Processor Design

Manchester University has a long history of processor design; the world’s first computer (in the modern sense) was designed and built here in 1948. For details of this, and other early Manchester University computers see: www.computer50.org

At that time miniaturisation had not yet begun; the Small Scale Experimental Machine (SSEM) was built using valves and would fill a reasonably sized room (the same machine these days would fit on a pinhead!). However the number of logic gates available was small, so its architecture had to be simple. The SSEM was also experimental, so it was used as the basis of an evolving design which later became the Manchester Mark 1.

A replica of the SSEM now resides in Manchester Museum of Science and Industry.
Processor Design

The CPU is usually the most complex part of a computer system. All other systems depend on the CPU for control.

The design of the whole computer is heavily influenced by the architecture of its CPU.

The following lectures outline the detailed design of an implementation of a computer CPU. The instruction set is already defined.

- How do we perform the detailed logic design of a processor, given an outline block diagram and a specification of its architecture?

MU0

MU0 is an abstract machine based on the SSEM. It is a complete processor specification and is quite capable of running useful programs; it is also simple enough to describe a complete implementation down to gate level in a few lectures.
MU0 Instruction Set Architecture

A 16-bit machine
- 16-bit data
- 16-bit instructions
- 12-bit address space

Instruction format
- 4-bit instruction
- 12-bit operand address

MU0
MU0 is a simple model computer. Its architecture is (simplified from, but) similar to the very early Manchester machines, such as the Manchester Mark 1. When beginning to design a new computer the architecture is one of the first things to fix. It is necessary to define the programmer’s view of the system and the instructions which it will execute. The word length (i.e. the ‘width’ or number of bits in the datapath) and size of the address space are also fixed here. When designing a new processor all these issues must be resolved. The word length, addressing range etc. are influenced by cost and available technology. When these have been set the processor’s instruction set and number of internal registers are determined, usually using computer simulations to experiment with the performance of different possible architectures. This sets what is known as the Instruction Set Architecture (ISA). When the ISA is determined the processor can be implemented. This involves the design of the hardware architecture (often called the “microarchitecture”). Processors often go through many different implementations with the same basic ISA (although this changes and grows over time). The direct ancestors of many processors in use today (Pentium, ARM, Coldfire, …) first evolved in the early ’80s; newer implementations have yielded speed increases of >1000X.
MU0 Instruction Set

<table>
<thead>
<tr>
<th>F</th>
<th>Mnemonic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>LDA</td>
<td>Acc := [S]</td>
</tr>
<tr>
<td>1</td>
<td>STO</td>
<td>[S] := Acc</td>
</tr>
<tr>
<td>2</td>
<td>ADD</td>
<td>Acc := Acc + [S]</td>
</tr>
<tr>
<td>3</td>
<td>SUB</td>
<td>Acc := Acc − [S]</td>
</tr>
<tr>
<td>4</td>
<td>JMP</td>
<td>PC := S</td>
</tr>
<tr>
<td>5</td>
<td>JGE</td>
<td>If Acc &gt;= 0, PC := S</td>
</tr>
<tr>
<td>6</td>
<td>JNE</td>
<td>If Acc ≠ 0, PC := S</td>
</tr>
<tr>
<td>7</td>
<td>STP</td>
<td>Stop</td>
</tr>
</tbody>
</table>

Only eight of the sixteen possible operations are implemented. The others are “reserved for future expansion”.

In the case of MU0 the ISA is already fixed:
MU0 is a **16-bit** machine
  - Memory is 16 bits wide
  - The internal data paths are 16 bits wide
  - The instructions are 16 bits wide
  - The address space is 12 bits long (i.e. 4 Kwords)
The instructions are **fixed format**
  - 4 bits instruction
  - 12 bits operand address
It has two user visible registers
  - Accumulator (Acc) – the only ‘user’ register
  - Program Counter (PC)
It is a single address machine
  - One operand is specified in the instruction
  - Other operands (such as ACC) are implicit in the instruction

1. As shall be seen shortly there can be registers which are not directly accessible via the instruction set.
Instruction Execution Sequence

Like any CPU, MU0 goes through the three phases of execution: These are repeated indefinitely. In more detail ...

- a) Fetch Instruction from Memory [PC]
- b) PC = PC + 1
- c) Decode Instruction
- d) Get Operand(s) from:
  - Memory {LDA, ADD, SUB}
  - IR (S) {JMP, JGE, JNE}
  - Acc {STO, ADD, SUB}
- e) Perform Operation
- f) Write Result to:
  - Acc {LDA, ADD, SUB}
  - PC {JMP, JGE, JNE}
  - Memory {STO}

MU0 Programming

Programming example

MU0 can be used to write ‘real’ programs; however programming this type of processor can be very tedious! Below is an example of a program to total the numbers in a data table:

```
Loop LDA Total ; Accumulate total
Add_instr ADD Table ; Begin at head of table
STO Total ;
LDA Add_instr ; Change address
ADD One ; by modifying instruction!
STO Add_instr ;
LDA Count ; Count iterations
SUB One ; Count down to zero
STO Count ;
JGE Loop ; If >= 0 repeat
STP ; Halt execution
```

; Data definitions
```
Total DEFW 0 ; Total - initially zero
One DEFW 1 ; The number one
Count DEFW 4 ; Loop counter (loop 5x)
Table DEFW 39 ; The numbers to total ...
DEFW 25 ;
DEFW 4 ;
DEFW 98 ;
DEFW 17 ;
```

Note:
- Much shuttling of data to/from the accumulator (tedious & slow)
- Constants (e.g. “One”) need to be preloaded into memory
- **Self-modifying code** needed to **index** through the data table

In particular self-modifying code (where the program alters its own instructions) is normally deprecated.

**Exercise**

Rewrite this program using the ARM instruction set used in CS1031. Use registers as appropriate. (Your answer should be considerably shorter!)
The next stage is to produce an RTL datapath picture:

Having produced a sketch it is necessary to check to see that all the required operations are possible. It is possible to determine all the required ALU functions. The control (such as the decisions about whether to jump or not) is still being neglected at this point.

Datapath Design

Instructions can be compressed into two cycles of execution. In many cases each phase requires a memory cycle:

- Fetch Read instruction
- Decode/Execute Read operand/store accumulator

We (in some cases, you) can verify the validity of the datapath by testing the different instructions and seeing which buses are used in each cycle. (In this case all instructions are possible.) Try to fill in the data paths for the following instructions.
ADD

Fetch
Data Out Address Data In

Decode/Execute
Data Out Address Data In

Timing and Control

STO

Fetch
Data Out Address Data In

Decode/Execute
Data Out Address Data In

Timing and Control

JMP

Fetch
Data Out Address Data In

Decode/Execute
Data Out Address Data In

Timing and Control
Registers

In our MU0 there are three registers: ACC, PC, IR. Not all of these are visible to the programmer. We will make these from sets of 16 D-type flip-flops.

Note that all the control signals are common for the whole register.

Register Banks

The registers described here are the same as those previously described in the notes on RTL.

All flip-flops within a register have a common clock which is the system clock. All registers in the design will use this clock to ensure synchronous operation. Each flip-flop has an individual input. However, these can be shared across more than one register.

The loading of the register is controlled by a Clock-Enable (CE) signal; if this is active when the system is clocked the register will adopt the input value. By activating the CE signal at the correct time the register can copy (“latch”) the value.

Similarly the outputs may feed into a shared bus, providing only (at most) one output is enabled at once for each bus. This is controlled by the Output-Enable (OE) signal. By activating the OE signal at the right time the register can drive its output bus.
A modern processor will typically have more than one programmer-accessible register; a typical RISC (Reduced Instruction Set Computer) will have 16 (ARM) or 32 (MIPS) registers, any or all of which can be used to store temporary operands. These registers are normally grouped together in a register bank – also known as a "register file".

A register bank is similar to a memory, although its address size is much smaller; a register bank with 16 registers needs only a 4-bit register address ($2^4 = 16$). (In MU0 there is only one register (ACC) and so it can be addressed using zero bits ($2^0 = 1$)). However, unlike memory, it is common to be able to perform several operations on the register bank simultaneously; for example an ARM instruction might specify:

```
ADD R1, R2, R3
```

which requires two read operations and a write operation to be performed at the same time.

Any register can be connected to any port at any time, including, for example:-

```
ADD R1, R1, R1
```

How might this be implemented?
To call it an ALU in MU0 is rather an exaggeration. The instruction set does not provide facilities for performing \textit{logical} operations (e.g. NOT, AND, XOR etc.) and thus only an \textit{arithmetic} unit is required. An enhanced version of the machine could include logic operations which could easily be supported.

The easiest example of a microprocessor ALU to present here is that of the ARM, as used in COMP10031. The ARM is a 1980s architecture but is still in common use today\textsuperscript{1}.

An ALU is an RTL component; it is therefore irrelevant to us how many bits it processes. It will usually have two input buses (let’s call them X and Y) and a single output bus (Z). An adequate number of bits are supplied to specify the function performed on the inputs. In general an ALU will perform both \textit{arithmetic} and \textit{logical} functions. Arithmetic functions are typically addition/ subtraction/comparison treating the input buses as numbers. Logical functions are the now-familiar Boolean operations performed by pairing off the bits in the input buses.

A subset of the ARM ALU functions is given below:

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Description</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV</td>
<td>Move</td>
<td>B</td>
</tr>
<tr>
<td>MVN</td>
<td>Move inverted</td>
<td>\textit{not}(B)</td>
</tr>
<tr>
<td>ADD</td>
<td>Add</td>
<td>A+B</td>
</tr>
<tr>
<td>SUB</td>
<td>Subtract</td>
<td>A-B</td>
</tr>
<tr>
<td>RSB</td>
<td>Reverse Subtract</td>
<td>B-A</td>
</tr>
<tr>
<td>AND</td>
<td>Bitwise AND</td>
<td>A and B</td>
</tr>
<tr>
<td>EOR</td>
<td>Bitwise exclusive-OR</td>
<td>A xor B</td>
</tr>
<tr>
<td>ORR</td>
<td>Bitwise OR</td>
<td>A or B</td>
</tr>
<tr>
<td>BIC</td>
<td>Bit clear</td>
<td>A and \textit{not}(B)</td>
</tr>
</tbody>
</table>

1. If you own a mobile 'phone you probably carry one around with you!
MU0 ALU

MU0’s ALU must be capable of doing the following:--

- \( Z = X + Y \) (for the **ADD** instruction).
- \( Z = X - Y \) (for the **SUB** instruction).
- \( Z = X + 1 \) (to allow \( PC \) to be incremented after an instruction fetch).
- \( Z = Y \) (for the **LDA** instruction & to allow the S-field of \( IR \) to be sent to the PC for **JMP** etc.). Other operations might prove useful in an enhanced version of the machine.

Each of the operations can be expressed as an **addition**:

\[
X + Y \quad X + (-Y) \quad X + 1 \quad 0 + Y
\]

Note that these functions are directly user accessible. The ALU may also provide other functions within the processor which are used for internal operations. An example above could be a ‘move’ from the ‘A’ bus.

The MU0 ALU is much simpler; it does not provide logical functions at all! However there are more functions than just the ADD/SUB visible in the instruction set.

Later we will look at how the MU0 ALU can be extended to include some of these and some other functions.
Clearly some form of adder is required inside the ALU! The 16-bit architecture of MU0 requires a 16-bit ALU …… and hence a 16-bit adder.

One way of providing this is to use a Ripple Carry Adder.
- Simple just a string of full adders
- Slow long critical path

Adders

There are many ways to build a single bit adder; two of these are shown below.

The first design (which should, by now, be familiar) comprises two half adders joined into a full adder. The second design is the result of minimising the function and is a ‘direct’ approach to the logic. Although the first design is slower as a single bit adder (count the gates in the worst case path) the designs are comparable when used in larger adders because the critical path is the carry propagation and the path from Cin to Cout is 2 gates in both designs. When several single bit adders are wires together the time for an addition is always dominated by the speed by which the carries can be generated. This is because, under certain circumstances (which?) the carry into the most significant bit depends on the data into the least significant bit, which is many gates away.
Although the ALU is based on an adder this is not all that it does. The input buses have some preconditioning function applied first.
- The X bus can be zeroed
- The Y bus can be zeroed or inverted

All the necessary functions can be supported by additions, providing the input buses are conditioned as follows:

<table>
<thead>
<tr>
<th>Function</th>
<th>X' bus</th>
<th>Y' bus</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>X</td>
<td>Y</td>
</tr>
<tr>
<td>SUB</td>
<td>X</td>
<td>-Y</td>
</tr>
<tr>
<td>INC</td>
<td>X</td>
<td>1</td>
</tr>
<tr>
<td>=Y</td>
<td>0</td>
<td>Y</td>
</tr>
</tbody>
</table>

Note that, for example “X-Y” is now expressed as “X+(-Y)”.

Some of these functions are relatively easy; others are harder. For example producing a value of one on the Y input is awkward if only because the bit values are dissimilar (i.e. binary 0000000000000001).

However a general purpose adder also has a carry input to the least significant bit. If we consider this, things become easier:

<table>
<thead>
<tr>
<th>Function</th>
<th>X' bus</th>
<th>Y' bus</th>
<th>Carry in</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>X</td>
<td>Y</td>
<td>0</td>
</tr>
<tr>
<td>SUB</td>
<td>X</td>
<td>not(Y)</td>
<td>1</td>
</tr>
<tr>
<td>INC</td>
<td>X</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>=Y</td>
<td>0</td>
<td>Y</td>
<td>0</td>
</tr>
</tbody>
</table>
General ALU’s

In a more general ALU it is often useful to be able to provide:

- True data - the data as supplied
- Complement data - the data with all bits inverted (NOT)
- Zero all data bits zero
- One all data bits one

The carry in is a single bit value which adopts the appropriate value. Now the input buses are transformed by bitwise operations (the \(-Y\) of the previous table has gone too).

Note the transformation: \(X - Y = X + (-Y) = X + (Y + 1) = X + Y + 1\)

Exercise

Prove this to yourself by working through the following examples.

<table>
<thead>
<tr>
<th>Evaluate ⇒</th>
<th>4 − 2</th>
<th>5 − 4</th>
<th>3 − 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>X (binary)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Y (binary)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>\textit{not}(Y)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X + \textit{not}(Y)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X + \textit{not}(Y) + 1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

In each case note that a “carry out” is generated (and ignored).
Operand Select Logic

The inputs to the 16-bit adder are \( X'[15:0] \) and \( Y'[15:0] \). The outputs are \( Z_{15} - Z_{0} \).

module xprecon(sx, x, xprime);
output [15:0] xprime;
input [15:0] x;
input sx;
assign xprime = sx' ? x : 0;
endmodule

module yprecon(sy, siy, y, yprime);
output [15:0] yprime;
input [15:0] y;
input sy, siy;
assign yprime = (sy' ? (siy' ? y : ~y) : 16'h0001);
endmodule

Alternative select logic could be as shown below:

What would the Verilog code look like to produce these on synthesis?

Operand Selection

In the generic case it is useful to be able to pass, invert, clear or set a particular bit. This function is often known as True/Complement, Zeros/Ones (T/C 0/1).

There are many methods of implementing this – a common one is shown below.

Enter the control settings for the unit in the table below:

<table>
<thead>
<tr>
<th>Function</th>
<th>C1</th>
<th>C0</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Complement</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero</td>
<td></td>
<td></td>
</tr>
<tr>
<td>One</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

What type of operation(s) might require an “all ones” output?
ALU Function Decoder

The three function control bits have been assigned in a way that allows relatively easy decoding in the ALU.

<table>
<thead>
<tr>
<th>M1 M0</th>
<th>SX SY SIY Cin</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1</td>
<td>1 1 0 0</td>
<td>X + Y</td>
</tr>
<tr>
<td>1 1</td>
<td>1 0 1 1</td>
<td>X − Y</td>
</tr>
<tr>
<td>1 0</td>
<td>1 0 0 1</td>
<td>X + 1</td>
</tr>
<tr>
<td>0 0</td>
<td>0 1 0 0</td>
<td>Y</td>
</tr>
</tbody>
</table>

- The choice of coding is **arbitrary**.
- There are 24 possible choices.
- Any choice can be decoded.
- However, some choices simplify the logic.
- Finding a good solution takes intuition & practice.

**Function Decoder Design**

The ALU control bits could be generated directly from the decoder. However, providing that it is inexpensive, it is sensible to compress the function code into the fewest possible bits. In this design we require four ALU functions, so this requires a 2-bit function code. The mapping of an ALU function code to the ALU function is quite arbitrary; however, sensible choices can simplify the logic design, as the two assignments below attempt to show.

### Arbitrary order

<table>
<thead>
<tr>
<th>SX</th>
<th>SY</th>
<th>SIY</th>
<th>Cin</th>
<th>M1</th>
<th>M0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

### Less arbitrary order

<table>
<thead>
<tr>
<th>SX</th>
<th>SY</th>
<th>SIY</th>
<th>Cin</th>
<th>M1</th>
<th>M0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

The simple inspection can reveal some of the optimisations. For example, SY and Cin are the inverse of each other and split 50/50 between “0”s and “1”s; matching these to one of the input bit therefore gives half the decoder outputs for the price of one inverter …

We will choose the right hand side code because it produces simpler logic.
Controlling the processor

- We have considered the design of most of the datapath of the very simple computer, MU0.
- We now consider the logic to control the sequence of actions necessary in the execution of an instruction.

Note all registers have clock and enable signals and the multiplexers have select lines. Also the ALU has function select inputs. These all need to be provided by the timing and control circuit.

Processor Control and Sequencing

Using our generic registers, each register has two signals, CE and CLK. We need not consider the clock (CLK) here because it is distributed to all registers in the same way. We also have a two bit code to generate to specify the ALU function. There are also some signals to control the memory which are not shown explicitly. For control purposes the memory can be regarded as just another latch which can be told to store or output a value. Which value it stores/outputs is controlled by the address, so this is not a control problem. Note that the Instruction Register (IR) is always enabled. We have added tristate buffers to control its access to the “Y” bus. The lower 12 bits are 'padded' with four zero bits whereas the top four bits (the S field) are fed into the Timing/Control unit.

Summary of Control Signals Required

- Address source control Asel
- Clock enables for registers AccEn, PCEn, IREn
- ALU operation selectors M[1:0]
- X-bus source Xsel
- Y-bus source Ysel
- Memory control signals Wen, Ren

The control signals have been given (reasonably appropriate) abbreviated names.
Status & Decisions

The ability to make decisions according to their calculated state distinguishes computers from simple automata. A computer is capable of the action:

\[
\text{IF } \text{<condition> THEN } \text{<something>}
\]

Although it may be hidden beneath layers of language syntax in almost all processors this is implemented as a conditional branch.

This diverts the processor to code (program) in another part of the memory IF its condition is fulfilled.

Flags

MU0 evaluates conditions purely on the state of its accumulator. Some other processors work in this way, using a ‘condition’ evaluated and stored in a register. Others, such as ARM, evaluate and store the results of comparisons in a separate condition code register. These results are usually known as “flags” and typically represent the result of the last ALU operation. They are independent of the destination of the result and, indeed, it is usually possible to affect the flags without any other destination. For example the “CMP” (CoMPare) operation will perform a subtraction but throws the result away.

Common flags include:

- Sign (Negative)
- Zero
- Carry
- Overflow
- Parity

The ARM contains the first four of these.
Conditional Branching

MU0 has two conditional branches:
  - JGE Jump if Acc is positive
  - JNE Jump if Acc is not zero
    which test properties of the Accumulator
  - Positive if the most significant bit is zero
  - Non-zero if any bit is a “1” (OR of all bits)
    NB In Verilog can be performed using the reduction form of the logic operator e.g.
    \[ Z \leftarrow |\text{Acc}; //\text{performs bit by bit OR on Acc from left to right} \]
    If a specific test is required an operation such as SUB can be performed first to ‘compare’ with a known value.

Sign
A copy of the most significant bit of the result. Set for a (two’s complement) negative result.

Zero
Set if the result was zero, otherwise clear.

Carry
Used to store the carry out of the most significant bit of the adder; in a 32-bit processor this would be the 33rd bit of the result of an addition. If the addition was two unsigned numbers the carry will be set if the result was too big to represent in the word length. The carry can also be used as an input to further additions, thus a 32-bit processor can perform a 64-bit addition by adding the two lower (less significant) words and then adding the higher (more significant) words together with the carry. Subtraction can also be done following similar rules.

Overflow
The overflow flag will be set if a two’s complement operation produced a result which was not representable, for example if adding two numbers produced an answer so large that the sign bit was set producing an (apparently) negative result. Note that this applies to signed numbers only; the carry flag performs a similar function. The CPU is not aware of whether the programmer thinks numbers are signed or not. It therefore will evaluate both carry and overflow and allow the program to use one or the other.

Parity
Every word will have a number of “0” and a number of “1” bits. If a word has an even number of “1”’s it is said to have even parity; if not is has odd parity. One or other of these states is sometimes indicated by the state of a flag bit.
Parity is primarily used for detecting errors in transmitted data where a bit may have been corrupted (“dropped”); any single bit change in a word changes its parity.
Possible Control Sequences

All the possible instruction execution sequences are summarised in the slide, opposite. A key to the meaning of the various functions is given here.

This picture can be regarded as a state diagram, although it contains more information.

All instructions (except STP) execute in two (clock) cycles: the first fetches the instruction and increments the PC, the second decodes and executes the instruction itself. This leads to a very simple, two state FSM. Let’s label these states “fetch” and “execute”.

If the processor is in the “fetch” state it performs an instruction fetch (a memory read from the address in the PC with the data being placed in IR); it also increments the PC so the next instruction will be fetched from a different address.

It does this irrespective of what might happen next. (If the instruction is a JMP the PC increment will be wasted, but the processor doesn’t know that yet!). It then moves, inevitably, to the “execute” state.

When the processor is in the “execute” state its behaviour is influenced by the “F” field of the fetched instruction; it follows one of eight possible paths. Unless it has encountered an “STP” it will then return to fetching the next instruction.

When executing instructions other than STP the control signals are all derived from the “F” field with the exception of the PC enable. Here this may be influenced by the contents of the Accumulator to allow conditional branches. Notice that all branches behave in the same way except for the decision to latch the new PC value or not; this will simplify the logic by reducing the number of cases which need to be designed.

This picture can be translated into a state transition table which includes all the control signals.
## State Transition Table

<table>
<thead>
<tr>
<th>state</th>
<th>F[2:0]</th>
<th>Next state</th>
<th>IRe</th>
<th>PCEn</th>
<th>AccEn</th>
<th>M[1:0]</th>
<th>Xsel</th>
<th>Ysel</th>
<th>Asel</th>
<th>Ren</th>
<th>Wen</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>xxx</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>10</td>
<td>0</td>
<td>x</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>000</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>00</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>001</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>xx</td>
<td>1</td>
<td>x</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>010</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>01</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>011</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>11</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>00</td>
<td>0</td>
<td>0</td>
<td>x</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>101</td>
<td>0</td>
<td>0</td>
<td>N</td>
<td>0</td>
<td>0</td>
<td>00</td>
<td>0</td>
<td>0</td>
<td>x</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>110</td>
<td>0</td>
<td>0</td>
<td>Z</td>
<td>0</td>
<td>0</td>
<td>00</td>
<td>0</td>
<td>0</td>
<td>x</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>111</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>xx</td>
<td>0</td>
<td>x</td>
<td>x</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
FSM Implementation
Firstly define the combinatorial control logic.

```
always @ (state, pc, ir)
if (state == 0)
beg
   Asel = 0; // sel pc
   :
   Ren = 1;
   Wen = 0;
end
else 
   // state must be 1
   Ren = 0;
   Wen = 0;
   // now control depends on instruction
   case (ir[15:12])
      0: begin
         Ren = 1;
         // LDA
         .
         etc.
```

FSM State Transition Table

<table>
<thead>
<tr>
<th>state</th>
<th>F[2:0]</th>
<th>Next state</th>
<th>IREn</th>
<th>PCEn</th>
<th>AccEn</th>
<th>M[1:0]</th>
<th>Xsel</th>
<th>Ysel</th>
<th>Asel</th>
<th>Ren</th>
<th>Wen</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>xxx</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>10</td>
<td>0</td>
<td>x</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>000</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>00</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>001</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>xx</td>
<td>1</td>
<td>x</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>010</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>01</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>011</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>00</td>
<td>0</td>
<td>0</td>
<td>x</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>101</td>
<td>0</td>
<td>0</td>
<td>N</td>
<td>0</td>
<td>00</td>
<td>0</td>
<td>0</td>
<td>x</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>110</td>
<td>0</td>
<td>0</td>
<td>Z</td>
<td>0</td>
<td>00</td>
<td>0</td>
<td>0</td>
<td>x</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>111</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>xx</td>
<td>0</td>
<td>x</td>
<td>x</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Notes:
- N and Z are the Negative and Zero state of the Accumulator, respectively. (used to reduce the size of the table, as drawn)
- If a value is not going to be latched it doesn’t matter what it is! (e.g. ALU output for STO)
- STP operates by remaining in its evaluation state.

Observations:
- Many control bits are trivial to derive (e.g. IREn = State)
- “Don’t cares” give added freedom (e.g. Asel = State, YDin = F[2])
- In conditional jumps (JGE, JNE) the jump target is always available (for simplicity)
State Transition

always @ (posedge clk or posedge reset)
if (reset)
begin
    pc<= 12'h000; // set pc to zero
    state <= 0;   // start with fetch
end
else
    case (state)
    0: begin
        // fetch state
        Ren <= 1;
        ....
    end
    1: begin
        // decode/execute state
        case(ir[15:12]); // action depends on instruction
        0: begin
            Ren <= 1;
            ....
        end
        ....
    end ......

Instruction codes

<table>
<thead>
<tr>
<th>F</th>
<th>Mnemonic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>LDA S</td>
<td>ACC := [S]</td>
</tr>
<tr>
<td>1</td>
<td>STO S</td>
<td>[S] := ACC</td>
</tr>
<tr>
<td>2</td>
<td>ADD S</td>
<td>ACC := ACC + [S]</td>
</tr>
<tr>
<td>3</td>
<td>SUB S</td>
<td>ACC := ACC - [S]</td>
</tr>
<tr>
<td>4</td>
<td>JMP S</td>
<td>PC := S</td>
</tr>
<tr>
<td>5</td>
<td>JGE S</td>
<td>If ACC &gt;= 0, PC := S</td>
</tr>
<tr>
<td>6</td>
<td>JNE S</td>
<td>If ACC != 0, PC := S</td>
</tr>
<tr>
<td>7</td>
<td>STOP S</td>
<td>Stop</td>
</tr>
</tbody>
</table>

ALU functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>X + Y</td>
<td>01</td>
</tr>
<tr>
<td>X - Y</td>
<td>11</td>
</tr>
<tr>
<td>X + 1</td>
<td>10</td>
</tr>
<tr>
<td>Y</td>
<td>00</td>
</tr>
</tbody>
</table>
Timing

An important aspect of this design is that it is fully **synchronous**. All state changes happen ‘at the same time’ in:

- Registers {PC, Acc, IR}
- The controlling FSM
- Memory – more of which in a later lecture

No state changes at any times other than an active clock edge. For example the various control signals begin to be calculated when the IR is latched and have a complete clock period to settle before they are used. This allowed the analysis of the system in a **static** manner.

**Assumptions**

- The clock distribution is good enough that the signal arrives ‘simultaneously’ at every flip-flop
- The clock is slow enough to accommodate the slowest possible set of logic changes

---

**Timing**

Our MU0 implementation fetches, decodes and executes instructions at a rate of two clock cycles per instruction (2 CPI). The majority of these cycles include a memory operation.

The clock is a regular square wave; its period is set by the **worst case critical path**. In order to find this the operation of each cycle should be examined. Some examples are given below, although only the major operations {memory, ALU} are accounted for – bus switching times are ignored for simplicity. The time taken to decode the instruction is also neglected here because the instruction set is so small/simple; note that in a ‘real’ modern machine this is definitely not the case!

**Instruction fetch**

A memory cycle is performed with the result routed to IR. An ALU cycle is also performed, but this is in **parallel** with the memory operation. The critical path for this cycle will be whichever is the longer time.

**ADD execution**

This clearly requires an ALU operation (the addition), but it first requires one of its operands to be fetched from memory. The critical path is therefore the sum of the memory and the ALU cycle times.

**STO execution**

Only a memory cycle is performed.

**JMP execution**

The S field of the IR is transferred (via the ALU) to the PC; this can be counted as an ALU operation. The memory is not used here.

From this (incomplete) analysis it appears that the critical path is the sum of the memory and ALU cycle times. This would be used to set the clock period. The unused time in other operations would be wasted.
Optimisations

How do we make our computer go faster?
- Improve the technology
  Make smaller transistors and put more on a chip
- Improve the implementation
  Speed up the clock by shrinking the critical path
- Change the architecture
  Restructure the design to do more in a given period

This leads to:
- Faster clock
- Fewer clocks per instruction (CPI)

The following slides introduce some examples of these techniques.

Optimisations

Technology
Since the introduction of integrated circuits (circa 1970) the size of the manufactured features (transistors, wires, etc.) has been shrinking steadily. Reduced feature size leads to larger number of components on a device and faster operation of those components.

The (empirically derived) Moore’s Law observes that the number of components available (and the overall processing speed available) approximately doubles every 18 months. This is equivalent to a 10x improvement every 5 years, or about 1 000 000 improvement from 1970 to 2000.

Implementation
A computer engineer has little influence on where the technology leads. However it is important both to exploit the available technology and to design efficient circuits with short critical paths. The implementation will specify such things as the type of latches, flip-flops and registers used, the internal design of the ALU, the type of multiplexers etc.

Note also that implementations of a given function (such as a processor instruction set) may be optimised towards different goals. For example a high-speed implementation may be different from a low-power one.

Architecture
The architecture of the computer is where the designer has, perhaps, the most impact in its success. The architecture includes all aspects of the hardware from the instruction set design to the RTL layout of the blocks.

At RTL – which is the aspect which concerns us most here – the objective is to achieve maximum unit occupancy so that all parts of the system are kept as busy as possible. This is often achieved through parallelism. An example of parallelism already introduced here is the MU0 instruction fetch operation, where the PC is used as a memory address and is incremented at the same time. These operations could be done in series, but then two cycles would be required for every instruction fetch, considerably slowing the processor’s operation.

In general adding parallelism increases performance. Often, however, extra resources (buses, multiplexers, registers, functional units, …) are required and the cost can outweigh the benefit.
Reducing CPI

One method of speeding up the processor is to reduce the average number of clocks per instruction. There are improvement opportunities in (for example) JMP …

Note:
- Uses existing datapaths
- Requires more complex control/sequencing
- Requires an additional ALU operation ($Y + 1$)

Reducing CPI

In a simple processor such as MU0 there are not many methods of speeding up the design. However there are some …

For example the memory is not used when executing a JMP.

![Diagram showing the process of Old Fetch and Jump](image-url)
This reduces the number of cycles for a JMP instruction to one, thus reducing the average number of CPI. Similar optimisations can be applied to conditional jumps, with different behaviours depending on whether the jump is/is not taken.

The disadvantage of this method is that it makes the control and sequencing more complicated. Also note that another ALU operation is required: the value sent to the PC is not the JMP destination (which is already being used) – it is the following address – thus an extra increment is required using the other ALU input. (The ‘move’ operation is still needed for LDA.)

Note that this particular modification can be made solely by changing the control logic; the RTL picture is the same as before.
In our MU0 the critical path includes both the memory and the ALU. Adding a register (Din) can break the critical path (roughly) in half.

- The cost is an extra clock cycle.
- The benefit is that the clock can be (nearly) twice as fast.
An ADD takes three cycles (instead of two) at ~twice the speed.

**MU0 Timing Analysis**

As a simplification let’s assume that the only blocks that impose significant delays are the memory and the ALU. This is a reasonable approximation for this design. Furthermore let us apply some values to these, say:

- The memory (read or write) takes 10 ns
- The ALU (any function) takes 8 ns

By examining all the possible cycles the critical path, and hence the clock speed, can be determined.

- Instruction fetch uses memory and ALU in parallel, therefore requires 10 ns
- LDA/ADD/SUB use memory and ALU in series, requiring a total of 18 ns
- STO uses only memory and so requires 10 ns
- JMP uses only ALU (original architecture) for a critical path of 8 ns

The ‘improved’ architecture has a parallel memory cycle, increasing this to 10 ns
- JGE/JNE are analogous to JMP, or faster if the jump is not taken

The worst case is therefore 18 ns, which sets the clock period; (the frequency would therefore be about 56 MHz). Executing an ADD instruction requires two clock cycles or 36 ns (fetch, then execute) as the clock is a constant frequency.

Note that a lot of time is wasted in some cycles!
A Faster Implementation

If an extra latch (Din) was added to the RTL design adjacent to the IR the operand read from memory could be stored temporarily on its way to the ALU (see slide). This would require an extra cycle to execute an ADD operation {instruction fetch, operand fetch (to Din), ADD (from Din)} which doesn’t sound sensible in terms of accelerating execution!

However the critical path is now reduced to the memory cycle time (10ns) so the clock can run faster (100 MHz). The ADD operation as a whole can therefore complete in 30ns – a 17% speed up. Furthermore not all operations require operands from memory, so a STO (for example) could still be done in two cycles or 20ns – nearly twice as fast as before!

Although not possible with the existing architecture it would be possible to execute a LDA in two cycles too; what added buses/multiplexers would be required?

The disadvantage with such modifications is, of course, added complexity (and, hence, development cost).

Reducing Execution Time

Although it is beneficial to make instructions go faster what a user wants is for a program to go faster; this is not quite the same thing. The program is made up of instructions in some mixture; for instance LDAs might be more common than JMPs (or vice versa).

In reducing execution time it is therefore more important to optimise the more common operations.

In this context “more common” means those encountered most dynamically (i.e. as the program runs) rather than statically (counting through a program listing). This, of course varies considerably, depending on the application program …
Implementation

Consider the full adder:

<table>
<thead>
<tr>
<th>$C_{in}$</th>
<th>$A$</th>
<th>$B$</th>
<th>$C_{out}$</th>
<th>$S$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

This can be rewritten as:

<table>
<thead>
<tr>
<th>$A$</th>
<th>$B$</th>
<th>$C_{out}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>$C_{in}$</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>$C_{in}$</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>$C_{in}$</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>$C_{in}$</td>
</tr>
</tbody>
</table>

The carry output is either a copy of the carry input or it is independent of the carry input (and therefore available at once).

Consider a two bit adder:

<table>
<thead>
<tr>
<th>$A_1$</th>
<th>$B_1$</th>
<th>$A_0$</th>
<th>$B_0$</th>
<th>$C_{out}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>$C_{in}$</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>$C_{in}$</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>$C_{in}$</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>$C_{in}$</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

ie

$$C_{out} = C_{in} \text{ if } (A_1 \neq B_1) \text{ AND } (A_0 \neq B_0)$$

Note here that $Cout$ can ‘ripple’ across two bit positions at a time.
Look-Ahead Carry

(Almost) as soon as the A and B inputs arrive it can be predicted that Cout will be:

- Zero (carry “killed”)
- Cin (carry “propagated”)
- One (Carry “generated”)

This can be extended across more than one bit (see notes).

This scheme is called **carry look-ahead**.

Furthermore this reasoning can be applied recursively to bigger and bigger blocks …

Verify that, in this circuit, the *maximum* logic depth is six ‘blocks’.

What would the maximum depth be for a 16-bit adder?

As the carry has fewer logic blocks (therefore gates) to negotiate it can reach the more significant bits more rapidly (the benefits improve as the width increases).

This reduces the critical path and makes the whole adder *faster*…
Parallelism

Parallelism involves executing more than one instruction at once.
It is not the purpose of this course to discuss parallel computer architectures.
Sufficient to say that there are two, possible means of achieving parallel execution.

- Starting one instruction before the previous one is complete ("pipelining")
- Starting several instructions at the same time ("superscalar" & multi-processor)

Parallelism

Parallelism is something exploited extensively and at all levels in hardware design. For example adding carry look-ahead logic increases the number of gates in the adder but these decrease the overall addition time because more gates are switching in parallel.

We have also applied parallelism at RTL by incrementing the PC in parallel with the instruction fetch and, later, fetching an instruction while still executing a JMP.

However usually the word “parallelism” is applied more explicitly at the architectural level. A naive example of this would be using two complete processors to go “twice as fast”. This may work if we have two independent tasks but, in general, we haven’t. A system with twice the cost would therefore give less than twice the performance.

Much of the art of designing parallel systems is finding a ‘sensible’ balance between the hardware investment and the performance return. Two commonly employed techniques are given below.
Parallelism cont.

Pipelining

A common analogue of a processor pipeline is the process of washing clothes. When several loads of washing the second load can go into the washing machine as the first load goes into the drier. This means that two loads can be at different stages of ‘processing’ at the same time – an example of parallelism.

This is a relatively ‘cheap’ solution if you were going to use both machines anyway. However notice that our MU0 uses the same hardware for both fetching and executing instructions (e.g. the same ALU increments the PC and ADDs the data) and could not be pipelined without adding extra hardware; it is more the ‘washer/drier’ solution!

Multiple issue

By adding extra hardware it is possible to execute more than one instruction at once. With two decoders and two ALUs two instructions may be fetched and decoded together. This, potentially, doubles the processor speed for roughly twice the hardware cost.

In practice things are not so simple because it is not always possible to issue two instructions concurrently; for instance if the ‘first’ instruction was a JMP then the other instruction would be wasted anyway. There can also be dependencies where the second instruction needs the result from the first and therefore has to wait (and hardware has to be added to detect this). Trying to issue two instructions at once therefore gives less than twice the speed at more than twice the cost. Nevertheless attempting to issue two, four, or even more instructions together is quite common in high-performance processors.

1. “Concurrently” – at the same time.
ALU Enhancements

- MU0 has simple instructions involving only **ADD** and **SUBTRACT** arithmetic operations.

- The range of operations could easily be expanded to include Boolean **logic** operations, **shift** operations and extra arithmetic operations such as **multiply**. (Division is too complicated for consideration on this course.)

- The ALU would require more control bits (e.g. M[3:0])

Such operations could use some of the spare instruction codes.

---

**ALU Enhancements**

**Bitwise Logic Operations**

These are Boolean operations applied to each of the bits of the two values presented to the ALU. Operand bits are paired with others at the same position (significance) in the words, hence the expression “bitwise” operation. Unlike addition, which propagates a carry, each set of bits is independent.

e.g. the AND operation, for all values of i between 0 and 15, would yield:

\[ Z_{bus_i} = X_{bus_i} \& Y_{bus_i} \]

For example an AND operation:

\[
\begin{align*}
0011 & 0101 1001 1110 \\
AND & 0101 0110 1111 0010 \\
yields & 0001 0100 1001 0010
\end{align*}
\]

Relatively simple changes to the ALU are needed to implement AND, OR, XOR etc. These are normally done by selecting different functions using a multiplexer.

**Signal Preconditioning**

It is usual to locate the logic functions after the input preconditioning T/C 0/1s. This allows (for example) operations such as the ARM’s Bit Clear (BIC) instruction.

\[ \text{Result} = A \ \& \ \neg(B) \]

This can also provide alternative codings for operations such as MOV, which may simplify the decode logic. For example in the initial MU0 design a move operation was coded as 0+Y; it could also be coded as 0 OR Y, -1 AND Y, etc. (remembering -1 = 1111111111111111).
Bit-Wise Logic Operations

These are easy to add to an ALU

A possible optimisation

The bit-wise XOR function could be implemented by disabling the carry between the 1-bit full adders. The “SUM” outputs will then be the XOR of the X and Y bits presented to the adder. Review the full adder circuit to see how this could be done.
Shift Operations

In decimal it is very easy to multiply or divide numbers by ten.

e.g. $123 \times 10 = 1230$

The above operation has **shifted** the input **left** by **one** place.

In binary it is very easy to multiply or divide numbers by two.

e.g. $11011010_2 \div 10_2 = 01101101_2$

The above operation has **shifted** the input **right** by **one** place.

---

**Shift Operations**

What are shift operations?

Shift operations are movements of the bits within a word. For example:

- **Shift left**, one place
  - A left shift, as shown above, moves all the bits to a *more* significant position; thus **left shifting** a number by one place is equivalent to **multiplication** by two. Similarly shifting left two places is multiplication by four (assuming no bits are ‘lost’ at the most significant end).
  - Contrariwise **shifting right** is equivalent to **dividing** by powers of two.
  - When shifting left it is normal to fill the ‘vacant’ position(s) in the least significant bit(s) with zero(s). When shifting right this rule can also be obeyed with the most significant bits; this will divide correctly (subject to remainders) for positive or unsigned numbers, but not for two’s complement negative numbers where shifting a zero in makes the number positive. To avoid this there are often two forms of right shift provided:
    - Logical shift right (LSR) – shift in zero
    - Arithmetic shift right (ASR) – shift in copies of the existing MSB
  - A couple of minutes running through a couple of examples of left and the various right shifts is probably time well spent!
Practical Shift Operations

In practice there are some limitations due to operands being finite:

- A left shift loses its Most Significant Bit (MSB)
  the answer will be ‘wrong’ if this bit was “1”
- A right shift loses its Least Significant Bit (LSB)
  the answer will be ‘wrong’ if this bit was “1”
  i.e. if the number was odd.

As well as losing a bit, a bit must be shifted in.

- Left shifts shift in zero
- Right shifts either shift in zero or copy the MSB

The latter case preserves the two’s complement sign bit
- A rotation (either way) shifts the ‘lost bit’ back in

When are they useful?
Primarily in multiplication and division algorithms. They are also used in graphics, cryptography, in fact lots of “bit fiddling” operations.

What else can I do?
Rotation is another common ‘shift’ operation. Rotating left and right is just like shifting except the bits that ‘fall off’ one end of the operation ‘wrap back’ onto the other end.

Barrel shifting
Many processors provide shift operations which move the bits only one place per instruction.
A “barrel” shifter is a device which shifts bits an arbitrary number of places.
The ARM processor used in CS1031 has a barrel shifter.
Implementing Shifts

It is usual to treat a shift as an ALU-type function. This can be a single operand (one place) shift or shift a specified number of places (two operands). A shift is merely a rearrangement of the bits; it requires no logic! A single place shift can be done purely with multiplexers.

Implementing Shifts

The slide shows part of the ‘middle’ of a one place left or right shifter. Clearly there are some ‘dangling’ at each end. The bit input shifted left into the LSB is always “0”. The bit shifted right into the MSB is a “0” for logical shift right, but for arithmetic shifts the MSB or sign bit is copied here as well as into bit 14.

Shifts of more than a single bit position are also possible. This is sometimes done by the control logic repeating a one-place shift the correct number of times, a solution which requires little extra hardware but takes several cycles to complete.

A true barrel shifter can shift any number of places in a single cycle. This requires considerable extra hardware because all the multiplexers get much larger (the wiring is more complex too) and so it exacts a significant cost.

1. In practice there are methods of implementing such multiplexers more cheaply, but they are still costly.
Shift Registers

A shift register is similar to a ‘normal’ register, which always inputs and outputs all its bits in parallel. A shift register has the added feature of being able to move bits in or out one at a time, typically by ‘shuffling’ all the bits one way so that one ‘falls off the end’ on each successive cycle. Of course an extra control bit is also needed to enable this function.

Although it would be possible to use one, shift operations are not normally implemented using shift registers; a shift register contains more circuitry than an ordinary register and is therefore more expensive. While not significant in our MU0 this would be a considerable overhead on all the registers in a processor like an ARM.

Shift registers are primarily used in interfacing to I/O devices; some examples will appear later in the course. They are occasionally useful in other tasks, such as multiplication and division,

\[
A <= \{C,A[15:1]\}; \quad //Shift right A \\
Q <= \{A[0],Q[15:1]\}; \quad //Shift right Q
\]
Multiplication

Multiplication is a more complex arithmetic operation than addition. Multiplication is repeated addition:
- To multiply two numbers, N and M, start with zero and add N to it M times

This works, but is very slow for big numbers. A short cut – long multiplication

\[
\begin{array}{c}
2 & 3 & 7 \\
\times & 8 & 6 & 3 \\
\hline
& 0 & 7 & 1 & 1 \\
1 & 4 & 2 & 2 \\
1 & 8 & 9 & 6 \\
\hline
2 & 0 & 4 & 5 & 3 & 1
\end{array}
\]

An algorithm

1. Start with zero in an accumulator
2. Make X the least significant digit of N
3. Multiply M by X – add the result to the accumulator
4. Multiply M by 10
5. Make X the next least significant digit of N
6. If unfinished then repeat from step 3.
7. Done

Points to note
- This loops three times (the number of digits in M) not 863 time (the number in M).
- Only multiplications by a single digit (or by ten) are required.
- Multiplying by 10 is trivial.
- Only an addition of two numbers (new partial product and accumulator) is needed in any one step.
- The result has more digits than either of the operands (in general as many digits as both the operands combined, or six in this case).

Binary multiplication
The same algorithm can be used for binary multiplication. The only differences are:
- The digits are single bits.
- Multiplication is only ever by 0 (easy) or 1 (also easy)
- the multiplication outcome is therefore either 0 or N.
- The accumulator is multiplied by 2 – this is a one place left shift (and trivial).

Note also that X can remain the least significant bit of M if M is right-shifted at each step.

The algorithm now becomes
1. Start with zero in an accumulator
2. Make X the least significant digit of N
3. Multiply M by X – add the result to the accumulator
4. Multiply M by 10
5. Make X the next least significant digit of N
6. If unfinished then repeat from step 3.
7. Done
Long Multiplication

This multiplication algorithm is (in general) much faster.

No doubt it is already familiar!

The same algorithm can be applied to binary numbers.

With binary digits the only multiplications needed are x0 and x1. This is easy since x0 gives 0 and x1 leaves the original value – so we need to add in 0 or the original multiplicand depending on the value of the digit we are looking at.

More Aspects of Multiplication

Termination
The loop described above iterates for every digit (bit) in the N operand. Thus a counter can be used to control and terminate the operation. However using the procedure described for the binary operation N is continually being divided by two (and integer division discards any fractions) and so will eventually become zero. This can be used to indicate completion because any subsequent cycles can only ever add zero to the total – and we might as well not bother. This is sometimes known as early termination because, in many cases, fewer cycles are performed than there are bits in the operands. This gives shorter multiplication times as well as easier control (no counter needed).

Modulo arithmetic
A general case addition of two 32-bit operands can require up to 33 bits to hold its result, because of the carry out. This is true whether the addition is unsigned or two’s complement. A general case multiplication of two 32-bit operands can require up to 64 bits to hold its result. Often such results are mapped back into a register (variable) of the same width as the operands. This results in modulo- arithmetic, where bits may be ‘lost’ off the right-hand end of the number.

In the examples above it is quite likely that the operations would be performed “modulo 32”. This is the same as dividing the result by $2^{32}$ and keeping the remainder hence the name.

Negative numbers
The multiplication algorithm described only works with positive or unsigned numbers. A simple extension to cope with signed numbers would be to convert the operands to positive numbers and then make the result negative if the operands had different signs. This is the normal method with decimal numbers. However if modulo arithmetic truncates the result to the same length as the operands then the algorithm will work anyway. Any potential errors occur in the high-order bits which are truncated.

(Try it!)
If the full answer is required this ‘trick’ can be exploited by first sign extending the operands to the full word length and truncating at the end.

1. Extending the number to the left with copies of the existing MSB, so the sign bit is preserved.
A Sequential Multiplier

The implementation of this multiplication algorithm may be done in software or hardware.

A software implementation may be coded simply by following the steps outlined in the notes.

A hardware implementation is shown here.

Other multipliers

The algorithm described is not the only way to build a multiplier. A number of other schemes employing the same basic ‘shift and add’ approach exist, but different operands may be shifted in different directions.

You may meet a different implementation in a reference book; however the principle will be the same.
Multiplier

This is a serial multiplier:

- A number of steps (clocks) are performed
- Only one adder is required

Simple FSM used for control.

Almost a processor datapath in itself!

Verilog listing for 8x8 bit multiplier

```verilog
//HDL COMP 10211 Multiply Example
//--------------------------------------
//RTL description of binary multiplier
//Block diagram in notes
//n = 8 to halt after all bits done
module mltp(S,CLK,Clr,Binput,Qinput,C,A,Q,P,Done);
input S,CLK,Clr;
input [7:0] Binput,Qinput;          //Data inputs
output C, Done;
output [7:0] A,Q;
output [3:0] P;
//System registers
reg C, Done;
reg [7:0] A,Q,B;
reg [3:0] P;
reg [1:0] pstate, nstate;           //control register
parameter T0=2'b00, T1=2'b01, T2=2'b10, T3=2'b11;
//Combinational circuit
wire Z;
assign Z = ~|P;                     //Check for zero
//State transition for control
//See state diagram in notes
always @(negedge CLK or negedge Clr)
    if (~Clr) pstate = T0;
    else pstate <= nstate;
always @(S or Z or pstate) //control register
    case (pstate)
        T0: if (S) nstate = T1;
        T1: nstate = T2;
        T2: nstate = T3;
        T3: if (Z) nstate = T0;
    endcase
always @(negedge CLK)   //Register transfer operations
    T0: B <= Binput;                //Input multiplicand
    T1: begin
        A <= 8'b00000000;            //Initialize counter to n=8
        C <= 1'b0;
        P <= 4'b1000;              //Initialize counter to n=8
        Q <= Qinput;              //Input multiplier
        Done <= 1'b0;            // Not Done
    end
    T2: begin
        P <= P - 4'b0001;          //Decrement counter
        if (Q[0])
            (C,A) <= A + B;           //Add multiplicand
        end
    T3: begin
        C <= 1'b0;                //Clear C
        if (Z) nstate = T2;   //Shift right A
    end
    endcase
always @(negedge CLK)
    Done <= Z;                    // Z = 1 when done
endmodule
```
// Testbench for HDL Multiply COMP 10211
//-----------------------
//Testing binary multiplier
module test_mltp;
//Inputs for multiplier
  reg S,CLK,Clr;
  reg [7:0] Binput,Qinput;
//Data for display
  wire C;
  wire [7:0] A,Q;
  wire [3:0] P;
  wire Done;
//Instantiate multiplier
  mltp mp(S,CLK,Clr,Binput,Qinput,C,A,Q,P,Done);
initial
  begin
    S=0; CLK=0; Clr=0;
    #5  S=1; Clr=1;
    Binput = 8'b00010111;
    Qinput = 8'b00010011;
    #15 S = 0;
  end
initial
  begin
    repeat (46)
      #5 CLK = ~CLK;
  end
endmodule
Processor Design – a Summary

• A processor can be quite simple to design
  – an entire processor can be described down to gate level in a few lectures

• A processor has a **datapath** which does the processing
  – an RTL (Register Transfer Level) design
  – many (16-, 32-, …) bits wide, but regular structures
  – the datapath may account for 90%+ of the gates
  – therefore it is designed and optimised first

• The datapath needs **control logic**
  – an FSM (Finite State Machine)
  – the control provides steering and timing for the datapath
  – relatively few gates, but more complex structures

• All CPUs are built this way
  – it's just that the instruction set gets bigger and the number of optimisations increases.

### Multiplication Exercises

Have a go at these:

0010 x 0010

0110 x 0101

0011 x 1110 (unsigned)

0011 x 1110 (signed)