Lecture 3

Control Unit, ALU, and Memory

In this lecture we develop models for the control unit, ALU and memory. It is the first that will require most care.

3.1 The control unit

The CU will be a D-type flip-flop "one-hot" sequencer, of the sort illustrated in Lecture 1. Any sequencer, for example, one based on a PROM, would do: the advantage of the one-hot method is that it makes very clearly what is going on.

The controller must first run the instruction fetch and appropriate decoding sequences before executing the sequence for the particular opcode fetched into **IR** (opcode).

We already understand that the outputs from the CU must be levels (CSLs) to establish data pathways between the various registers and memory, and pulses (CSPs) to fire register transfers. In addition, the levels are also used to to set the modes of operation for the ALU, PC and SP.

3.1.1 What levels and pulses are required?

To progress we need to look at the wiring in the CPU at a greater level of detail, as given in Fig. 3.1.

Output Enable(s): We know that every register has an Output Enable (OE) input which determines whether the tri-states on its output lines have a high or low resistance. These are driven by levels such as OEac, OEpc and so on. We assume that OEac=1, and similar, sets the tri-state for the AC in low resistance mode.

Next, as a more careful study of the CPU diagram Fig. 3.1 shows, some extra tri-state buffers are required to satisfy every register input that has more than one potential path into it. These are labelled OE1 to OE7.

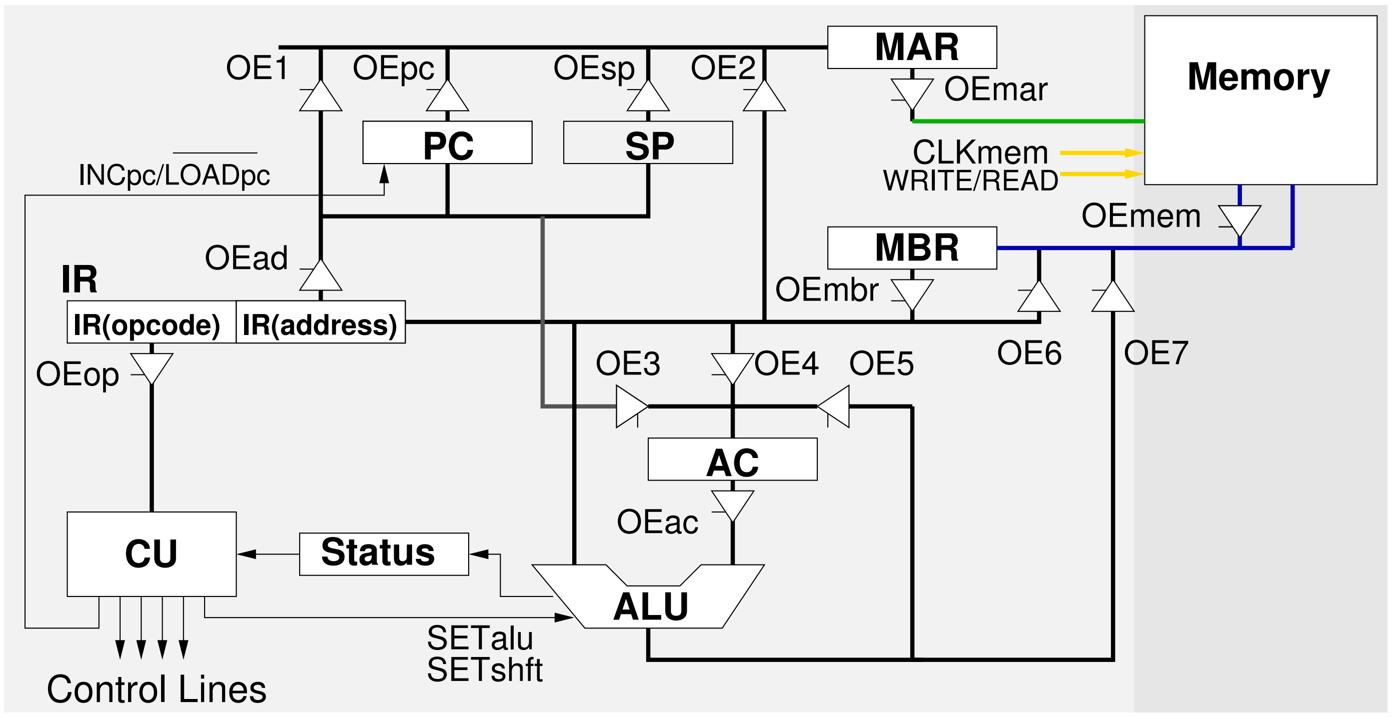


Figure 3.1: Tri-state Output Enables. 8 extra buffers are required.

ALU: We develop internal hardware for the ALU in §3.2. For now, we treat it as a black box with 8 or fewer functions, which thus require 3 level bits SETalu[2:0] to define. These are specified in the following table.

SETalu	Operation	Comment
000	ALUnoop	Do nothing. Let the AC input appear on output
001	ALUnot	Invert each bit in the AC input
010	ALUor	Output = AC .OR. MBR
011	ALUand	Output = AC .AND. MBR
100	ALUadd	Output = AC + MBR
⋮	⋮	

PC: The Program Counter has a one-bit level input which tells it whether to load the input or to increment when the clock pulse is received.

SP: The Stack Pointer has a two-bit level input which tells it whether to load the input, increment, or decrement, when the clock pulse is received.

LOADpc	When CLKd
0	Increment
1	Load from bus

LOADsp	INCsp	When CLKd
0	1	Increment
0	0	Decrement
1	X	Load from bus

Now we can rewrite the instruction fetch in terms of levels and pulses as follows:

Instruction fetch (levels and pulses)
1. OEpc=1; CLKmar;
2. OEmar=1; WRITE=0; OEmem=1; CLKmbr;
3. OEmbr=1; CLKir; INCpc=1; CLKpc;
4. Then decode using **IR** (opcode)

3.1.2 Execution levels and pulses

Now consider the execution phase of a few of the instructions in terms of levels and pulses

LDA x (levels and pulses)
10. OEad=1; OE1=1; CLKmar;
11. OEmar=1; WRITE=0; OEmem=1; CLKmbr;
12. OEmbr=1; OE4=1; CLKac;

STA x (levels and pulses)
13. OEad=1; OE1=1; CLKmar;SETalu=ALUnoop; OEac=1; OE7=1; CLKmbr
14. OEmar=1; WRITE=1; OEmbr=1; OE6=1; CLKmem;

NB: SETalu=ALUnoop (=000) allows the AC's input to appear at the ALU output with no change at all.

ADD x (levels and pulses)
15. OEad=1; OE1=1; CLKmar;
16. OEmar=1; WRITE=0; OEmem=1; CLKmbr;
17. OEmbr=1; OEac=1; SETalu=ALUadd; OE5=1; CLKac;

and so on ...

We can already start sketching out a one-hot controller design — that in Fig. 3.2. The first three D-types handle the fetch, then at D-type #4 comes the decoding. Then if STA were high, say, the "hot 1" would pass to D-type #13 then #14 which execute STA, then back to the fetch, and so on.

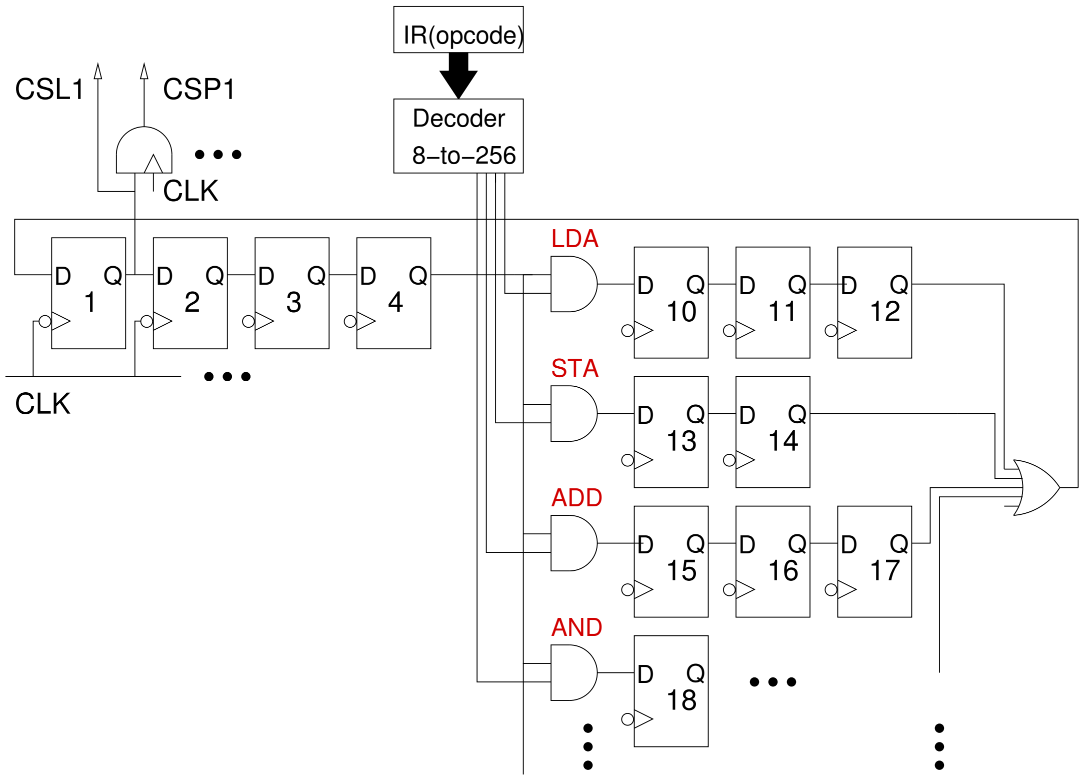


Figure 3.2: The control unit for the first four opcodes.

3.1.3 Hardware for Decoding

You will have spotted in Fig. 3.2 that to decode the opcode we use (what genius!) a 8-to-256 decoder. The example in Figure 3.3(a) considers just the low 3 bits of the opcode. If we ignore the long opcode problem:

Decoding (this is RTL)

4. →(LDA,STA,ADD,AND, ..., SHR,HALT)/(10,13,15,18,...,25,99)

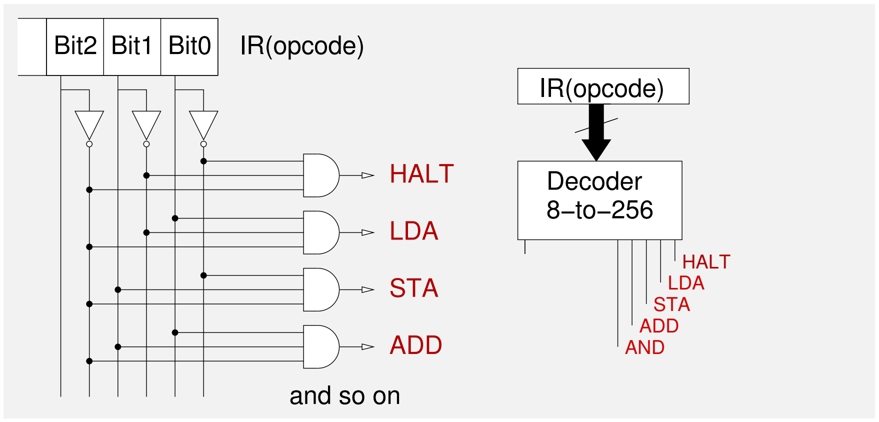


Figure 3.3: Decoding the opcode. (a) With gates for using bits, and (b) as a black-box.

3.1.4 Hooking up the CSPs and CSLs

What we have to do now is to figure out is to what to connect the various CSPs and CSLs. Don't panic — this is quite straightforward … For the first step of the Fetch cycle we need to Output Enable the **PC** and clock the **MAR**– so a 1 goes in these columns. All blank spaces are zeros, and X denotes "don't care". Then carry on filling in the 1's for the remaining lines of Fetch and for the lines of the execute phases of the instructions — that is, filling in horizontally.

		Levels																				Pulses						
What	Line	LOADpc	LOADsp	INCsp	OEpc	OEsp	OEad	OEop	OEmar	OEmbr	OEac	OE1	OE2	OE3	OE4	OE5	OE6	OE7	SETalu[2]	SETalu[1]	SETalu[0]	OEmem	CLKpc	CLKsp	CLKmar	CLKmbr	CLKir	CLKac	CLKmem
Ftch	1.				**1**														X	X	X				**1**				
	2.								**1**										X	X	X	**1**				**1**			
	3.									**1**									X	X	X		**1**				**1**		
Dcd	4.							**1**											X	X	X								
LDA	10.						**1**					**1**							X	X	X				**1**				
	11.								**1**										X	X	X	**1**				**1**			
	12.									**1**					**1**				X	X	X							**1**	
STA	13.						**1**				**1**	**1**						**1**	0	0	0				**1**	**1**			
	14.								**1**	**1**							**1**												**1**
ADD	15.																												
	16.																												
	17.																												
AND	18.																												
	19.																												
	20.																												
JMP	21.																												
BZ	22.																												
	23.																												
NOT	24.																												
SHR	25.																												
HLT	99.																												

Now look down the columns for each signal. As a work in progress, after filling in the Fetch, Decode, and the execute phases of LDA and STA, we have already
OEpc = CSL1
OEad = CSL10 .OR. CSL13
OEmar = CSL2 .OR. CSL11 .OR. CSL14
CLKmar = CSP1 .OR. CSP10 .OR. CSP13

3.2 The Arithmetic Logic Unit

The ALU is the only part of the CPU that does anything to the information — the rest just shovels information from one place to another. The ALU is designed to perform both logical operations and arithmetic operations. Looking back at the CPU diagram, you will recall that it has two inputs, one from the **AC** and the other from the **MBR**.

Figure 3.4 shows a 1-bit slice of the ALU. Input A is one bit from the AC and Input B is one bit from the MBR. The logic unit here allows 4 operations, including a no-op and inversion of the AC input. Also shown is a full adder with carry in and out. Three lines are used to select the ALU's function via a decoder (which is not bit sliced of course).

Notice that all the logic is combinatorial. The speed of the ALU is limited only by delay time in the logic gates. There is no synchronizing clock. Effectively this means that the settling time should be less than the register transfer clock rate.

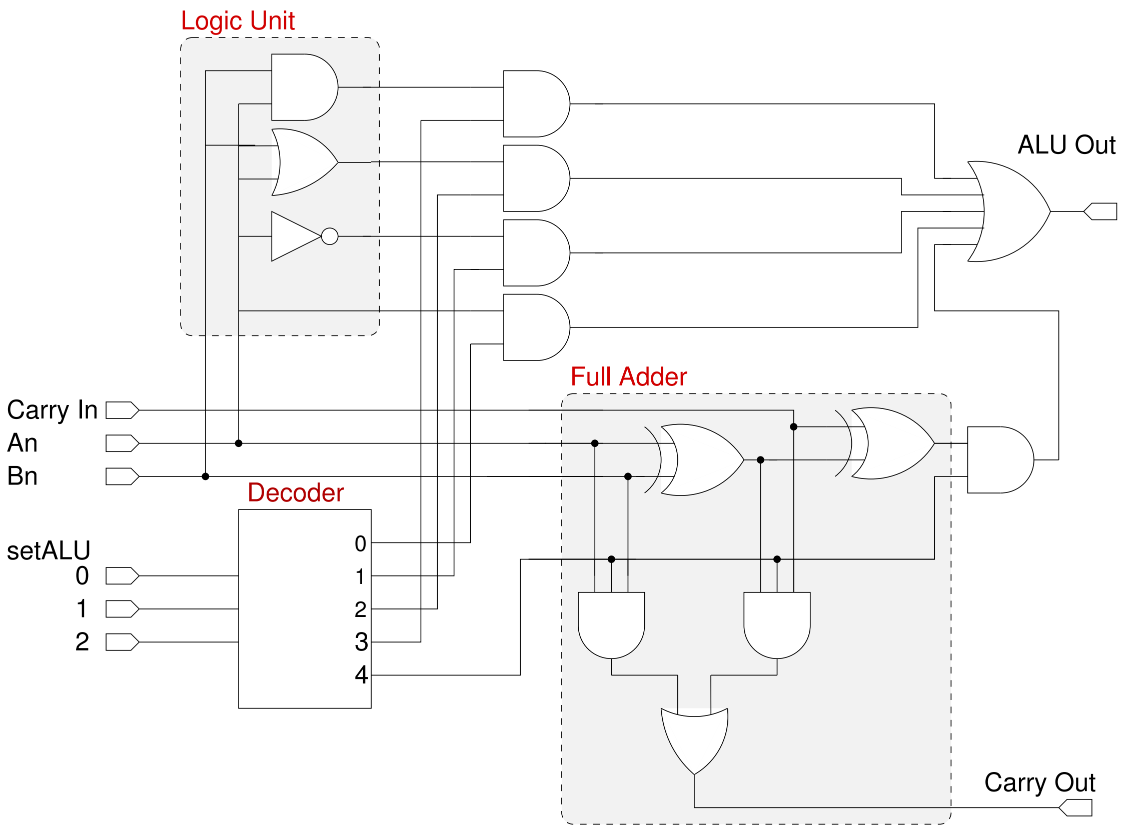


Figure 3.4: A bit-slice of the ALU.

3.2.1 Multi-bit bit-slice ALU

To build a multi-bit ALU, we simply stick the 1bit ALUs together, as shown in the Figure 3.5. You learned in your 1st Year lectures on Digital Logic that using a ripple carry is slow, and that speed-up can be achieved by inserting carry-look-ahead circuitry between a number of bits.

The ALU contains a none/left/right shifter at its output. This is operated separately from the other functions using a 2-bit setSHFT input (so one can add two numbers and rightshift them all in one pass). Some designs use a shift register, but here we regard it a black box which *transparently* allows bit[n] of the input to appear as one of bit[n], bit[n-1], or bit[n+1] of the output.

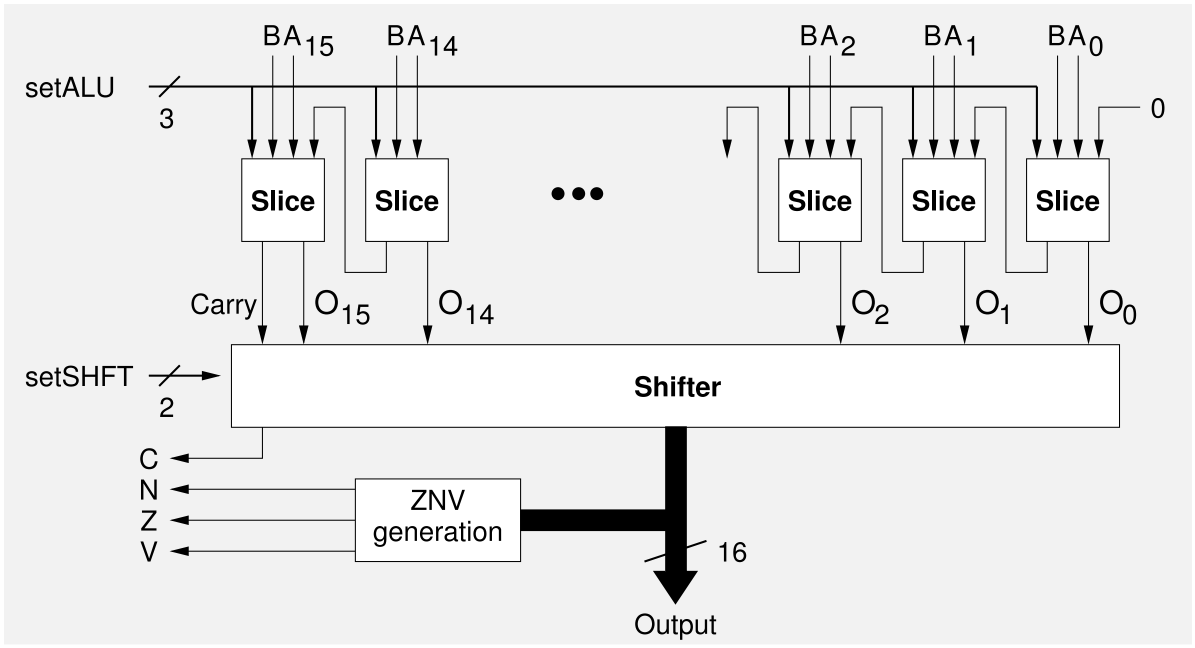


Figure 3.5: Multi-bit ALU. The carry-in on the right must be 0. A shifter is placed on the back end. In our design it is not a clocked register.

3.2.2 Flags set in the Status Register by the ALU

An important function of the ALU is to set up bits or flags which give information to the control unit about the result of an operation. The flags are grouped together in the status word.

As the ALU has only an adder, to subtract numbers one has to use 2s-complement arithmetic. The ALU has no knowledge of this at all — it simply adds two binary inputs and sets the flags. It is up to control unit (or really the programmer's instructions executed by the control unit) to interpret the results.

Z **Zero flag:** This is set to 1 whenever the output from the ALU is zero.

N **Negative flag:** This is set to 1 whenever the most significant bit of the output is 1. Note that it is not correct to say that it is set when the output of the ALU is negative: the ALU doesn't know or care whether you are working in 2's complement. However, this flag is used by the controller for just such interpretations.

C **Carry flag:** Set to 1 when there is a carry from the adder.

V **oVerflow flag**: Set to 1 when $A_{msb} = 1$, $B_{msb} = 1$, but $O_{msb} = 0$; or when $A_{msb} = 0$, $B_{msb} = 0$, but $O_{msb} = 1$. Allows the controller to detect overflow during 2's complement addition. Note (i) that the ALU does not know *why* it is setting the flag, and (ii) in our BSA, the msb=15.

3.3 Counter Registers

The PC and SP are counting registers, which either act as loadable registers (loaded from the **IR** (address) register) or count, both on receipt of a clock pulse. The internal workings appear complicated, but are unremarkable, so we leave them as black boxes here.

3.4 Memory

You will have gathered that memory is no more than a very large collection of registers held on an array on a chip, one register being accessible at a time via an addressing mechanism.

To write to memory, one needs to set the address, place the data on the data bus, and clock the recipient register. To read from memory, one needs to set the address, output enable the relevant register onto the data bus, and clock the MBR.

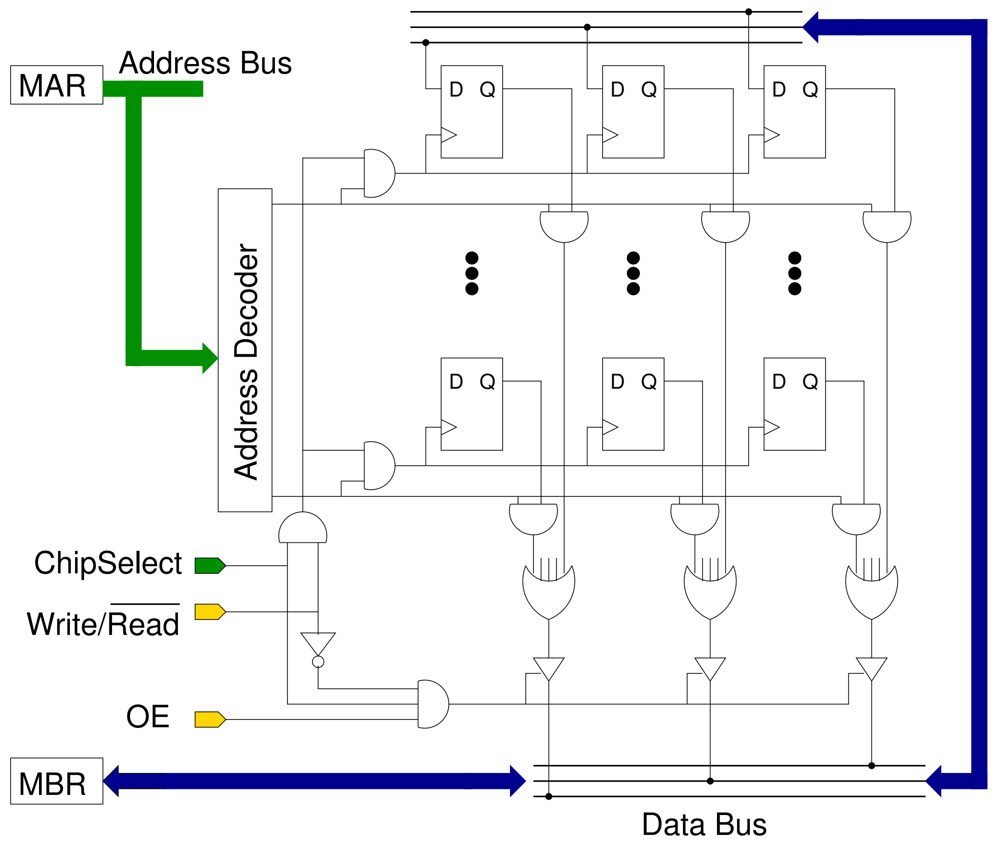


Figure 3.6: Memory hardware Note that the input data lines (top) and output data lines (bottom) are different. They are not of course — they are the same bus and have been drawn like this to avoid crossing wires.

Figure 3.6 shows the design of a memory with contents 3 bits wide.

The address lines enter a decoder, which selects one register, ie one row of three D-type latches with couple clock inputs. Notice that that each output from the decoder is used in two ways: first for writing it is ANDed with the clock signal, and second for reading it is ANDed with the register outputs.

The ChipSelect (CS) input is unexpected, as one might expect the memory to be permanently selected! We shall see in a moment that it actually used as part of address selection in a memory made from multiple chips. Assume it is CS=1 for now.

The OE input is expected, but instead of a CLKmem input we now use the more usual name WRITE/$\overline{\text{READ}}$. (By the way, this notation means WRITE is the "equivalent" of $\overline{\text{READ}}$.)

At first this looks like a level selector signal but, by process of elimination, this *must* be the clocked input.

3.4.1 When reading:

WRITE=0, and the output of (CS.AND.WRITE)=0 so that all register CLK inputs are low (which is good). Now, CS=1 and OE=1, so that the 3-input AND gate enables the tri-state outputs. The actual register outputting is determined by the Address decoder's output. Looking at the outputs from the latches, the enabled outputs are ORed with all the disabled outputs, and the three outputs head off towards the data bus. Notice that the outputs are Output-enabled using tri-state logic onto the bus.

The timing is shown in Fig. 3.7.

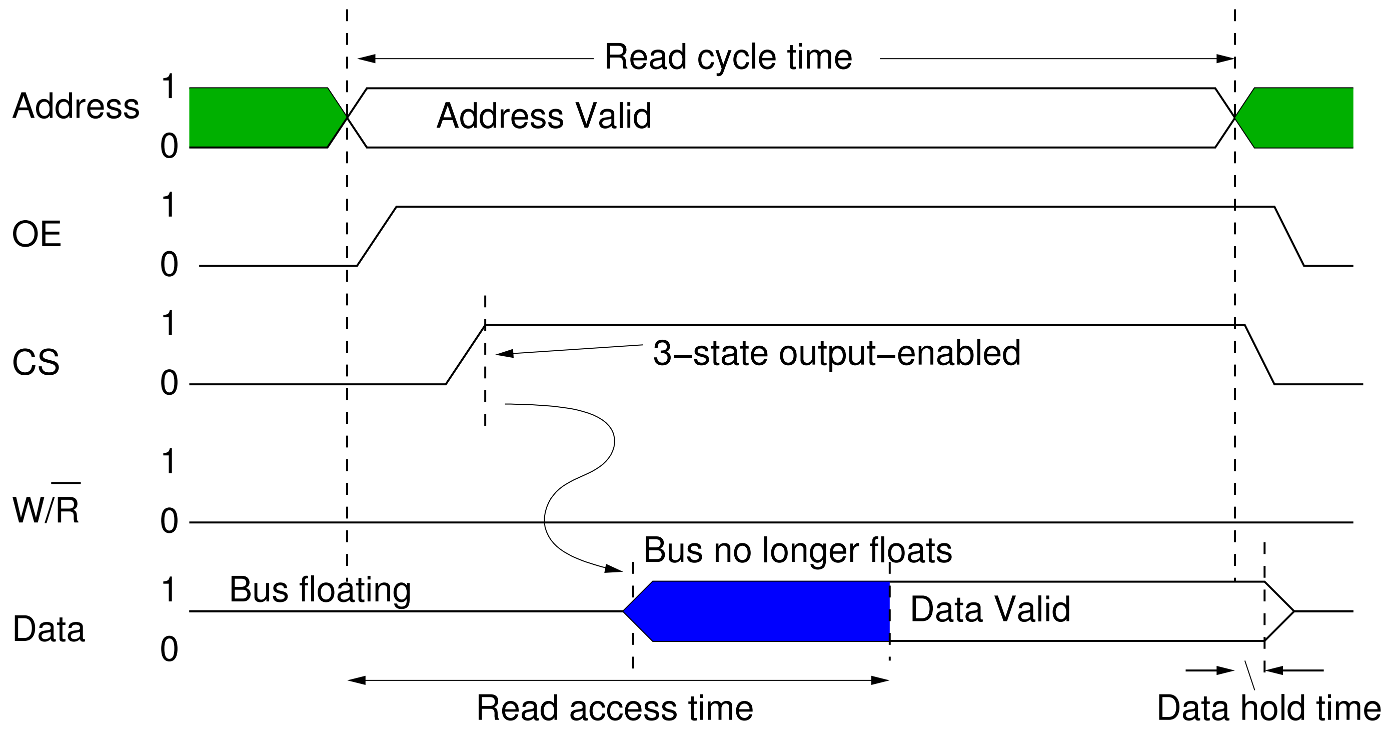


Figure 3.7: Timing of signals for (a) memory read.

3.4.2 When writing:

CS=1, OE=0, and WRITE changes from 0→1, so that the CLK inputs on the register selected by the address are all high. Then WRITE changes from 1→0 causing the clocks to fall triggering the register transfer.

The timing is shown in Fig. 3.8.

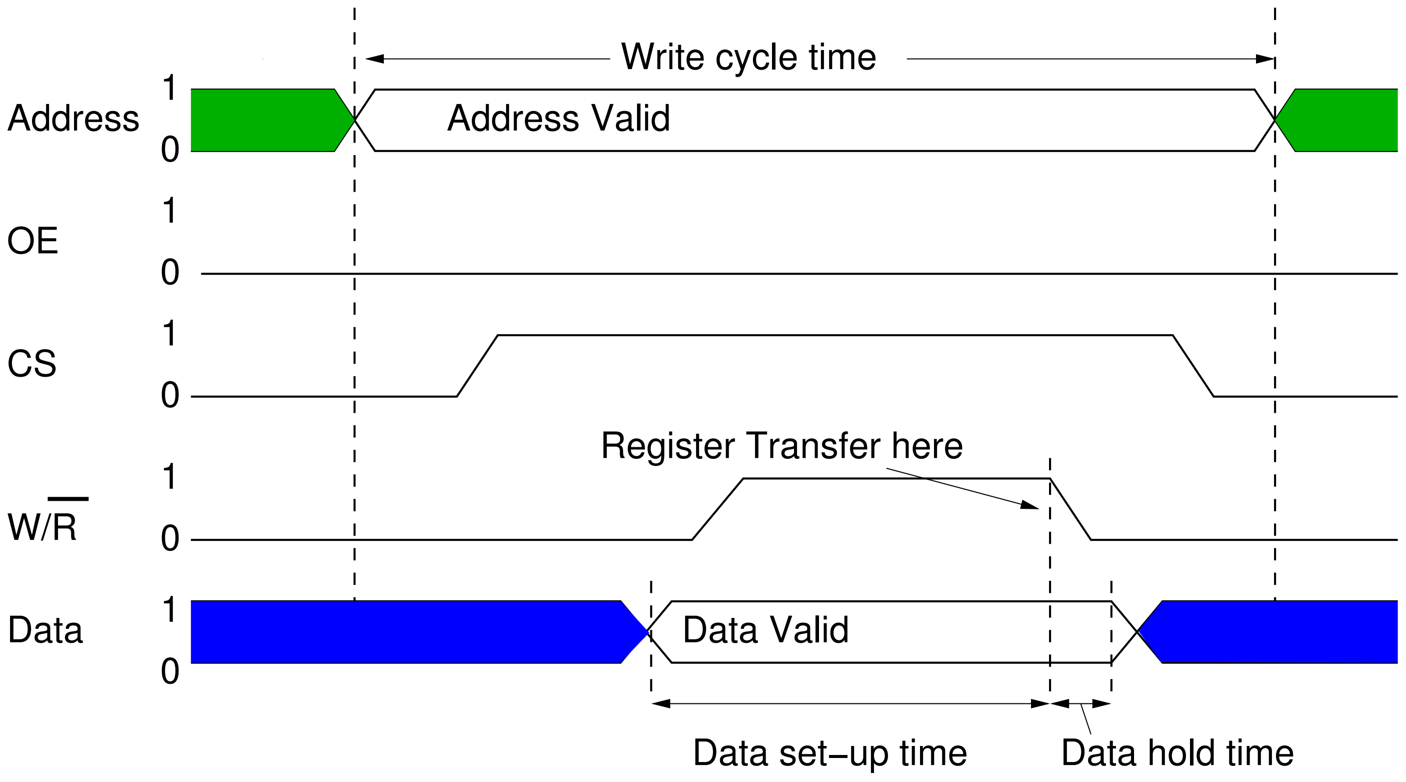


Figure 3.8: Time of signals for the memory write. Notice that the signal which clocks the register on the memory during write is the WRITE/$\overline{\text{READ}}$ signal.

3.5 Memory organization in hardware

The size of memory chips has risen over the years. Though technically feasible to build very large single chip memories, potential sales and problems of yield make it much more economical to produce "reasonably sized" chips that find application in small memories, but can be built up into larger memories.

Data Width: Standard memory chips are 1Byte wide, so our 16bit data bus requires two chips side by side. The same address lines enter both chips, but the data lines are split between the high 8 bits and low 8 bits. The arrangement is shown in Fig. 3.9.

Address Height: n address lines can access 2^n locations — in the case of 16 lines that is 16 M locations. Now suppose the available memory chips were 8 MByte. We need

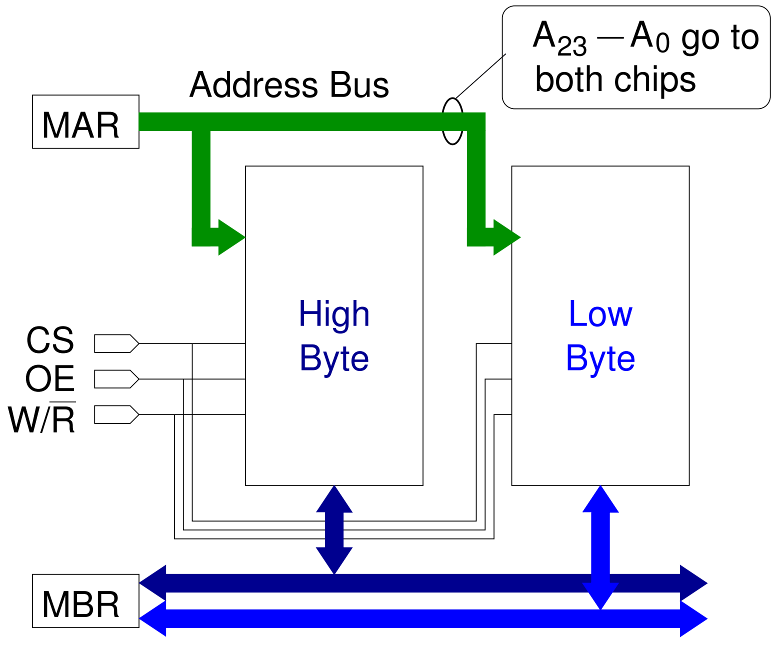


Figure 3.9: Using two Byte wide chips to make a 16-bit wide memory.

to generate an array 2 chips high and, as before, 2 wide, as shown in Fig. 3.10.

But each chip has only 23 address lines, $A_0 - A_{22}$. What happens to A_{23}? It is input into a 1-to-2 line decoder, whose output is connected to the ChipSelect inputs. If $A_{23} = 0$, the lower pair is selected, and if $A_{23} = 1$ the upper pair is selected. The OE and W/$\overline{R}$ inputs are connected to *all* chips.

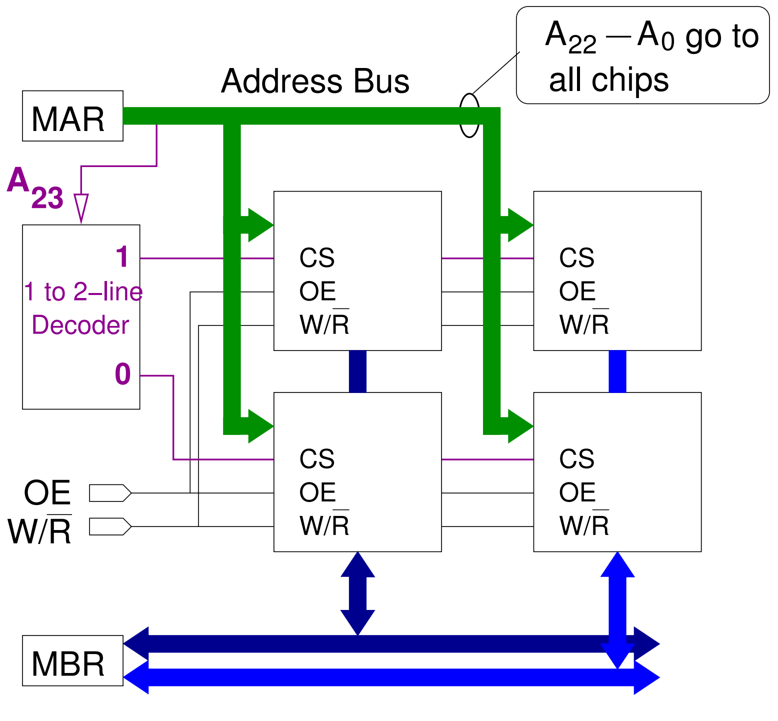


Figure 3.10: Using 8MByte chips to create a memory with 16M locations and a width of 2 Bytes.

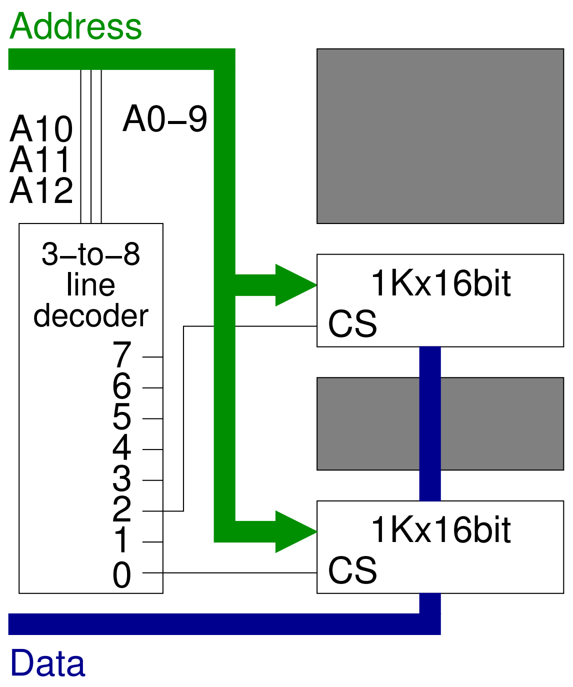
3.5.1 Memory: address space versus physical memory

The *n* address lines give the ability to address 2^n different locations. These locations span the *address space* 0x0 to 0xFFFFFF for our 24-bit address bus. However, there is no need either for (i) for the entire address space to be occupied by physical memory, or (ii) for the physical memory that is fitted to be located contiguously in address space. There can be gaps.

Exactly how the physical memory is mapped onto the memory space depends on how the address lines are decoded.

As an example, suppose we have 13 address lines A0–A12. These can address 8K (ie 8192) locations in memory. Suppose also that we have just two 1K (ie 1024) word memory chips M1 and M2. Each *must* use the lowest ten address lines A0–A9.

If all the lines were decoded (Fig. 3.11) the mapping between address and location is unique. The valid and invalid address ranges are shown in the table.



Into Dec			Into Mem										Hex
A12	A11	A10	A9	A8	A7	A6	A5	A4	A3	A2	A1	A0	Addr
1	1	1	1	1	1	1	1	1	1	1	1	1	0x1FFF
					No physical memory								
0	1	1	0	0	0	0	0	0	0	0	0	0	0x0C00
0	1	0	1	1	1	1	1	1	1	1	1	1	0x0BFF
0	1	0	0	0	0	0	0	0	0	0	0	0	0x0800
0	0	1	1	1	1	1	1	1	1	1	1	1	0x07FF
					No physical memory								
0	0	1	0	0	0	0	0	0	0	0	0	0	0x0400
0	0	0	1	1	1	1	1	1	1	1	1	1	0x03FF
0	0	0	0	0	0	0	0	0	0	0	0	0	0x0000

Figure 3.11: Full address decoding, but using only 2× 1K memories in an 8K address space.

3.6 Memory organization in software: memory addressing modes

Our knowledge of memory hardware tells us that a memory works in just one way — you stick the address on the address lines and then either read or write to the contents at that address. The different modes of memory addressing refer then not to the hardware level, but to different ways of using what you read from the memory. We shall consider

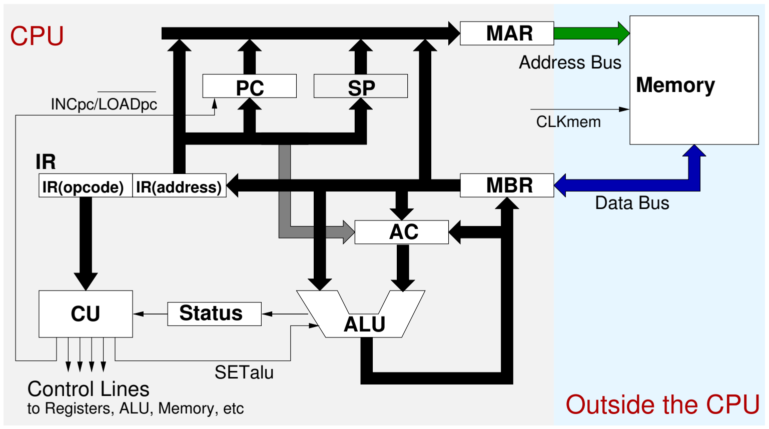
(1) **Immediate**, (2) **Direct**, (3) **Indirect**, and (4) **Indexed** addressing.

3.6.1 Immediate addressing

Immediate addressing does not involve further memory addressing after the instruction fetch. It provides a method of specifying a *constant number* rather than an address in the *operand*. For example LDA# x loads the accumulator immediately with the operand x.

LDA# x
AC←**IR** (address)

ADD# 22
AC←**AC** + 22



Looking back at our Standard architecture, you will see that there is a direct link from the **IR** (address) to the **AC** to allow this to happen.

Immediate addressing allows statements like "n=n+10" written in some high level language to be turned into assembler, using modified versions of other instructions. For example ADD# 22. How would you actually realize this on our machine?

3.6.2 Direct addressing

We have already use direct addressing in the lectures. This is where the operand is the address of the data you require. Another way of saying this is that the operand is a *pointer* to the data.

LDA x
MAR←**IR** (address)
MBR←⟨**MAR** ⟩
AC←**MBR**

ADD x
MAR←**IR** (address)
MBR←⟨**MAR** ⟩
AC←**MBR** + **AC**

etc

♣ **Quick Example:** What does location 23 and the AC contain after this code snippet?

Code	AC	Loc22	Loc23
LDA #21			
STA 22			
ADD #1			
ADD 22			
STA 23			

3.6.3 Indirect addressing

In indirect addressing the operand is the address of the address of the data. That is, if we look in the memory at address *x* we don't find the data but rather another address. We then have to look at this new address to find the data.

LDA (x)
MAR$\leftarrow$**IR** (address)
MBR$\leftarrow\langle$**MAR** $\rangle$
MAR$\leftarrow$**MBR**
MBR$\leftarrow\langle$**MAR** $\rangle$
AC$\leftarrow$**MBR**

It is obvious that we need an extra memory access to use indirection — so why is it used?

The key reason is that it makes possible the use of data arrays for which space is allocated *during execution* not *during compilation* of a program.

There is a fuller explanation later.

3.6.4 Indexed addressing: an example of register addressing

LDA x,X

Cpus often provide a number of registers for temporary storage which avoid the need to hold and access pointers in main memory (with obvious savings in time). Methods which use these registers are known as register addressing modes.

Indexed addressing is a straightforward example of register addressing, and the only one we consider in these lectures. In the example above, x is an address, and X is an *index register* holding an offset. The effective address given to LDA is x+X, the sum of the two.

The register X will be a counter register, and can be loaded separately, incremented and decremented.

Here is some half-baked code for Matlab-esque addition of two arrays of length 100, and placing the result in a third array ...

```
      LDX  #0       // zero the index register
Loop: LDA  100,X    // load AC with Xth of array
      ADD  200,X    // add the Xth of another array
      STA  300,X    // store as Xth element on a third array
      INX           // increment X
      JMP   Loop    // do it again
```

What needs fixing?

3.6.5 Addressing Examples

The example shown in Figure 3.12 shows LDA (2), compared with LDA 2, LDA #2 and LDA 2,X. The contents at address 2 are 47, so that we look in 47 for the data, 38, which is then loaded into the accumulator.

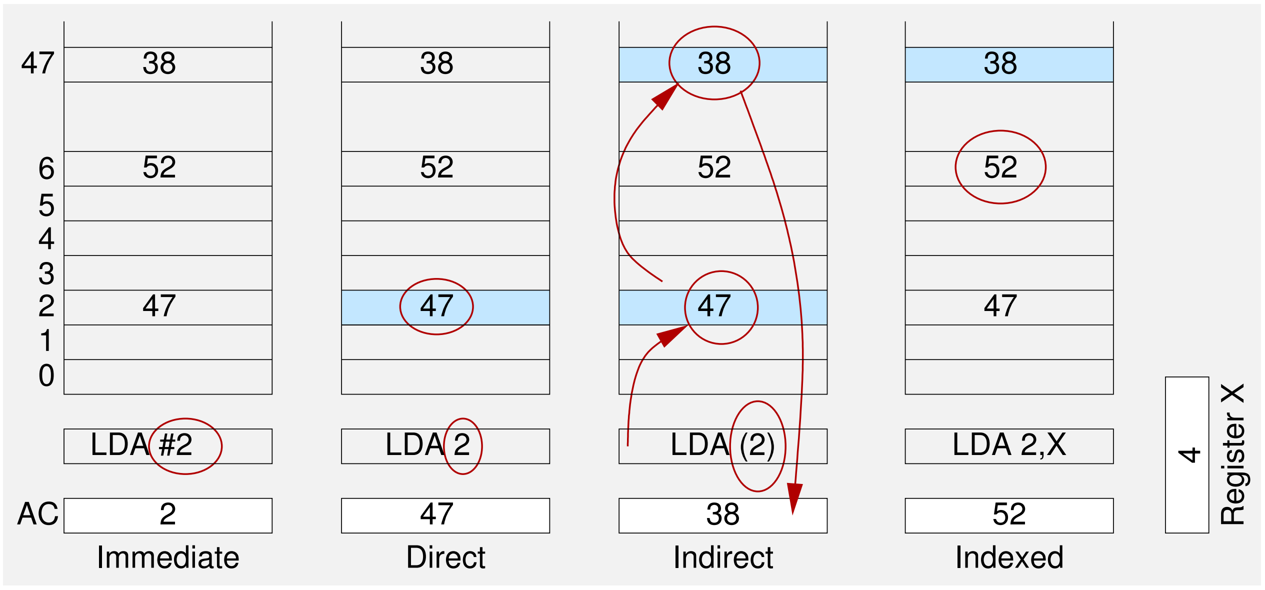


Figure 3.12: Addressing modes: the contents of the AC after LDA immediate, direct, indirect and indexed

3.7 A small program

It is worth pointing out that you now know enough to build (and program) a simple computer by looking at a small program written in assembler and seeing how it appears in the memory.

```
        LDA 20     // LOAD AC with contents at location 20
AGAIN:  SUB 22     // SUBTRACT from AC contents of location 22
        BZ STOP    // If ALU gives 0, Z=1, so jump to label STOP
        LDA 20     // LOAD AC with contents of location 20
        ADD 21     // ADD contents of location 21
        STA 20     // STORE in location 20
        JMP AGAIN  // JUMP back to label AGAIN
STOP:   HALT
```

We have introduced another assembler mnemonic SUB — let it have opcode 9, ie %00001001. Also we will use just 8-bit operands, so that the high 16-bits of the address are all zero.

Let us also assume that the first instruction gets stored at location 0 in memory. (This is not a general requirement. All that is required is that the PC is initialized at the correct starting location.)

The location and contents of the memory relating to the program are as follows. Notice how the line labels are assembled as *memory locations*. This is because they are going to supply information to the **PC** during the JMP and BZ instructions. Data has been inserted at locations 20-22.

Instru-ction	Loca-tion	Memory contents High Byte OPCODE	Low Byte OPERAND	Comment
LDA 20	0	00000001	00010100	Program starts
SUB 22	1	00001001	00010110	
BZ 7	2	00000110	00000111	
LDA 20	3	00000001	00010100	
ADD 21	4	00000011	00010101	
STA 20	5	00000010	00010100	
JMP 1	6	00000101	00000001	
HALT	7	00000000	00000000	Program ends
	:	:	:	
	20	00000000	00000101	These are data: dec 5
	21	00000000	00000001	dec 1
	22	00000001	00101101	dec 300

Initialize **PC** to 0 and start the controller running by starting the clock. The **AC** gets loaded with 5 (from location 20), and then has 300 (from location 22) subtracted from it. Because 5 isn't equal to 300, the Z flag does not get set, and so we reload the **AC** with 5, add 1 (from location 21) to it, and then restore it in location 20. The program eventually halts when location 20 is incremented up to 300. (Halting simply stops the clock.)

Notice that there is no *requirement* for assembler mnemonics or for that matter a high level language. The raw form of the executable program and data is **binary code**. In the good old days, when folk were short of things to do, binary code was loaded into the memory by hand.

3.8 Hardware support for higher level languages (Reading)

The emphasis in this short course is on the register-level operation of hardware in CPU and memory. However, using RTL we *have* been able to make the link between the register-level and what is called the "macro-level" or assembler language level. This has probably happened without your being aware of it. The link was made by writing down LDA x as an overall register transfer **AC** $\leftarrow \langle x \rangle$, then breaking this down into the individual transfers that are reliaziable on our actual architecture.

Unfortunately, there has been and is no time to make the link between the macro-level and the way all of us programme computers using high-level languages such as C++, C, Java, Fortran, and so on; languages which allow complete abstraction from the details of the cpu that actually executes the program.

This section of the notes describes the process of compilation, which will in turn clarify why Indirect Addressing is useful. It ends with an explanation of how subroutines appear at the macro-level, and how they use the memory stack.

3.8.1 Compilation of programmes

Before you can execute a high-level program on a computer, it has to be compiled into a set of instructions for the particular cpu being used. In their binary opcodes and operand form, the program is said to be in machine code. One level higher than this is assembler code, which uses mnemonics for the opcodes, and allows one to label locations (eg the JMP AGAIN seen earlier: AGAIN was a label for a location). Now, each cpu comes with its own assembler program which turns assembler into machine code (you will use one of these on the lab course), and so there is no need for a compiler to produce machine code: it need only produce assembler.

The aim of the compiler, in the simplest cases, is to produce a program where instructions occupy one section of memory and where allocated data memory lies above the instructions. When you execute the program, you rely on the computer's operating system to be able to provide extra free space (called the Heap) above the program and allocated data, and to provide a stack. (In early days, when memories were small, the programmer could specify the amount of memory to be given to Heap and Stack, so that the OS knew *exactly* how much space to allocate to your program. The OS would refuse to run the program if there was insufficient space.)

The compiler works in two stages.

On the first pass

- it reads the file of high level statements, checks the syntax, and replaces each statement with the relevant sequence of assembler instructions.
- it places declared variables, and the names of labels, into a symbol table.
- each time the variable or label is encountered, it replaces the name in the code by its location in the symbol table.

The instructions are written out into a temporary file. Since each instruction is of known length, the compiler can associate with it provisional memory addresses, relative to the Beginning of the Program (BOP). So, after the first pass, the compiler knows the full extent of the program — or, more strictly, the extent of the instructions. After the first pass, the compiler therefor knows where it can start placing allocated data. We will call this memory location BOD (for Beginning of Data).

On the second pass the compiler

- replaces locations in the symbol table by proper locations in memory, relative to BOP, the Beginning of Program.

For example, consider the fragment

```
int a,b,c; // declaration of variables and their lengths
a=1;
b=2;
c=a+b;
if(c!=0) {
   a=3;
}
b=4;
```

After reading the declaration of variables, the compiler could start making the symbol table, where BOD indicates the unknown Beginning Of Data.

Name	SymTab	Actual Location	Words
a	var0	BOD	1
b	var1	BOD+1	1
c	var2	BOD+2	1

Let the beginning of the program be located at BOP, and let us count in words. The compiler would start making its temporary file:

Address	Instruction
BOP	LDA #1
BOP+1	STA var0
BOP+2	LDA #2
BOP+3	STA var1
BOP+4	ADD var0
BOP+5	STA var2
BOP+6	BZ lab0

At this point, the compiler knows that the label will be just after the closing bracket, but doesn't know where that is. So it adds an entry to the symbol table

Name	SymTab	Actual Location	Words
a	var0	BOD	1
b	var1	BOD+1	1
c	var2	BOD+2	1
	lab0	?	-

then carries on.

Address	Instruction
BOP	LDA #1
BOP+1	STA var0
BOP+2	LDA #2
BOP+3	STA var1
BOP+4	ADD var0
BOP+5	STA var2
BOP+6	BZ label0
BOP+7	LDA #3
BOP+8	STA var0
BOP+9	LDA #4
BOP+10	STA var1
BOP+11	HALT

By the LDA #4 instruction, the compiler knows that lab0 is actual BOP+9, so the symbol table is updated

Name	SymTab	Actual Location	Words
a	var0	BOD	1
b	var1	BOD+1	1
c	var2	BOD+2	1
	lab0	BOP+9	-

When it reached the end of the program, the compiler knows that the beginning of data BOD = BOP+12. It then rewrites the symbol table as:

Name	SymTab	Actual Location	Words
a	var0	BOP+12	1
b	var1	BOP+13	1
c	var2	BOP+14	1
	lab0	BOP+9	-

There are now two possibilities for the second pass. The compiler can either rewrite var0, var1 and var2 and lab0 leaving BOP as a variable to be filled in at run time, or it can set a value a value for BOP. Choosing the latter with BOP=100 (dec), we end up with:

Address	Instruction
100	LDA #1
101	STA 112
102	LDA #2
103	STA 113
104	ADD 112
105	STA 114
106	BZ 109
107	LDA #3
108	STA 112
109	LDA #4
110	STA 113
111	HALT
112	0
113	0
114	0

Now simply replace the opcodes by the relevant binary, and we have executable machine code. Notice that the assembler has to use a different opcode for LDA, LDA#, and so on.

3.8.2 Indirect addressing

Knowledge of compilation helps with, but is not essential for, undertanding the value of indirect addressing.

Suppose we wrote the following snippet of code ... The binary executable program will comprise instructions (opcodes and operands) and fixed size data whose locations in memory are all known once the program is loaded.

```
my_prog() {
// First some declarations
 int d1, d2, array[10];
 float x;
// Now the actual instructions
  d1=2;  d2=d1+1;
  array[1] = d1*d2;
  x=0.5;
}
```

We can imaging the program appearing memory as in Fig. 3.13(a).

During execution you are allowed (of course!) to change *values* of the declared data, but not their *locations* — and you cannot change the values or locations of the opcodes and operands.

However, suppose you wanted to read in an array (an image, say) whose size you did not know beforehand. You could declare a fixed size array that would be big enough to handle anything, but that is wasteful. Instead, you declare in the fixed-size data area space to hold the address of the array when it becomes known during execution. The actual space for the array is found on the "heap" of free memory during execution. See Fig. 3.13(b).

The program knows the location where the address will be placed, and so can access the array using indirection.

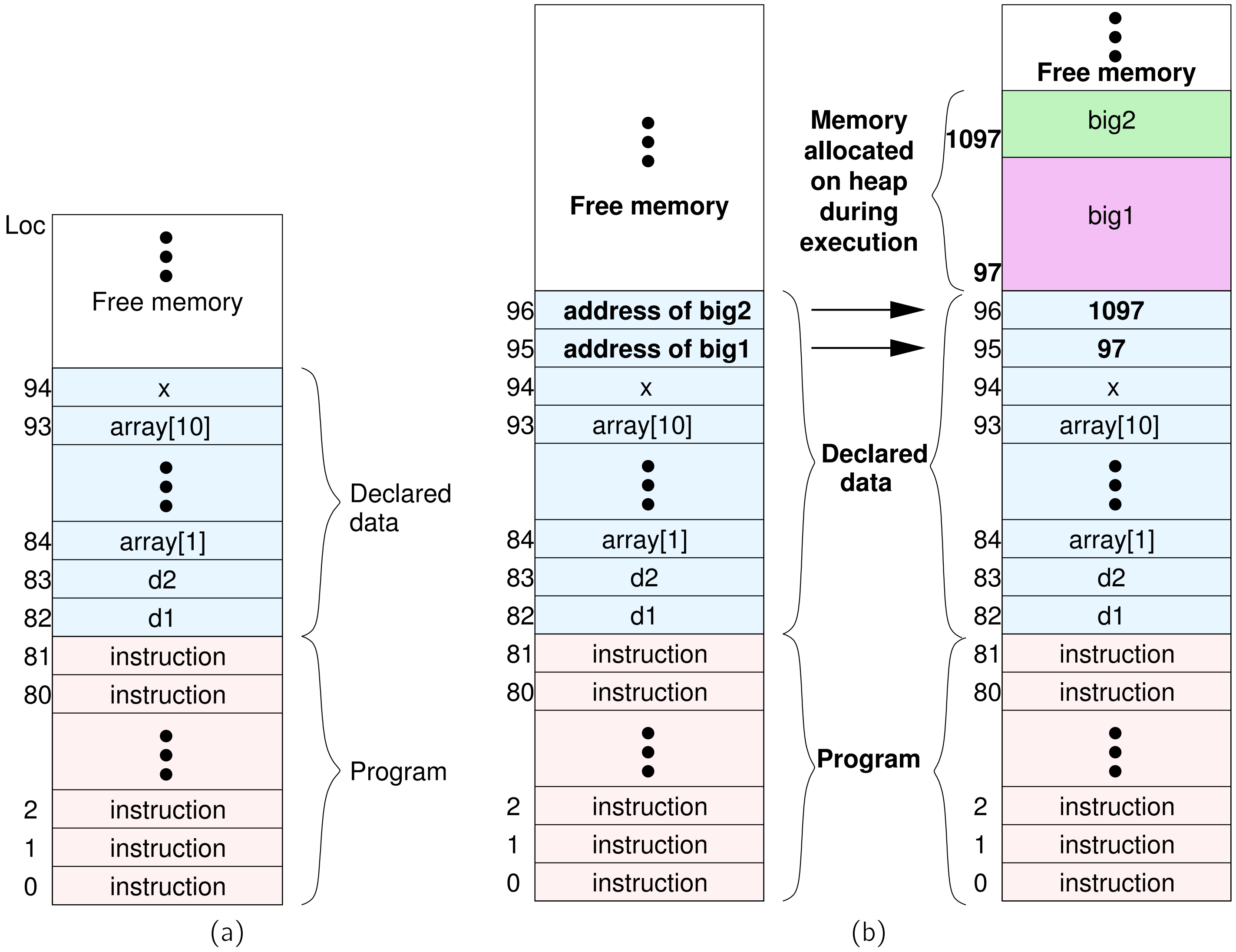


Figure 3.13: (a) All data of known size when the program is written. (b) Sizes of big1 and big2 unknown beforehand. Space is allocated on the heap. Only the address of the address is known beforehand, not the address itself.

3.8.3 Stacks (then Subroutines)

Looking back at the Bog Standard Architecture, you will recall that there is a register called **SP**, the stack pointer. This register holds the address of the entrance to an area of memory reserved by a program as temporary storage area. The stack pointer uses memory as a last-in, first-out (LIFO) buffer. Usually it is placed at the top of memory, as far away from the program as possible.

Figure 3.14 shows a model of memory. The stack currently contains 5 items and grows downwards. The stack pointer points to the next free location.

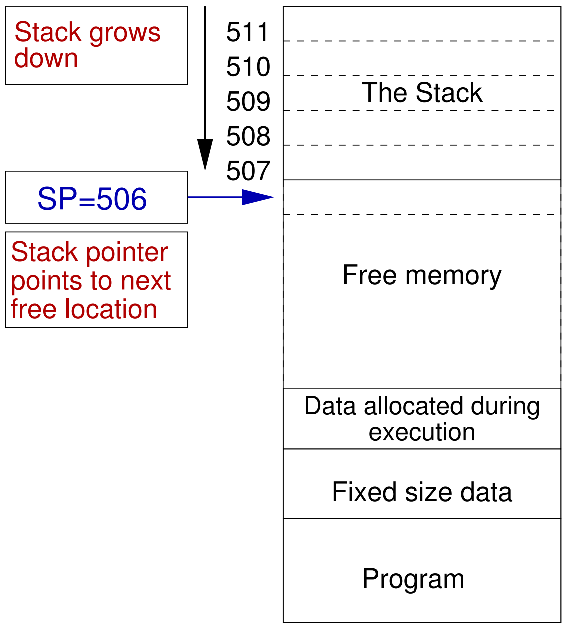


Figure 3.14: The program and its fixed size data is at the bottom of memory. Data allocated during execution fills up free space (the heap) bottom to top, while the stack is at the top of memory and grows downwards.

Two instructions work the stack, PUSHing and PULLing[1].

In PUSH, the accumulator gets pushed onto the stack at the address pointed to by the stack pointer. The stack pointer is then decremented.

In PULL, the stack pointer is first incremented and the contents pointed to transferred to the accumulator.

```
PUSH
MAR←SP
MBR←AC
⟨MAR⟩ ←MBR; SP←SP-1
```

```
PULL
SP←SP + 1
MAR←SP
MBR←⟨MAR⟩
AC←MBR
```

[1]often called POPing

3.8.4 Subroutines

Subroutines allow the programmer to modularize code into small chunks which do a specific task and which may be reused. For example,

```
main()                /* main program */
{
int a,b,v1;
a = 1;    b = 10;
v1    = mysub(a,b);   /* subroutine call */
c   = 0;
..
}

mysub(a,b)            /* subroutine specification */
{
ms = 2*(a+b);
return(msq);
}
```

How can we call a subroutine in assembler language. We need to (1) jump to the subroutine's instructions, (2) transfer the necessary data to the subroutine, (3) arrange for the result to be transferred back to the calling routine, and (4) jump back to handle the instruction after the calling point.

Let us consider the assembler for a subroutine where there are two parameters. One is in location 47, the other in location 48.

```
...  etc ...
     LDA #1    \\ a=1
     STA 47
     LDA #10   \\ b=10
     STA 48
     LDA 48    \\ 2nd parameter is datum b in loc 48
     PUSH      \\ push it onto stack
     LDA 47    \\ 1st parameter is datum a in loc 47
     PUSH      \\ push it onto stack
     JSR MYSUB \\ Now Jump to  SubRoutine at location MYSUB
     LDA #0    \\ rest of prog: these lines do c=0
     STA 49
... etc ...
                 \\ Subroutine code from here
MYSUB LDA SP,2
      ADD SP,3
      SHR
      RTS        \\ ReTurn from Subroutine
```

Notice that the subroutine starts at labelled address, so JSR is very like JMP. The difference is that we have to get back after the subroutine. After the fetch, the program counter is already incremented to point at the next instruction in the calling program (ADD in this example). So in its execute phase, JSR pushes the current value of the **PC** onto the stack, and then loads the operand into the **PC**, causing the next instruction fetched to be the first in the subroutine. The RTS command ends the subroutine by pulling the stored program counter from the stack. Because the stack is a LIFO buffer, subroutines can be "nested" to any level until memory runs out.

Missing out the detail the JSR and RTS instructions perform the following:

JSR x
⟨**SP** ⟩ ←**PC**
PC←**IR** (address); **SP**←**SP**-1

RTS
SP←**SP** +1
PC←⟨**SP** ⟩

When the subroutine has parameters, we also have to worry how to transfer the parameters to the subroutine. There are various ways, described in Clements §6.5 and Hill and Peterson § 15.2. Here we mention just one method which again uses the stack. Two methods are

- to pass parameters on a reserved area of memory
- to pass parameters on the stack

In the latter method, the calling routine pushes the parameters onto the stack in order, then uses JSR which of course pushes the return PC value onto the stack.

Figure 3.15 shows a very correct way of using the parameters from the stack by popping and pushing. The return **PC** is pulled and stored temporarily, and then the parameters are pulled, and the return **PC** pushed.

The problem with this is that it is very time consuming. A more efficient method is to use the parameters by indexed addressing relative to the stack pointer, **SP**. That is, the subroutine would access parameter 1, for example, by writing

```
LDA SP,2
```

When the subroutine RTS's it will pull the return **PC** off the stack, but the parameters will be left on. However, the calling routine knows how many parameters there were. However, rather than pulling them off, it can simply increase the stack pointer by the number of parameters (ie, by three in our example). This leaves the parameters in memory, but as they are now outside the stack, they will get overwritten on the next PUSH.

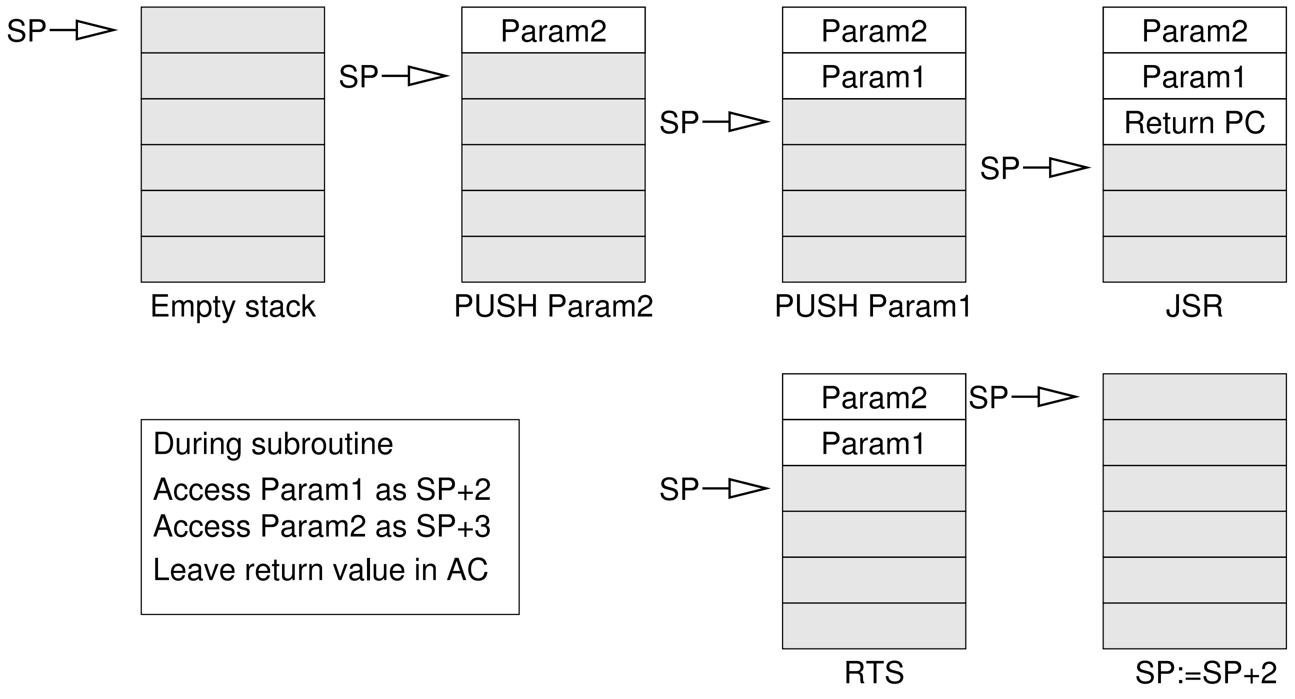


Figure 3.15: Passing parameters on the stack.