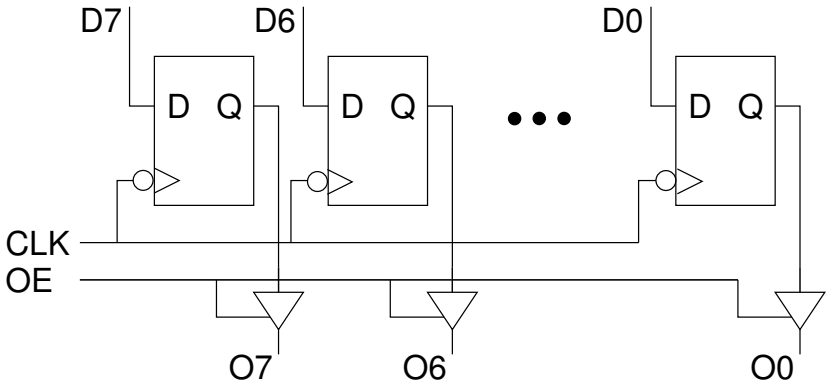


Figure 2.2: An 8-bit register with 3-state output enable.

2.1.2 Units in the CPU

☆ **CU** The **Control Unit** is responsible for the timing and execution of the various register transfers required to fulfill an instruction held in the IR. It has a number of control lines coming out of it, which transmit CSL and CSP levels and pulses to the various registers and logic units.

We shall develop its hardware as a one-hot sequencer later.

☆ **ALU** The **Arithmetic Logic Unit** is responsible for bit operations on data held in the **AC** and **MBR** and for storing the results. It contains arithmetic adders, logical AND-ers and OR-ers, and so on.

A special requirement in our architecture is a "null operation" or "no-op" which simply allows the output of the AC to appear at the output of the ALU.

Again we will detail its hardware later.

☆ **SR** Closely associated with the ALU is the **Status Register** or Condition Control Word or Status Word. It is not quite the same as the other registers in that it really just a collection of 1bit flags that indicate the outcome of operations that the ALU has just carried out. There are the flags (you met in P2) Carry C, Overflow V flags, negative flag N, and zero flag Z. These are monitored by the CU.

2.1.3 Buses, registers, and their widths

The buses carry *words* of information which are many bits wide, and on diagrams a bus is indicated either by a wide line, or by a single line with a dash through it often accompanied by the bus width in bits.

Data: Microcontrollers have data bus widths of 4 bits, 8-bits, 16-bits and 32-bits, while the most advanced PCs use 64 bits. In these lectures we will assume that the "memory width" is 16 bits or 2 Bytes. This means that each location can store 2 Bytes. We will also assume that the data bus is 16 bits wide, and the MBR and AC registers on the data side of the CPU are therefore also 16 bits wide. The ALU is also 16 bits wide.

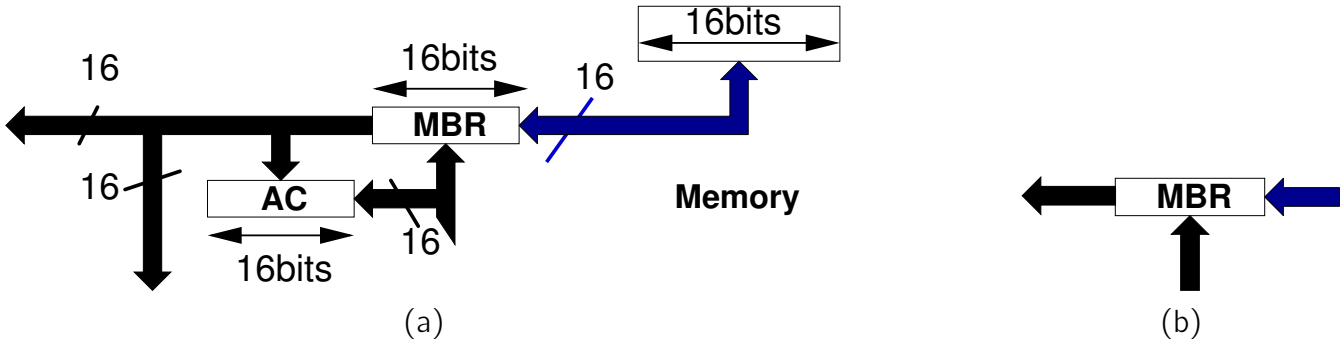


Figure 2.3: (a) The data side is 2 Bytes or 16 bits wide. The ALU has been omitted here, but is also 16 bits wide. (b) You should not think that the **MBR** register (for example) has grown multiple electrical inputs. The actual wiring involves tri-state buffers, as becomes clear in Lecture 3.

Address: The address bus does not have to be the same width as the data bus. The width on CPUs over time has increased in step with contemporary memory technology,

with the the Intel 8086 (from 1979) having $n = 20$ address lines to current processors having $n = 36 - 40$.

Having n address lines means that that there are 2^n addresses or locations in the address space. A convenient method of figuring out 2^n is to remember that 2^{10} = 1024, so $n = 10$ lines address 1K locations, $n = 20$ lines address 1M locations, and $n = 30$ can address 1G locations. Of course microcontrollers tend to have a smaller amount of memory, because they are not designed to multitask (i.e., run multiple programs), and 256K locations is the largest number spotted (in 2010).

However, for lecturing purposes it is useful (i) to have different numbers on the address and data side, and (ii) to keep things in multiple of 8 — so **here we will assume a 24 bit address bus**, able to access 16M location. (Note this is not necessarily 16MByte of memory. Why not?) The PC, SP, and MAR in our cpu will therefore be 24 bits wide.

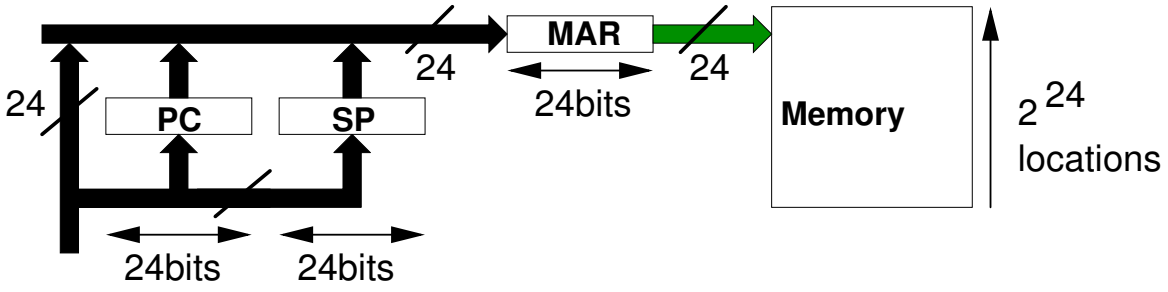


Figure 2.4: The address side is 24 bits or 3 Bytes wide. The address space has 2^{24} locations.

The **IR** is special. The **IR** (opcode) part should be wide enough to take the largest opcode. We assume the opcode is a fixed 8 bits wide, allowing 256 different instructions — which is plenty enough. The **IR** (address) part has to have the same width as the address bus, 24 bits. So the whole **IR** is 32 bits wide.

It is however fed from the internal data bus which is only 16 bits wide in our architecture. We will return to solve this conundrum in §2.6.1.

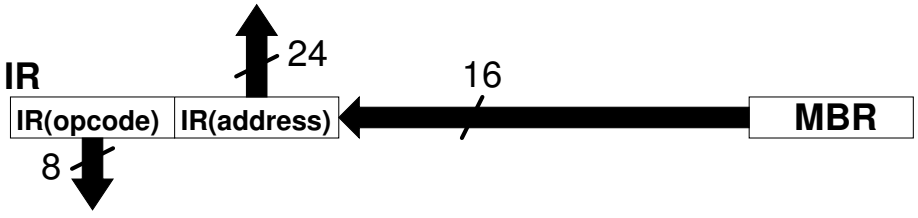


Figure 2.5: The IR must be $8 + 24 = 32$ bits width.

2.1.4 Introduction to the Main memory

The memory will comprise mostly random access memory (RAM) with some additional read-only memory (ROM) to help the machine start up. The main memory does not reside in the cpu chip but sits on the motherboard and is connected to the cpu via a bus — in fact two (or three) buses, the data bus and address bus (and also a control bus, which carries timing pulses and the level to indicate writing or reading).

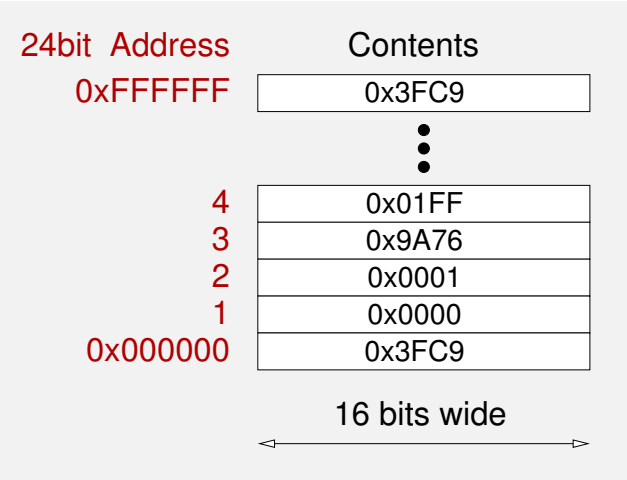


Figure 2.6:

The address bus has been chosen to be 24 bits wide, so the address space is from 0x0 to $2^{24} - 1$ or 0xFFFFFF in hex. The data bus is 16 bits wide, and so the contents width is 16 bits. In Fig.2.6, for example, the contents of address 4 are 0x01FF. The largest (unsigned) integer number that can be held is $2^{16} - 1$ or 0xFFFF.

Memory hardware is considered in more detail later in Lecture 3. For now, it is enough to know how to describe reading from and writing to the memory. The main memory is effectively just a large stack of registers, each with its own address. To read or write from memory the register transfers are written as

MBR$\leftarrow\langle$**MAR** $\rangle$	read from memory
$\langle$**MAR** $\rangle \leftarrow$**MBR**	write to memory

$\langle$**MAR** $\rangle$ means the memory register whose address is given by the **MAR**. The **MAR** is said to *point* to the memory location.

2.2 The Fetch, Decode, Execute Cycle

With the structure of registers, units, memory and buses laid out, let us be clear that the overall operational aim is very simple.

We want our CPU repeatedly to

- FETCH the next instruction from memory into the instruction register
- DECODE the instruction (that is, work out which it is)
- EXECUTE the instruction

Fetching and an Executing an instruction simply require the CPU's Control Section to issue Levels and Pulses which set up pathways and fire register transfers so that

- Data is moved from memory to registers, and between registers
- Data is passed (sometimes) through the ALU, and
- Data is stuffed back into the memory

If you are in need of an analogy, we are doing little more than "playing trains" with data. The Control Secton uses Levels to "set the points" and create the route between A and B, and uses a Pulse to send the train from A to B.

2.3 Fetching and decoding an instruction

To start processing the cpu needs to fetch the first instruction in the program from the main memory. The Program Counter is the key register here. The **PC***always* holds the address of the next program instruction in the main memory. It is said to *point* to the next instruction[1]. But remember that the memory address register acts as a gatekeeper to the memory, so the first thing to happen is that the program counter gets copied into the memory address register. The register transfer is

MAR$\leftarrow$**PC**

Because it is the **MAR** that is clocked, this leaves the **PC** unaltered. Now read the memory into the **MBR**.

MBR$\leftarrow\langle$**MAR** $\rangle$

The next step is to copy the instruction from the **MBR** to the instruction register.

IR$\leftarrow$**MBR**

In our standard architecture the **IR** is split into two parts, **IR** (opcode) and **IR** (address). As far as the instruction fetch is concerned it the **IR** (opcode) that is important. The opcode is decoded by the control unit, as described later.

Last comes a touch of housekeeping. Usually the next instruction in the program is located in the next memory location, so the program counter is incremented.

PC$\leftarrow$**PC** + 1

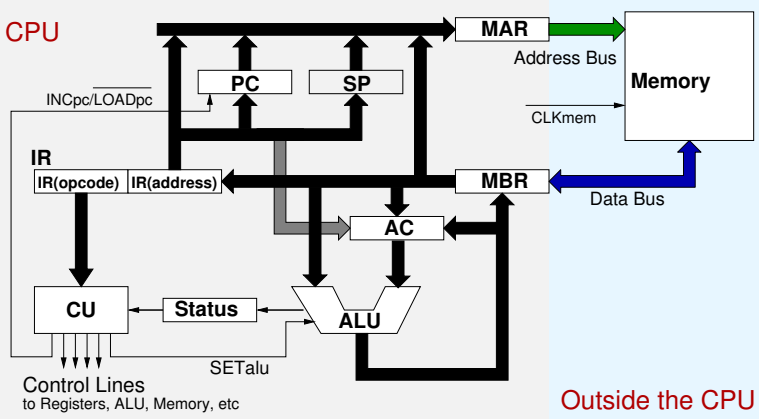
[1] Any memory address points to the memory contents at that address.

So to summarize, the instruction fetch requires the following in RTL, where you should note that the program counter can be incremented at the same clock tick as loading the instruction register.

Instruction fetch
1. **MAR**←**PC**
2. **MBR**←⟨**MAR** ⟩
3. **IR**←**MBR**; **PC**←**PC** + 1

(Then decode the opcode)

NB: these line numbers will soon turn into RTL Control Steps!



2.4 A few instructions

Our CPU uses 8-bit opcodes, so could distinguish 256 different instructions. For the purpose of explanation we give just nine from our *instruction set*. Column 1 contains the **assembler language** mnemonic, which is shorthand for several lines of RTL. Column 2 gives an overall "RTL-like" description. Column 3 is the binary **opcode**.

Inst	Overall RT	Opcode	Meaning
HALT		00000000	Stop the clock
LDA x	**AC**←⟨x⟩	00000001	Load AC with contents of mem address x
STA x	⟨x⟩ ←**AC**	00000010	Store AC in memory at address x
ADD x	**AC**←**AC** + ⟨x⟩	00000011	Add mem contents at x to AC
AND x	**AC** =**AC** ∧ ⟨x⟩	00000100	Logical and ...
JMP x	**PC**←x	00000101	Jump to instruction at address x
BZ x	if Z=1 then **PC**←x	00000110	if Z-flag is set then jump
NOT	**AC**←$\overline{\textbf{AC}}$	00000111	Two's complement the AC
SHR	**AC**←RightShift(**AC**)	00001000	Shift the AC 1bit to right

An assembler language is designed around a particular cpu, and there is no standard set of mnemonics. However, once you understand the purpose of the instructions, it is trivial to convert between languages.

2.5 Executing an instruction

During the instruction fetch, an opcode is put into the **IR** (opcode), and is decoded by the control unit (exactly how we will see later). The CU now "knows" which instruction it should execute, and can therefore output a sequence of of Levels and Pulses to set up paths and effect the desired the register transfers.

Let's suppose the opcode was 00000001. From the table one sees that the opcode's mnemonic is LDA, and that its action is to copy the contents of memory at address x into the accumulator.

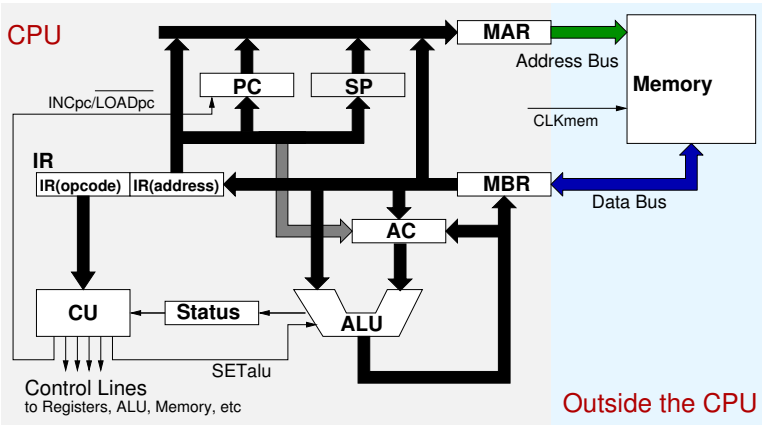
Inst	Overall RT	Opcode	Meaning
LDA x	**AC**←⟨x⟩	00000001	Load AC with contents of mem address x

What is x? It is the **operand**, a useful piece of data that is bound to the instruction. At its simplest in our BSA, the operand comprises the remaining 8 bits of the 16 bit instruction after removing the 8 bit opcode. These are the 8 bits in **IR** (address). Thus in six of our model instructions, the instruction contains an actual instruction to the control unit (the opcode), together with a piece of data (the operand) which the opcode can utilize.

2.5.1 Execution phase of LDA x

The overall effect of LDA x is **AC**←⟨x⟩. This means transfer the contents of the memory at location x to the **AC**. Looking at the BSA's diagram we see that in detail this must be:

LDA x
10. **MAR**←**IR** (address)
11. **MBR**←⟨**MAR** ⟩
12. **AC**←**MBR**; →(ℓ_{fetch})



Notes:

1. No computation has be been done by this instruction. Information has been moved around from memory to accumulator, but not altered. The only part of the cpu that can alter data is the ALU.

2. 10,11,12 will later become RTL Control Step numbers.

3. ℓ_{fetch} is used to denote the RTL step number of the start of fetch. Earlier we place the instruction fetch at RTL line 1. Hence →(ℓ_{fetch}) could be written as →(1).

2.5.2 Example of LDA x fetch and execute.

Figure 2.7 gives an example of the information flows in the entire fetch and execute of LDA x. In the example, the instruction is found in memory location 2, and the actual instruction is LDA 5.

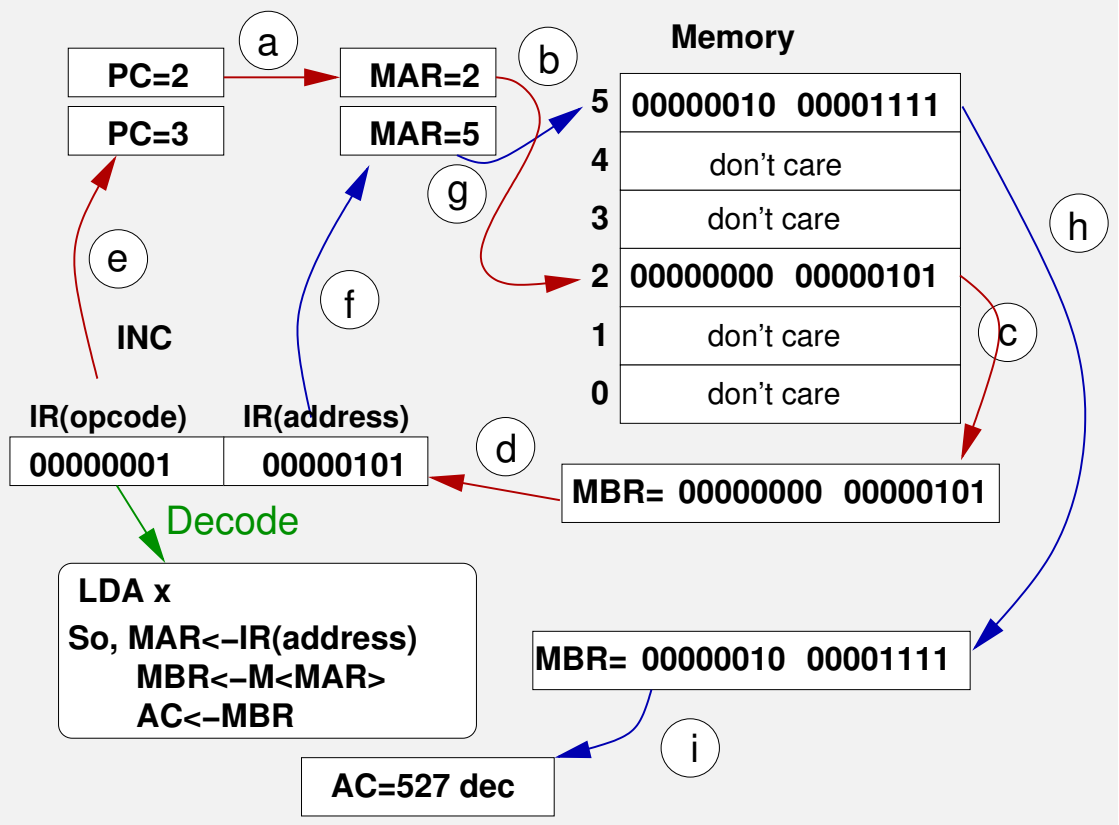


Figure 2.7: Example of the Fetch and execute of the LDA x instruction with x=5 and starting with **PC** =2. The steps follow a,b,c, ...,i.

(a) During the fetch, **MAR**←**PC**

(b) Addressing location 2

(c) Reading the memory **MBR**←⟨**MAR** ⟩

(d) Now the MBR is transferred to the IR.

(e) The last part of the fetch is to increment the PC.

(f) Decode, then first step of execute is **MAR**←**IR** (operand)

(g) Now addressing location 5.

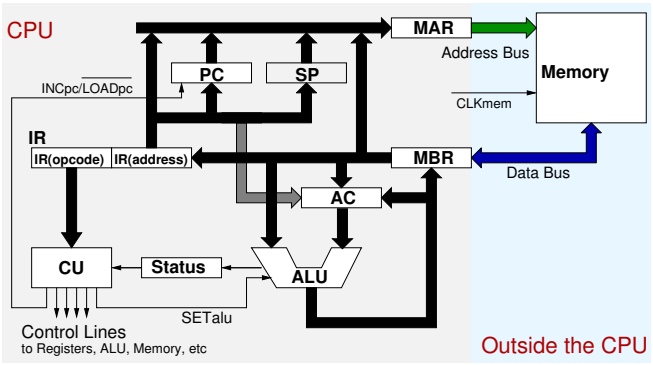
(h) Reading the memory **MBR**←⟨**MAR** ⟩ again.

(i) Now transfer to the Accumulator **AC**←**MBR**.

2.5.3 Execute STA x

Store the contents of the **AC** in the memory location x which is held in the **IR** (operand).

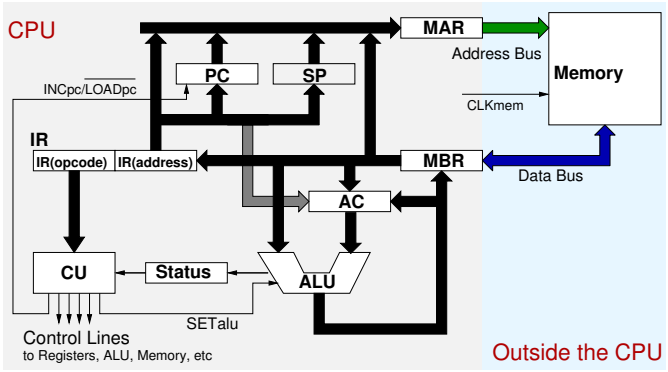
STA x
13. **MAR**←**IR** (address); **MBR**←**AC**
14. ⟨**MAR** ⟩ ←**MBR**; →(ℓ_{fetch})



2.5.4 Execute ADD x

Add the memory contents of x to the AC. This involves the ALU. Looking back at the BSA diagram we see that the ALU has two inputs one from the **AC** and the other from the **MBR**. To perform this line of RTL we must, before firing the transfer, set up the ALU to perform the binary addition of its inputs, rather than say the logical AND or logical OR. As you might guess, and as we will see in Lecture 4, this requires the configuration of combinational circuitry using a *level* signal. Obviously we are going to need a CSL in addition to a CSP for this line of RTL.

ADD x
15. **MAR**←**IR** (address)
16. **MBR**←⟨**MAR** ⟩
17. **AC**←**AC** + **MBR**; →(ℓ_{fetch})



2.5.5 AND x

This again involves the ALU, but now we are performing a logical AND.

AND x
18. **MAR**←**IR** (address)
19. **MBR**←⟨**MAR** ⟩
20. **AC**←**AC** ∧ **MBR**; →(ℓ_{fetch})

2.5.6 JMP x

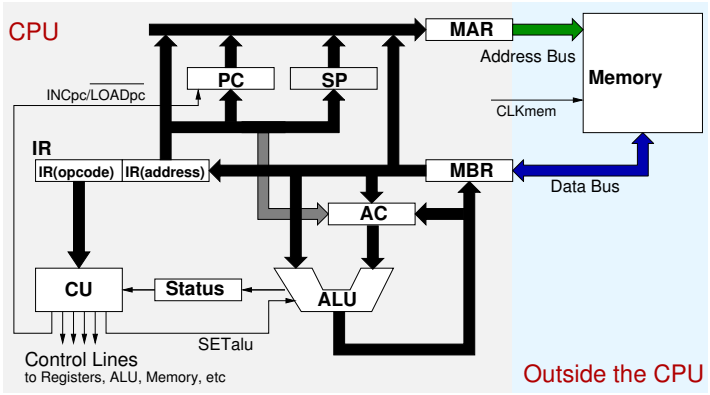
Branch unconditionally to new location for next instruction. The PC is always incremented during the fetch cycle on the assumption that the next instruction is in the next memory location. This instruction allows an *unconditional branching* to a non-consecutive instruction.

JMP x
21. **PC**←**IR** (address); →(ℓ_{fetch})

2.5.7 BZ x

This mnemonic is short for "branch if equal". To enable *conditional branching* (that is, "if (condition is true) then branch") we need a logical condition to test for true or false. The condition is taken as one of the bits from the status register. Here we use the Z bit or Z flag, which is set Z=1 by the ALU when it makes a calculation whose result is zero, and Z=0 when it isn't. Note that because the PC is always incremented during the instruction fetch, if the condition is false (ie Z=0) the PC needs no alteration.

BZ x
22. →($\overline{\mathbf{Z}}$)/(ℓ_{fetch})
23. **PC**←**IR** (address); →(ℓ_{fetch})



2.5.8 NOT and RSH

NOT complements (inverts) the contents of the **AC**. An ALU operation, so again we need a level to configure the ALU. RSH involves use of a shifter, often placed at the back end of the ALU.

NOT
24. **AC**←$\overline{\mathbf{AC}}$; →(ℓ_{fetch})

SHR
25. **AC**←RightShift(**AC**); →(ℓ_{fetch})

2.6 Decoding the opcode

Earlier on, when discussing the Instruction Fetch, we wrote "then decode the opcode". Although it is common to talk of the fetch-execute cycle, it is useful to think about decoding as a separate distinct phase in a fetch-decode-execute cycle.

Now suppose that

- we have a mechanism of producing a set of level signals [LDA=1, STA=0, ADD=0, etc] when the opcode is LDA; and similarly for other opcodes; and
- we have written the RTL for all the execute phases, and know that the execute phases start at control steos 10,13,15,18, and so on;

then we could write a decoding step using RTL's conditional goto (see lecture 1):

Decoding (NB! this is RTL)
4. →(LDA,STA,ADD,AND, ..., SHR,HALT)/(10,13,15,18,...,25,99)

where 10, etc, are the first lines of the RTL which execute the respective instruction.

2.6.1 Decoding detail — engineering to the rescue

So far, so good. However, in our BSA, the **MBR** and data bus are 16 bits wide. Hence, **IR** ←**MBR** can only supply the 8 bit opcode with an 8 bit operand. If operands are only 8 bits long, we can only access 256 of our 2^{24} locations. How can we fill the operand up to its full 24 bits?

The engineered solution is to make the LDA instruction (and any other instruction that needs a full operand) to read the next 16-bit word of memory into the **MBR**, and then into the **IR** (address), building up a 24 bit address for transferral to the **MAR**.

In our instruction set the first 6 opcodes LDA, STA, ADD, AND, JMP, BZ require this extra read of memory. We could write the decoding stage in RTL as follows:

Longer Decoding
4. →(NOT,SHR,...)/(24,25,...) //Ie, all that don't need extra
5. **MAR**←**PC**
6. **MBR**←⟨**MAR** ⟩
7. **IR** [23:8] ←**MBR**; **PC**←**PC** + 1
8. →(LDA,STA,...)/(10,13,...) //Ie, all that do need extra

This solution effectively divides up the opcodes into two sets: those that need full operands and those that don't.

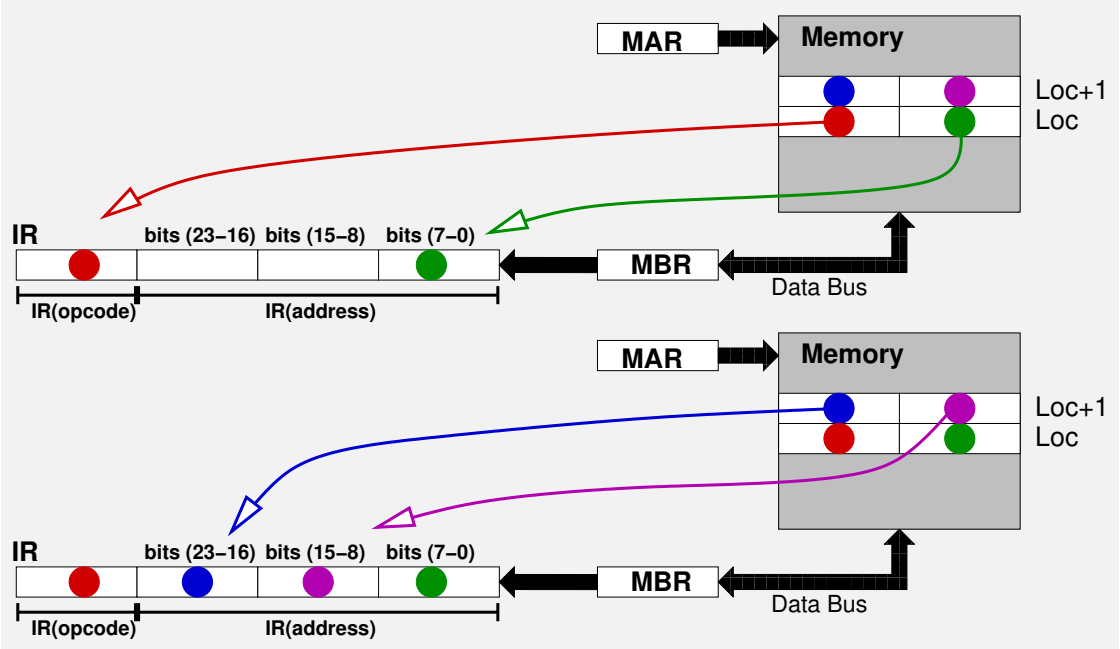


Figure 2.8: The fetch delivers the opcode and the low Byte of the operand. After decoding, if the opcode requires a full operand with 3 Bytes another read of the memory can take place.

The detail of this step is often missed out in text books, and it is assumed that the fetch (lines 1-3) provides an operand of full length. We too will neglect the problem — unless explicitly asked to worry about it!

(If we want to worry about it, there are changes of detail required to the RTL of line 3 of the instruction fetch. What are they, and once you've written the RTL could you draw the hardware involved?)

2.7 Summary

In this lecture we have

- laid out an architecture for a simple CPU, introduced its components, and described how the CPU connects to memory.
- noted that the data section of our CPU divides into two halves, one more concerned with addresses, the other with data, with the instruction register at the junction
- described using RTL the fetch of an instruction from memory into the instruction register, and learned that the instruction is made up of opcode and operand
- described, again using RTL, the execute phase of several common instructions
- discussed "simple" decoding, and how it may be made more elaborate to overcome constraints imposed by the architecture.

In addition we have realized that

- the Control Unit is responsible for delivering Levels and Pulses involved in the various control steps. The Levels must (among other things) establish paths between registers, and Pulses must perform the transfer.

In the next lecture we will

- first work on the design of the control section.
- Then we will design the ALU for the data section, and
- thirdly think more about the main memory.

2.8 Postscript

The architecture of our CPU is inevitably rudimentary, but it may come as a surprise when you look in textbooks to see that there are other equally rudimentary processors that appear rather different.

A common alternative is for authors not to differentiate so strongly between data and address when considering the CPU's internal buses, with all registers sharing them equivalently. I find it helps to make a strong distinction while learning the ropes.

Whatever the architecture, the underlying principles really are the same, and as you become comfortable with the operation of our architecture, you will be able to understand how others function and appreciate the differences.