COMP22111

Processor Microarchitecture

Lecture Notes

Part 4: Implementing Stump

(approx. 2 lectures)

Dr Paul Nutter

Email: p.nutter@manchester.ac.uk
Office: IT119 (Ground floor IT building)

Version 2014

About these notes

These notes form part of the teaching materials for the second year course unit COMP22111: Processor Microarchitecture (formerly known as VLSI Design, and COMP20241).

The module aims to give a view of the role of a digital hardware designer, taking an idea and implementing it as a silicon chip. A processor is a representative example of logic used in today's chips, also giving further insight into how computers actually *work*. Having completed the module you should have developed the confidence to be able to take a concept and realise it in hardware. You should also appreciate the test and verification processes involved so that your chips work efficiently and reliably ... first time, every time!

This module aims to develop the two key aspects of COMP12111, namely hardware design and microprocessors. COMP12111 gave an overview of the hardware development process; COMP22111 builds on these skills to introduce and exercise industrially relevant hardware skills with a design flow from concept to implementation. It used microprocessors as design examples to illustrate and reinforce how machine code, output from a compiler, is interpreted and executed by a computer.

The lectures support the laboratory exercises where you will complete a design for a 16-bit microprocessor, the STUMP, a RISC machine similar in concept to an ARM processor.

Organisation of these notes

These handouts contain copies of the lecture slides as well as "extra" information relevant to the material being taught. All material is examinable.

As well as providing information, I also aim to provide examples in the notes in order for you to gauge your own level of understanding; examples are clearly identified.

Q.

What is your name?

References

I acknowledge the following references that I have used to prepare these notes:

1. COMP22111 Lecture Notes (pre-2011) by Dr Linda Brackenbury, School of Computer Science, The University of Manchester.
2. "ARM System Architecture", S Furber, Addison Wesley, ISBN 0-201-40352-8
3. "Principles of Computer Hardware", A Clementa, 4th Edition, Oxford University Press, ISBN 978-0-19-927313-3.
4. "The Verilog hardware description language", Thomas & Moorby, 5th Edition, Springer, ISBN 978-0-387-84930-0.
5. "Digital VLSI Systems Design", S Ramachandran, Springer, ISBN 978-1-4020-5828-8, (ebook available to download via JRULM)

Assessment

This course unit is assessed by a formal examination and laboratory work. The breakdown of the assessment is

- exam – 55%
- lab – 45%

The examination lasts 2hrs and you must answer three questions out of the four questions provided. The questions are split evenly between the two halves of the course unit (2 from PWN, 2 from JDG).

This course was modified extensively for the 2011-2012 academic year, and is undergoing some updates for the 2013-2014 academic year. Hence, when looking at past papers (all of which are available of the University intranet) take care when looking at papers from earlier years. If the question looks unfamiliar, the chances are the subject matter is no longer covered. No sample answers are given for past papers. However, I will be happy to discuss any questions relating to the material I cover in lectures, providing some attempt to answer them has been made beforehand.

Course Website

Further information, such as copies of these notes, copies of the slides, corrections to the notes, etc, can be found on the course unit website

http://studentnet.cs.manchester.ac.uk/ugt/2014/COMP22111/

If anything is incorrect on the course website, or there is anything you would like to see there (apart from your exam questions ☺) then let me know.

Additional material can be found on the School of Computer Science Engineering Wiki:

http://wiki.cs.manchester.ac.uk/engineering/

If you're stuck ...

Then please come and see me … I'll be happy to answer any questions relating to the module (well, my course material certainly). Ask questions during the lecture or catch me at the end of lectures, or in the lab, or come and see me in my office. If possible, please email me to check my availability, that way I'll make sure I set aside enough time to answer your questions. If you knock on my door out of the blue, then I may not have the time to see you!

Further information, such as copies of these workbooks, copies of the slides, extra notes, corrections to the notes, etc, can be found on the course unit Blackboard site.

Finally, if you notice any mistakes, then please let me know … I'm not perfect and my notes definitely aren't … ☺.

Implementation of the Stump Processor

In this part of the course we will look at the implementation of the Stump Processor, starting from an RTL design.

- MU0 as an example
- Use of path usage diagrams
- Verilog implementation
- Verilog test bench

Notes

What will we be looking at?

We will be looking at the design of a processor with a load/store RISC architecture … the STUMP. First we will look at a general RISC processor and discusses the advantages of such an approach to processor design (Jim will cover CISC architectures in another lecture). We will then look at how pipelining can be used to improve the throughput of the RISC processor, and calculate the associated speed-up benefit of pipelining.

We will then look at the design of the STUMP processor starting with a specification and the development of an instruction set. We will look in-depth at the instruction set features and some example instructions. Following this we will work through an architectural design and we will use path usage diagrams to confirm that the architectural design is sufficient. We will look at the RTL design of the STUMP processor that you will build in the lab. Finally, we will look at the design of the control block using MU0 as a design example.

Instruction Set Architecture

The instruction set architecture (ISA) is the view the assembly language programmer sees of the processor architecture. The ISA includes the processor registers, along with address and data formats. On the other hand the microarchitecture includes the functioning parts of the processor and their interconnectivity.

Architectural Design

We now have a correct (tested) specification, now we have to move in to the architectural design.

Our model was a black box, we identified the interface, and what it does; we have no idea (yet) on how it will be implemented.

In the architectural design stage we identify functional blocks and the data paths between them.

The STUMP is a RISC processor so we can identify 4 functional blocks for handling data along with a control block.

The architectural diagram emerges from the description of these blocks.

Notes:

The STUMP is a RISC processor that has four functional blocks for handling data, plus a control block.

Register Bank

The *Register Bank* provides general purpose register storage and contains the Program Counter (R7). It receives data from *Execute Unit* or data from the *Data Interface* (load instruction).

Execute Unit

The *Execute Unit* takes operands from the *Register Bank*, or an immediate value from the *Data Interface*, and performs the arithmetic or logical operation specified to get a result and new status.

Data Interface

The *Data Interface* interfaces between the processor and external memory. It handles data to be written to memory (store instructions), it takes data read from memory and sends it to the *Execute Unit* (immediates), *Control* (instruction bits) or *Register Bank* (load instructions). This implies that the Instruction Register is in the *Data Interface*.

Address Interface

The *Address Interface* acts as the processor to memory interface for all memory addresses. It handles the addresses for instruction fetches and for load/store instructions.

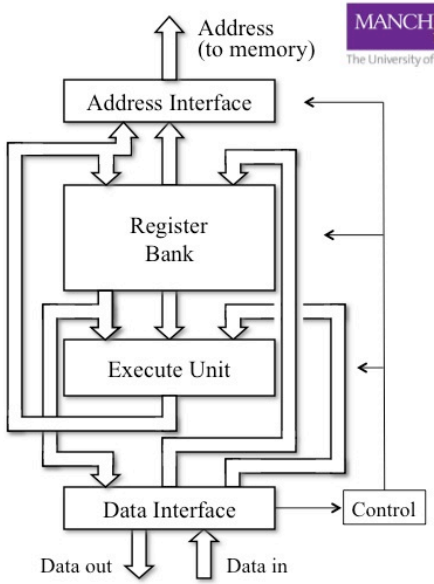
Control Block

The *Control Block* interprets instruction bits sent by the *Data Interface* and status bits from the *Execute Unit* to produce control signals to the four functional blocks. Control is about producing the correct control signals at the correct time so that the data flow and operations of the datapath: fetch, execute and writeback phases, produce results as specified by the instruction stream.

The above functional blocks are found in most processor designs, not just the STUMP we are designing.

From the above description the architectural diagram emerges.

STUMP Architecture



Shows all blocks on the datapath (not control) and the paths between the functional blocks.

This is one conceptual view.

Notes:

STUMP Architecture

We build up an architecture design knowing how data is passed around processor by the different types of instructions. We have identified the four functional blocks (we can treat control separately):

- the *Address Interface* – for interfacing to the address bus
- the *Register Bank* – containing the 8 systems registers (including the PC)
- the *Execute Unit* – for performing the required operation
- the *Data Interface* – for interfacing to the data bus.

As an example, consider a Type 1 instruction:

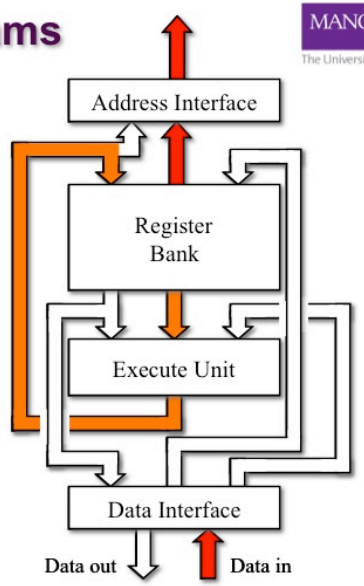
1. We need to output an address from the PC via the *Address Interface*
2. We read the instruction from the address via the *Data Interface*
3. The decoded instruction specifies the two source registers and the operands are passed from the appropriate registers in the *Register Bank* to the *Execute Unit*
4. The *Execute Unit* generates a result which is either stored to a destination register in the *Register Bank*, or to an address register in the *Address Interface* in the case of a load and store instruction, in which case an address has been generated
5. If we have a load instruction then the address specified in the *Address Interface* is passed to the address bus and the data contained in the memory location is loaded into the destination register in the *Register Bank*
6. If we have a store instruction then the contents of the destination register in the *Register Bank* is passed through the *Data Interface* and written to the address specified in the *Address Interface* .

The architectural diagram for the STUMP shows all the blocks on the datapath (i.e. not control) and the paths required between the functional data blocks. The diagram allows the designer to check that:

1. all paths between the blocks that are required are provided
2. that no path is used simultaneously for different purposes.

We do these checks using usage diagrams that show the use of paths in each phase and for each instruction type. Note sparse use of paths in the non-pipelined system (we will discuss pipelining later).

Path Usage Diagrams



Instruction fetch

We also update the PC to PC+1 in the fetch cycle

No overlapping of data paths so there should be no problem

Notes:

Usage Diagrams

The paths used in the STUMP architecture depend on the current phase and the type of instruction.

Fetch Phase

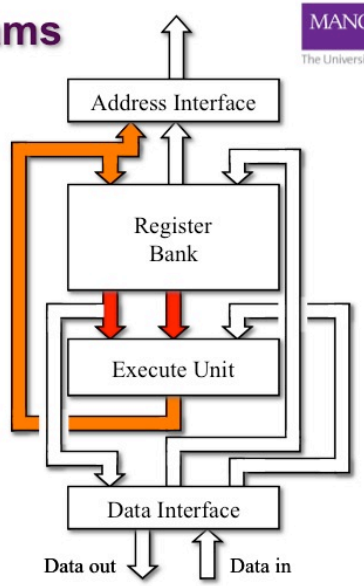
The paths used are common to all instructions in the instruction fetch phase.

The *Register Bank* provides the address of the current instruction (from the Program Counter) to the *Address Interface* and the *Data Interface* receives the instruction from memory and stores it in the **Instruction Register** (IR) located in the *Data Interface*.

The PC is passed through to the execute unit where it is incremented and written back to R7.

Once the instruction has been written to the IR it is decoded and the control determines the following behaviour of the processor, such as specifying the operation of the *Execute Unit*, or specifying to the *Register Bank* which register should be updated etc.

Path Usage Diagrams



Type 1 Execute

We need to writeback the result to a register

No overlapping of data paths so there should be no problem

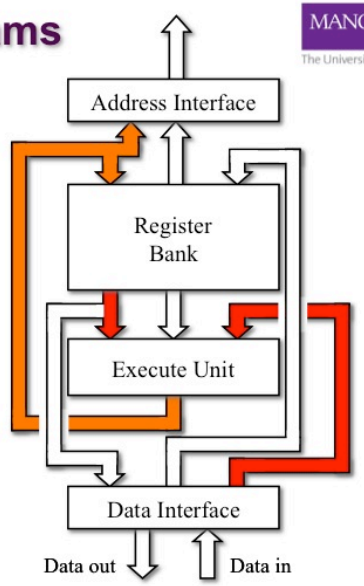
Notes:

Usage Diagrams

Type 1 Execute Phase

In the case of a Type 1 instruction the data to be operated on by the *Execute Unit* is taken directly from the *Register Bank* (Source A and Source B register contents). The result is written back to either the register bank, or to the address register in the address interface if the instruction is a load or store instruction.

Path Usage Diagrams



Type 2 Execute

We need to writeback the result to a register

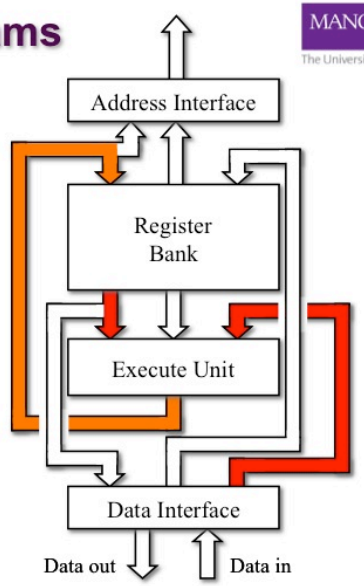
No overlapping of data paths so there should be no problem

Notes:

Type 2 Execute Phase

In the case of Type 2 instructions the data for the *Execute Unit* is taken from the *Register Bank* (Source A register contents) and the *Data Interface*, which supplies the immediate value directly from the instruction. The result is written back to either the register bank, or to the address register in the address interface if the instruction is a load or store instruction.

Path Usage Diagrams



Type 3 Execute

We need to writeback the result to a register

No overlapping of data paths so there should be no problem

Notes:

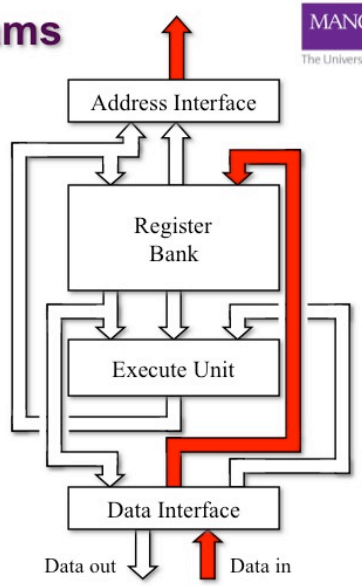
Type 3 Execute Phase

In the case of Type 3 instructions the operation is the same as Type 2, where the data for the *Execute Unit* is taken from the *Register Bank* (Source A register contents) and the *Data Interface*, which supplies the immediate value directly from the instruction.

However, recall that a Type 3 instruction is a conditional branch instruction to the result is written to R7 (the PC) if the branch is to be taken. In this case the address to jump to is calculated by adding the appropriate offset to the PC value (taking into account that the PC has been incremented previously).

Path Usage Diagrams

Load operation



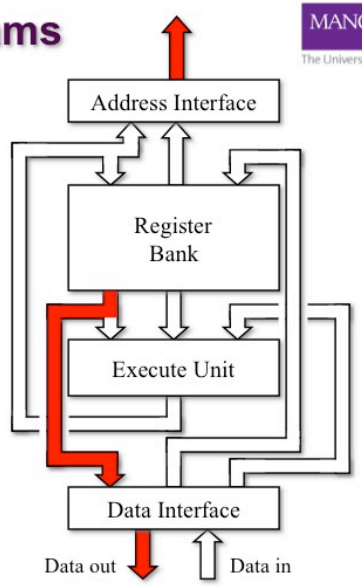
Notes:

Load Instruction Phase

The result from the *Execute unit* supplies the address to the *Address Interface,* which is stored in an address register in the *Address Interface*. In the next clock cycle (the 3rd state) the contents from the memory address stored in this register is read into the *Data Interface* and stored in the appropriate register in the *Register Bank* as identified by the destination register value in the instruction.

Path Usage Diagrams

Store operation



Notes:

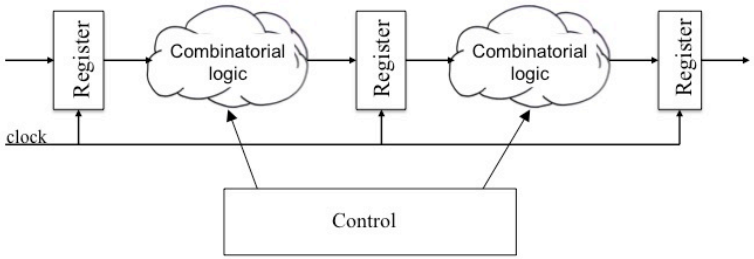
Store Instruction Phase

The result from the *Execute unit* supplies the address to the *Address Interface,* which is stored in an address register in the *Address Interface*. In the next clock cycle (the 3rd state) the contents of the register specified as the destination register in the instruction is passed to the *Data Interface* for writing to memory.

.

RTL Design

RTL describes the datapath in terms of registers and combinatorial logic and is the first level at which actual hardware is described.



Data flow is synchronised to a clock using registers and combinatorial logic operates on the stored data.

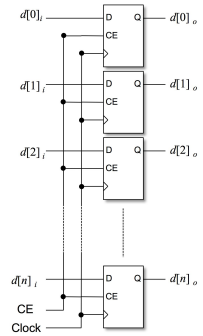
External control signals control the flow of data as well as the operations performed.

Notes:

Registers

Recap …

What is a register? A register is the name given to a number of flip-flops controlled by the same clock that hold together a single coherent value, such as a number, an instruction, etc.



A typical register is constructed from a number of flip-flops (D-types), which each have their own input and output bit but all have the same common clock input; this enforces synchronisation – all the bits switch at the same time.

A clock is normally free-running but it may not be desirable for the register not to change every cycle. Usually, therefore, there will be a clock enable (CE) input, which is also common to the flip-flops in one register. Different clock enables control different registers, whereas the clock will typically be the same signal in all the registers in the system.

More rarely, other inputs are also present. For example, there may be a 'clear' signal that sets all the flip-flops to a known value; this may be used for initialising the system. (It has not been shown here.)

Register Transfer Level

At the architectural level we write behavioural models of each block including the control. In a complex system, we would want to devise tests to test the individual blocks for correct operation. The block would then be put together to form the complete system and the system specification tests run on it. The behavioural models would be modified until the test results are identical to those from the system specification level. At this point you can start designing at the next level down – **Register Transfer Level**.

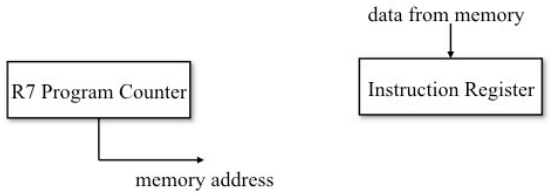
The simplicity of the STUMP design allows us to bypass the architectural level in the lab (there are also time constraints), however, in a more complex system you would be expected to complete the architectural level.

Register Transfer Level (RTL) describes the datapath in terms of registers with combinatorial logic blocks providing some required function between them.

The RTL level is the first level at which actual hardware is described. It is also the level at which any system clock first appears, so timing needs to be considered. One way to arrive at a datapath RTL design is to consider the hardware required to perform each phase and then combine diagrams, i.e. a divide and conquer approach.

RTL Design of the STUMP

Fetch phase



Notes:

RTL Design – Fetch Phase

For an instruction fetch the *Program Counter* (R7) sends the address of instruction to memory, the data is read from the memory address and is placed in the *Instruction Register* at the end of the phase.

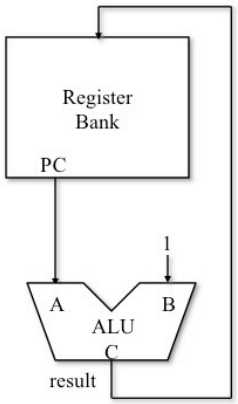
This is the same for all instructions since the data being read from memory and placed in the *Instruction Register* is the next instruction.

Here we would need two registers to hold the instruction (*Instruction Register*) and memory address (*Program Counter*).

RTL Design of the STUMP

MANCHESTER 1824
The University of Manchester

Fetch phase



Notes:

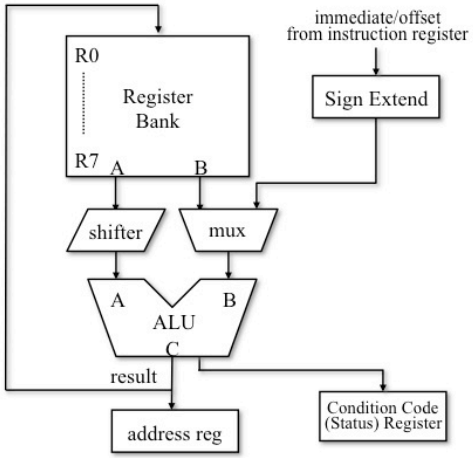
RTL Design – Fetch Phase

Don't forget, we also need to update the PC during the fetch operation. We could do this in some covert way within the *Register Bank*, however, we have an *ALU*, which not use it!

So also during the fetch phase we must copy the PC from the *Register Bank* into the *ALU* and add 1. The result must then be written back to the PC in the *Register Bank*. This happens at the same time that a memory read is being initiated to read the instruction from memory.

RTL Design of the STUMP

Execute phase



Notes:

RTL Design – Execute Phase

Execution depends on the type of instruction stored in the *Instruction Register*.

In the case of a Type 1 instruction, the inputs to the *ALU*, A and B, are two source operands from the *Register Bank*, with operand A shifted if required by the *Shifter*.

In the case of a Type 2 instruction, the inputs to the *ALU* come from the *Register Bank*, A, and the immediate value from the *Instruction Register*. The immediate value is 5-bits so it needs to be sign extended so that it is 16-bits as required for input B to the *ALU*.

In the case of a Type 3 instruction, the inputs to the *ALU* come from the *Program Counter* (R7) as input A to the *ALU* and the offset value from the *Instruction Register*. The offset value is 8-bits so it needs to be sign extended so that it is 16-bits as required for input B to the *ALU*.

A *MUX* is used to select between the register and immediate/offset value to input B of the *ALU*.

In all cases the result from the *ALU* operation is either written back to a register in the *Register Bank* or it is stored in an *Address Register* for load/store operations.

In the case of a branch instruction where the branch is taken, the *Program Counter* (R7) is updated with the value from the *Result Register*. If the branch is not taken, the value can still be written to the R7 except the *Register Bank* is not enabled for writing so R7 is not overwritten. Alternatively, for a branch that is not taken, the result can be written to R0, since R0 is hard wired to '0's so will have no effect.

The *Condition Code Register* should be updated where appropriate.

Q.

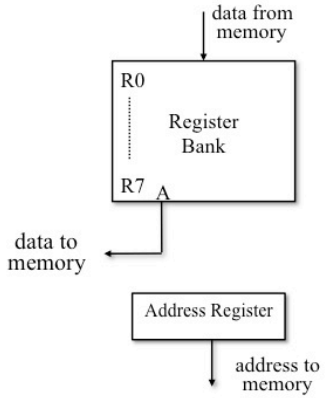
How do you sign extend a 5-bit 2's complement number to 16-bits?

Q.

What bit from the instruction register can be used to control the MUX at the input B in the ALU?

RTL Design of the STUMP

Load/Store phase



Notes:

RTL Design – Load/Store phase

In the case of load/store instructions an extra phase is required to perform the memory operation. The *Address Register* contains the memory address used to load/store data.

For a store, the data to be written to memory comes from the *Register Bank* and is written to the memory address given in the *Address Register*.

In the case of a load, the data to be loaded into the *Register Bank* is read from the memory address specified by the value in the *Address Register*.

2's Complement Revisited

How do you represent +ive and –ive values in 2's complement form?

Positive numbers are the same, no different from the normal binary representation. The negative representation can be found by taking the positive value, inverting all the bits (i.e. '0'⇒'1' and '1'⇒'0') and adding '1' to the result.

For example, -5 is given by taking +5 = 00101, inverting the bits to give 11010, and adding '1' to the result, i.e. 11011.

An alternative approach, that requires no addition process, is to copy the bits starting from the least significant bit (the rightmost bit) to the first '1', and then invert all the remaining bits.

Sign Extension

Remember, the most significant bit of a 2's complement number is the sign bit, which is '0' for a positive number and '1' for a negative number. In order to extend a two's complement number from say 5-bits to 16-bits, you simply copy the sign bit into the extra bits.

For example, if +5 = 00101, then extending this value to 16-bits will involve copying the sign bit, which is zero, to the most-significant bit of the number, i.e. **0000000000000**00101, for -5 = 11011, this would involve copying '1' into all the bits up to the most significant bit, i.e. **1111111111111**11011.

Q.

Sign extend the following 2's complement numbers.

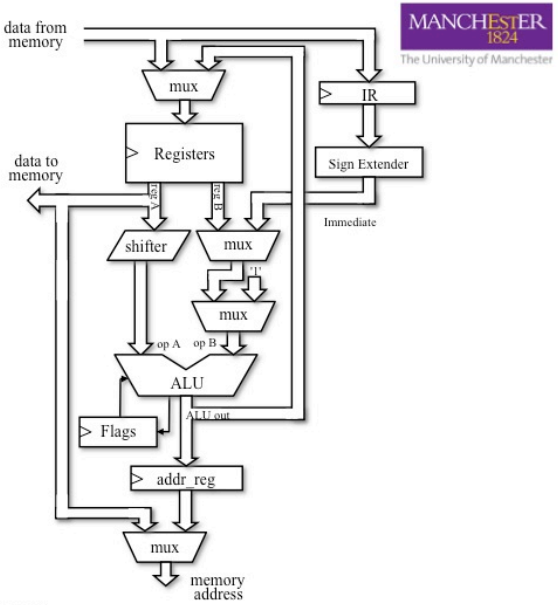
i) 01101 to 16 bits

ii) 101 to 8 bits

iii) 0000000000000001 to 16 bits

iv) 101011110000011 to 16 bits

RTL STUMP



Notes:

Full RTL Diagram for the STUMP

By combining the RTL diagrams of the three stages we arrive at an RTL design for the STUMP processor.

The only addition to the full RTL design is a multiplexer to output the address to memory. This switches between the *Program Counter* during instruction fetches and the *Result Register* during the writeback of a load/store operation.

Appendix C in the lab manual gives a full description of the RTL design for the STUMP, along with the control signals from the control block that control the operation of the data path. In addition, the signals required to interface the processor to the outside world, the interface, are also presented.

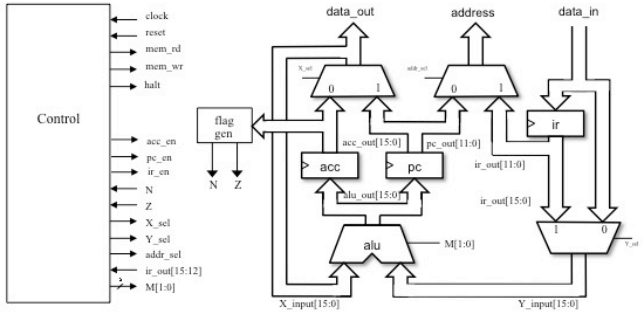
Implementing a Processor

MANCHESTER 1824
The University of Manchester

We will now look at implementing the control …

… we won't be doing the STUMP, as you'll be doing that in the lab!

Earlier we looked at MU0 and how to implement the datapath in structural Verilog – the datapath and control are shown below.



We have (largely) implemented the datapath, so now we will look at implementing the control block.

Notes:

MU0 Instructions

Recall, MU0 has the following instruction set:

LDA	S	ACC := mem[S]
STA	S	mem[S] := ACC
ADD	S	ACC := ACC + mem[S]
SUB	S	ACC := ACC – mem[S]
JMP	S	PC := S
JGE	S	if ACC ≥ 0 PC := S else PC remains unchanged
JNE	S	if ACC ≠ 0 PC := S else PC remains unchanged
STP		halt processor

The MU0 Control Signals

The MU0 datapath has the following control signals:

- addr_sel – address multiplexer select
- X_sel – X-bus multiplexer select
- Y_sel – Y-bus multiplexer select
- acc_en – accumulator register enable
- pc_en – program counter register enable
- ir_en instruction register enable
- M[1:0] – alu function select
- mem_wr – memory write enable
- mem_rd – memory read enable
- halt – halt signal to indicate the processor has halted

The following data from the datapath is used by the control block to make decisions:

- ir – the contents of the ir to determine the required operation
- N and Z flags for determining whether a branch should be taken (to control pc_en)

The alu operation is determined by the signal M which provides the following operations:

1. pass one of the values through, i.e. Y – M = 00
2. add the two inputs X and Y, i.e. X+Y – M = 01
3. add '1' to one input, i.e. X+1 – M = 10
4. subtract the two inputs X and Y, i.e. X-Y – M = 11

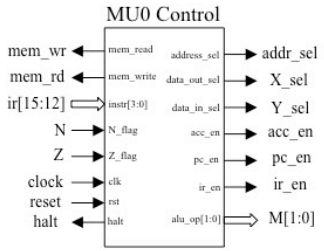
Why do we need these operations?

MU0 control block

The University of Manchester

The MU0 control block contains the MU0 fsm as well as the control logic that determines the status of the control signals.

We have defined the interface to the control block previously.



We could define the Verilog module header for the MU0 control block:

```
control MU0_control(input             clk, rst, N_flag, Z_flag,
                    input       [3:0] instr,
                    output  reg       mem_read, mem_write, halt,
                                      data_out_sel, data_in_sel,
                                      address_sel, acc_en,
                                      pc_en, ir_en,
                    output  reg [1:0] alu_op);
```


Notes:

The Control Block

The control logic is all about ensuring that the right things happen at the correct time by activating control signals at the appropriate time in the instruction cycle.

The control block is instantiated within the MU0 module as:

```
control MU0_control(.clk(clock),
                    .rst(reset),
                    .mem_read(mem_rd),
                    .mem_write(mem_wr),
                    .halt(halt),
                    .N_flag(N),
                    .Z_flag(Z),
                    .data_out_sel(X_sel),
                    .data_in_sel(Y_sel),
                    .address_sel(addr_sel),
                    .alu_op(M)
                    .instr(ir_out[15:12]));
```

Implementing the control

What does the control do?

… remember it issues commands to the datapath & contains the fsm!

We will be implementing the control block in Verilog.

It contains the state machine for the MU0, this can be implemented easily as an `always` block within the control, one implementation may be:

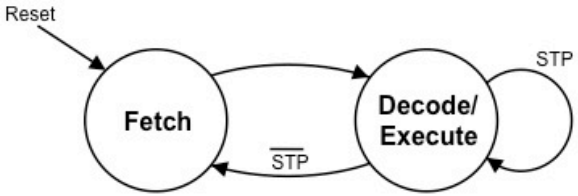
```
always @ (posedge clock, posedge reset)
begin
  if(reset)
    state <= `FETCH;
  else
    case(state)
      `FETCH: state <= `EXECUTE;
      `EXECUTE: state <= `FETCH;
    endcase
end
```

The control also has to set the control signals to the rest of the datapath. There are many ways we could do this …

Notes:

MU0 FSM

Remember, MU0 has a straightforward finite state machine with 2 states:



where

Fetch – read the instruction from the memory address specified in the PC into the IR
Decode/Execute – decode IR and perform the required operation. A STP instruction will halt the processor and the fsm should remain in the Decode/Execute state.

`define pre-compiler directive

In the code example in the slide we assume that the labels FETCH and EXECUE have been defined in the file, i.e.

```
`define  FETCH    0
`define  EXECUTE  1
```

which are used in the Verilog code as

```
`FETCH
`EXECUTE
```

Q.

The Verilog code in the figure is incomplete – the control provided by the STP instruction is not included. How could the fsm implementation be modified to include this?

Implementing the control

We need to identify which datapath control signals are asserted in which state and for which instruction …

One way to simplify this process is to use signal usage charts, which are essentially a state transition table that lists all the control signals in datapath against each type of instruction, with a table for each state of the fsm, i.e. `fetch` and `execute`.

Consider the **Fetch** phase … there's no dependency on the type of instruction

mem_read	1
mem_write	0
acc_en	0
pc_en	1
ir_en	1
data_out_sel	1
data_in_sel	x
address_sel	0
alu_op	2'b10
halt	0

Notes:

Signal Usage Charts

When designing any complex sequential system it is useful to identify how each signal changes as a result of the signals/data that influence its behaviour – you can view a signal usage chart as being something similar to a truth table/state transition table – it is another tool available to the designer to help in the design of complex systems. In the case of the MU0 control block we are interested in how the state of the control signals is influenced by the current state of the MU0 fsm and the current instruction. We could tabulate these in one (very large) table, but this could end up being confusing. Consequently, we know that the MU0 resides in two states: fetch and decode/execute, we can draw two tables, one for each state, to show how the state of the control signals during each state.

In the fetch state the state of the control signals does not depend on the instruction (in fact we don't know what the instruction is yet) so we only have one column.

Implementing the control

Consider the **Decode/Execute** phase

… depends on the type of instruction

instr	LDA	STA	ADD	SUB	JMP	JGE	JNE	STP
	4'b0000	4'b0001	4'b0010	4'b0011	4'b0100	4'b0101	4'b0110	4'b0111
mem_read	1	0	1	1	0	0	0	1
mem_write	0	1	0	0	0	0	0	1
acc_en	1	0	1	1	0	0	0	0
pc_en	0	0	0	0	1	$\overline{N}$	$\overline{Z}$	0
ir_en	0	0	0	0	0	0	0	0
data_out_sel	x	0	0	0	x	x	x	x
data_in_sel	0	x	0	0	1	1	1	x
address_sel	1	1	1	1	x	x	x	x
alu_op	2'b00	2'b00	2'b01	2'b11	2'b00	2'b00	2'b00	2'b00
halt	0	0	0	0	0	0	0	1

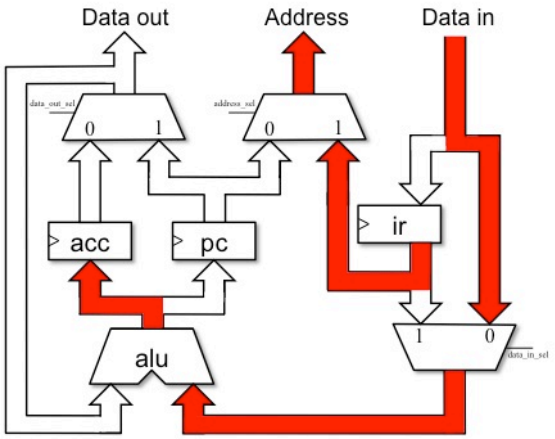
Notes:

Signal Usage Charts

In the decode/execute state the control signals depend upon the type of instruction, hence we must tabulate the state of each control signal as a function of the instruction. In each case we need to identify what components in the datapath are used for each instruction.

Example: LDA Instruction

The path usage in the RTL datapath is shown below.



Where we take the address from the instruction, place this on the address bus, initiate a memory read and move the data from the data in bus to the accumulator. Hence,

- the address select mux need to select the address from the ir – address_sel = 1,
- we are reading from memory so mem_read = 1, mem_write = 0,
- data is read in from the data in bus so the data in mux needs to select data in bus – data_in_sel – 0,
- the alu is not doing anything, so we need the bypass operation – M = 00,
- we need to write the data to the accumulator, so we need to enable it – acc_en = 1.

The remaining control signals are assigned as follows:

- the ir is not updated so ir_en = 0,
- the pc is not updated so pc_en = 0,
- we don't care what the data out mux is doing so data_out_sel = x,
- halt = 0.

Implementing the control

MANCHESTER 1824
The University of Manchester

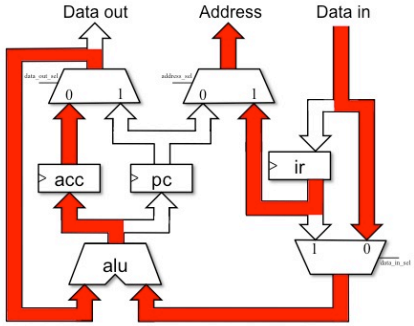
Now we know how the control signals depend on the type of instruction and the phase (fetch or execute) of the instruction.

There are many ways we could approach the implementation in Verilog…

- We could set the control signals within the fsm `always` block – this could be difficult to get 'correct'
- We could create another `always` block to set all the control signals
- We could create an `always` block for each control signal that determines the signal value depending on whether it is the fetch or execute phase, and the type of instruction.
- … and more – each would probably create a block of synthesizable Verilog, although some would be easier to implement that others.
- … there's no right or wrong way, as look as you focus on implementing synthesizable Verilog.

Notes:

Example: ADD Instruction



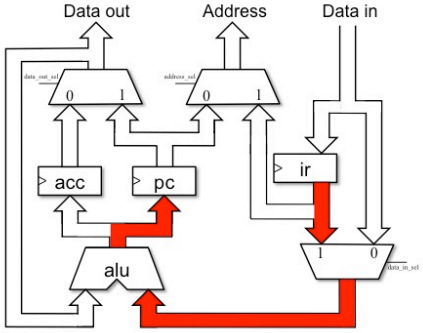
The path usage in the RTL datapath is shown opposite. For the ADD instruction we must take the address from the instruction in the ir and place it on the address bus, the data on the data_in bus is then passed to the alu and added to the current value in the accumulator. The result is stored in the accumulator.

- the address select mux need to select the address from the ir – address_sel = 1,
- we are reading from memory so mem_read = 1, mem_write = 0,
- data is read in from the data in bus so the data in mux needs to select data in bus – data_in_sel – 0,
- the alu is performing addion so M = 01,
- we need to add the data to the current value of the accumulator so the data out mux needs to select form the acc, so data_out_sel = 0.

we need to write the data to the accumulator, so we need to enable it – acc_en = 1.The remaining control signals are assigned as follows:

- the ir is not updated so ir_en = 0,
- the pc is not updated so pc_en = 0,
- halt = 0.

Example: JNE Instruction



The path usage in the RTL datapath is shown opposite. Where we take the address from the instruction and write to the program counter ONLY if the jump is to be taken. Hence,

- data is read in from the ir so the data in mux needs to select data in bus – data_in_sel = 1,
- the alu is not doing anything, so we need the bypass operation – M = 00,
- we need to write the data to the pc ONLY if the jump is to be taken – we can determine this from the N status flag. If Z ≠0, so we need to enable it when Z ≠0, i.e. pc_en = $\bar{N}$.

The remaining control signals are assigned as follows:

- we are not doing a memory access so mem_read = mem_write = 0,
- the ir is not updated so ir_en = 0,
- the accc is not updated so accc_en = 0,
- we don't care what the data out mux is doing so data_out_sel = x,
- we don't care what the address mux is doing so address_sel = x,
- halt = 0.

Implementing the control

We'll have a go at implementing a single `always` block for setting the control signals ...

```
always @ (*)
  case  (state)
    `FETCH:   begin
                mem_read = 1'b1;
                mem_write = 1'b0;
                acc_en = 1'b0;
                pc_en = 1'b1;
                ir_en= = 1'b1;
                data_out_sel = 1'b1;
                data_in_sel = 1'bx;
                address_sel = 1'b0;
                alu_op = 2'b10;
                halt = 1'b0;
              end
    `EXECUTE: begin
              case (instr)
                // see notes
              endcase
  endcase
```


Notes:

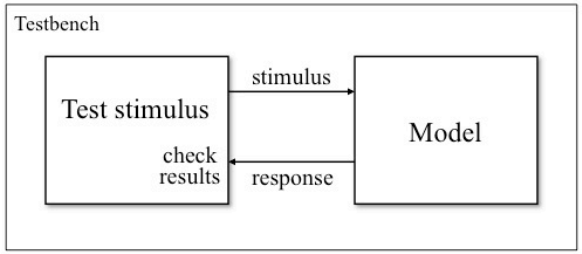
Full always block:

```
always @ (*)
 case  (state)
  `FETCH:   begin
              mem_read = 1'b1;
              mem_write = 1'b0;
              acc_en = 1'b0;
              pc_en = 1'b1;
              ir_en= = 1'b1;
              data_out_sel = 1'b1;
              data_in_sel = 1'bx;
              address_sel = 1'b0;
              alu_op = 2'b10;
              halt = 1'b0;
            end
  `EXECUTE: begin
              ir_en = 1'b0;
              address_sel = 1'b1;
              data_out_sel = 1'b0;
              case (instr)
                `LDA: begin
                        mem_read = 1'b1;
                        mem_write = 1'b0;
                        acc_en = 1'b1;
                        pc_en = 1'b0;
                        data_in_sel = 1'b0;
                        alu_op =  2'b00;
                        halt = 1'b0;
                      end
                `STA: begin
                        mem_read = 1'b0;
                        mem_write = 1'b1;
                        acc_en = 1'b0;
                        pc_en = 1'b0;
                        data_in_sel = 1'bx;
                        alu_op =  2'b00;
                        halt = 1'b0;
                      end
                `ADD: begin
                        mem_read = 1'b1;
                        mem_write = 1'b0;
                        acc_en = 1'b1;
                        pc_en = 1'b0;
                        data_in_sel = 1'b0;
                        alu_op =  2'b01;
                        halt = 1'b0;
                      end
                `SUB: begin
                        mem_read = 1'b1;
                        mem_write = 1'b0;
                        acc_en = 1'b1;
                        pc_en = 1'b0;
                        data_in_sel = 1'b0;
                        alu_op =  2'b11;
                        halt = 1'b0;
                      end
                `JMP: begin
                        mem_read = 1'b0;
                        mem_write = 1'b0;
                        acc_en = 1'b0;
                        pc_en = 1'b1;
                        data_in_sel = 1'b1;
                        alu_op =  2'b00;
                        halt = 1'b0;
                      end
                `JGE: begin
                        mem_read = 1'b0;
                        mem_write = 1'b0;
                        acc_en = 1'b0;
                        pc_en = ~N_flag;
                        data_in_sel = 1'b1;
                        alu_op =  2'b00;
                        halt = 1'b0;
                      end
                `JNE: begin
                        mem_read = 1'b0;
                        mem_write = 1'b0;
                        acc_en = 1'b0;
                        pc_en = ~Z_flag;
                        data_in_sel = 1'b1;
                        alu_op =  2'b00;
                        halt = 1'b0;
                      end
                `STP: begin
                        mem_read = 1'b0;
                        mem_write = 1'b0;
                        acc_en = 1'b0;
                        pc_en = 1'b0;
                        data_in_sel = 1'b0;
                        alu_op =  1'b0;
                        halt = 1'b1;
                      end
              endcase
            endcase
```

Test Bench

How do we check that the model is correct?

…we test!



Same test should be applied at every level of the design hierarchy and on the fabricated chip. Test must be passed before proceeding to the next level down.

Notes:

Test Bench

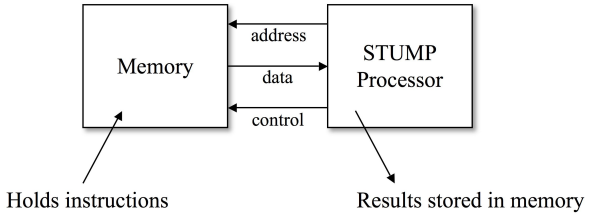
Testing is one of the most important stages of ANY design process, otherwise how do you know that the design conforms to the specification.

It is important that at every stage of the design process that you test your design for correctness. How do we do this? We use a test bench and associated tests to test different aspects of the system for functionality and correct operation against the user's specification. The test bench provides data to the system under test whose outputs are tested against *expected* results. If the output data is in error then the model needs to be corrected and tests re-run until the functionality is correct.

Tests can't be exhaustive (particularly if you consider the many states a 32-bit processor can reside in – there won't be enough time in the history of mankind to do this) instead they are confidence tests, which often test functionality and not data. If all the tests are passed then you are confident the system works and conforms to the user requirements.

Tests are very important as they confirm the validity of the design. It is important that that **same** tests are used at *every* level of the design process and on the final fabricated chip, and the tests must pass at each level before proceeding to the next level down.

The STUMP test bench is



Here the STUMP processor is tested using a memory that holds instructions. As instructions are executed results are stored back to memory so that the results of the test can be viewed to check whether the processor operates correctly.

Top-level model test

When the system spec (the top-level behavioural model) passes all the tests, this functional behaviour is then used as a gold standard by all other levels and we can move onto the next stage in the design process which involves the architectural design of the system.

Verilog Test Bench

MANCHESTER 1824
The University of Manchester

```
module MU0_test();

// declarations, reg etc. – removed for clarity

MU0 processor(.clock (clk),                      // Instantiate the device
              .reset (reset),
              .data_in (data_in),
              .data_out (data_out),
              .address (address),
              .mem_rd (memory_write),
              .mem_wr (memory_write),
              .halt (halt));

initial                                          // This block resets the processor
begin                                            // and, later, stops the simulation
reset = 0;
#200;
reset = 1;
#100;
reset = 0;
#1000;
$stop;
end

initial memory[0] = 16'h0000;                    // Memory initialisation
initial memory[1] = 16'h2002;                    // Read from file really @@@
initial memory[2] = 16'h1000;
initial memory[3] = 16'h4000;

always @ (address)                               // Trivial memory read @@@
  #20 data_in = memory[address];


endmodule
```


Notes:

Verilog Test Bench

Rather than building a test bench in a schematic, it is possible to produce one in Verilog where you instantiate the module under test in the Verilog code. This is actually what happened in COMP12111, but it was hidden from you! The example is part of the test bench for the MU0, where we instantiate the MU0 processor and define the input/output signals. The following stimulus then exercises the input signals in order to perform the test and the resulting output signals from the processor are monitored in order to verify correct operation.

Type 3 (Branch) Instructions

bal #-5

instruction: 11110000 11111011

action: R7 = R7 - 5

The offset can be specified as a label that the compiler will convert to the appropriate offset value.

Q.

Write the instruction and action for the following Type 3 instructions.

i) bne #7 when N=0, Z=0, V=0, C=0, current PC value is FE (in Hex)

ii) bcc #-8 when N=0, Z=0, V=1, C=0, current PC value is 5C

iii) beq #-20 when N=0, Z=1, V=0, C=0, current PC value is 20

iv) bhi #5 when N=0, Z=0, V=1, C=1, current PC value is 2D

Overview

We have looked at:

- RISC v CISC
- A general RISC instruction format
- Simple RISC control
- Pipelining
- Developed a specification for a simple RISC processor – the STUMP
- Looked at the STUMP instruction set
- Briefly looked at testing of the STUMP
- Developed n architectural design for the STUMP and checked it using path usage diagrams
- Developed an RTL design for the STUMP
- We've looked at implementing the control using MU0 as a design example

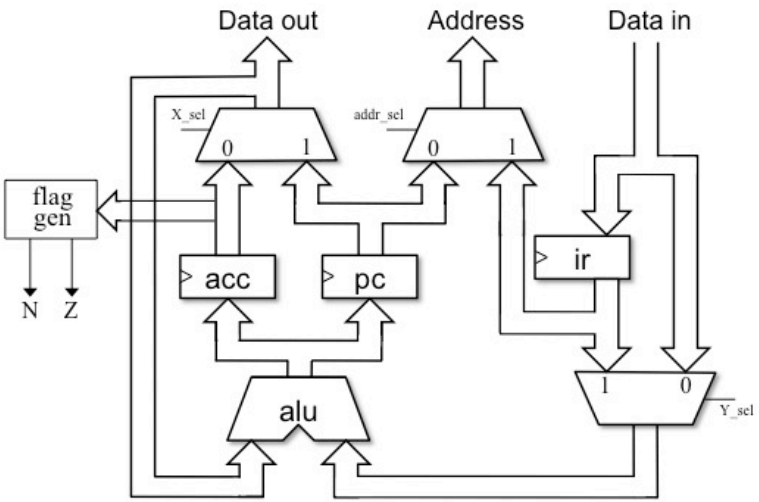
… you will now build the STUMP in the lab!

Notes:

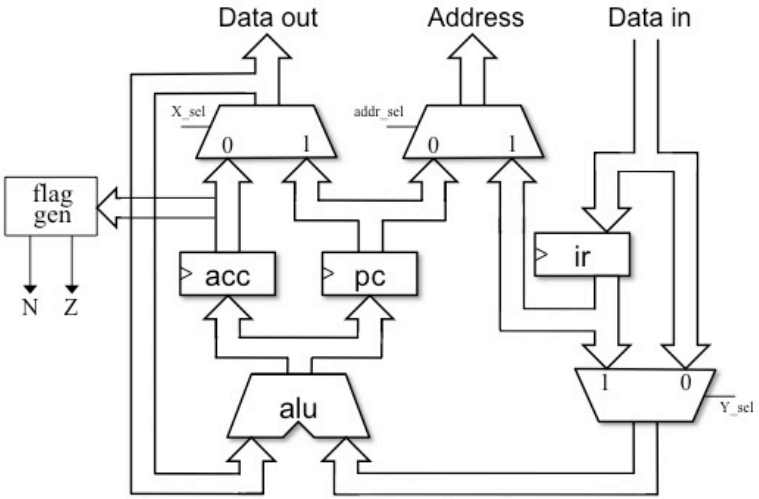
Full RTL Diagram for the STUMP

We have looked at a general RISC processor and the enhancements that can be made by pipelining (plus some of the hazards). We then went on to look at the STUMP specification and from this developed an architectural model of the STUMP, which we checked using path usage diagrams. We then went on to develop an RTL implementation of the STUMP.

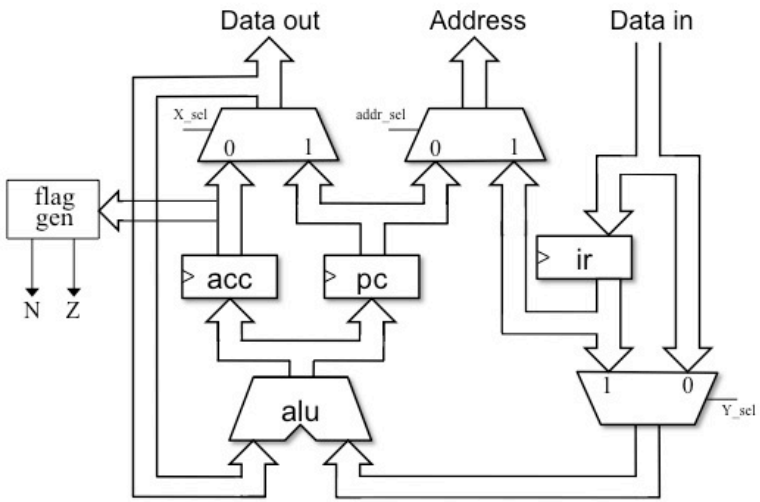
Fetch



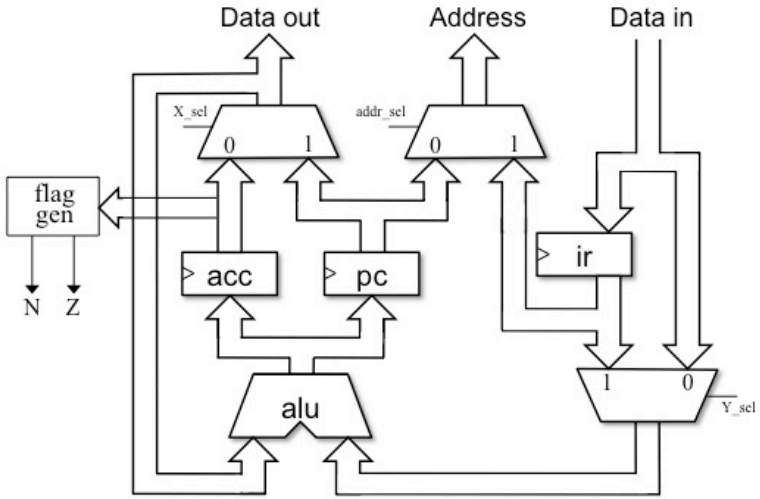
LDA



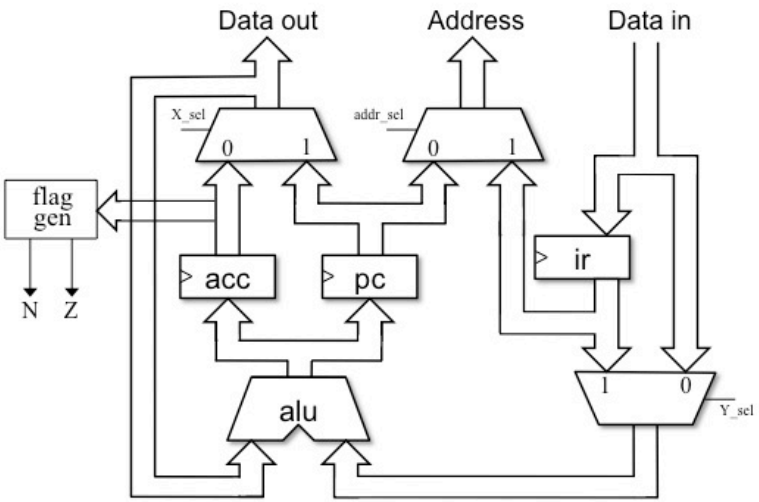
STA



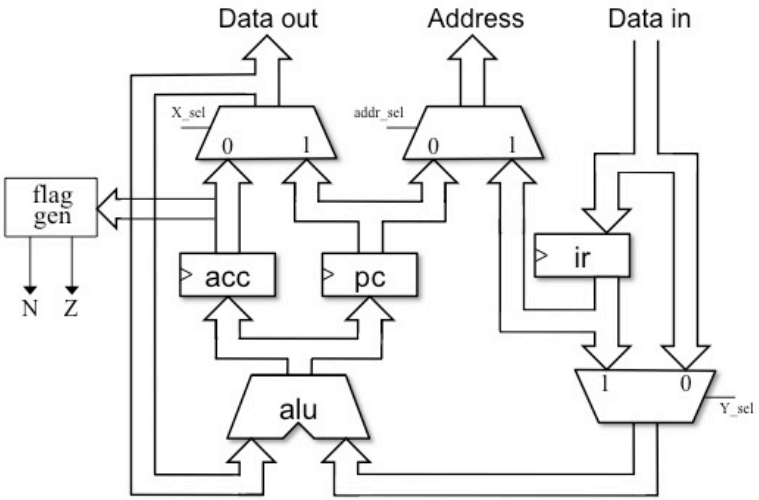
ADD



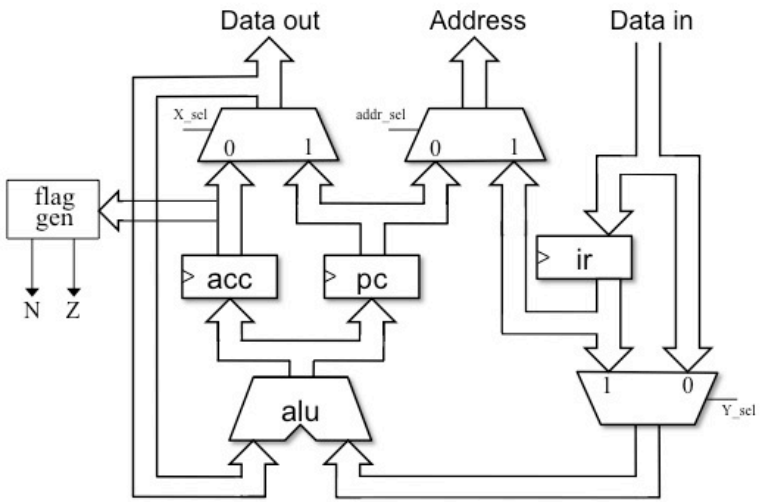
SUB



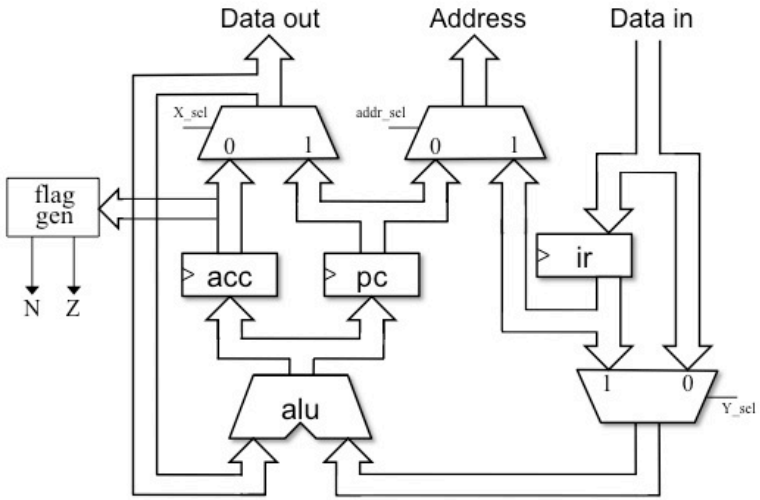
JMP



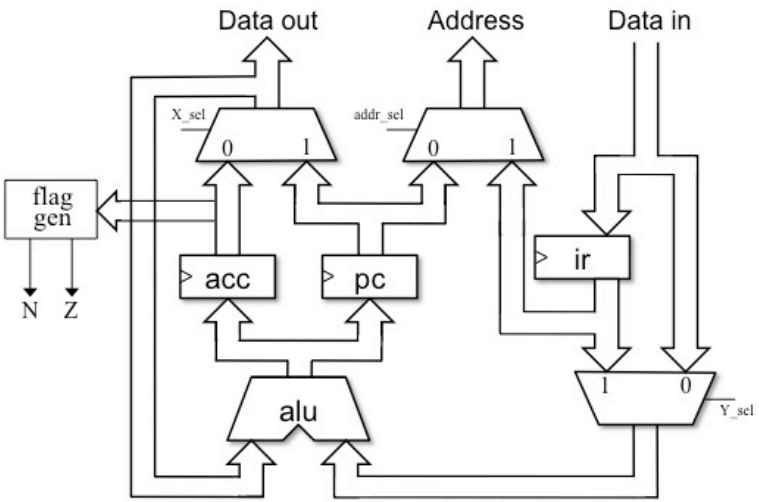
JGE



JNE



STP


Data out
Address
Data in
X_sel
addr_sel
0
1
0
1
flag
gen
N
Z
acc
pc
ir
alu
1
0
Y_sel