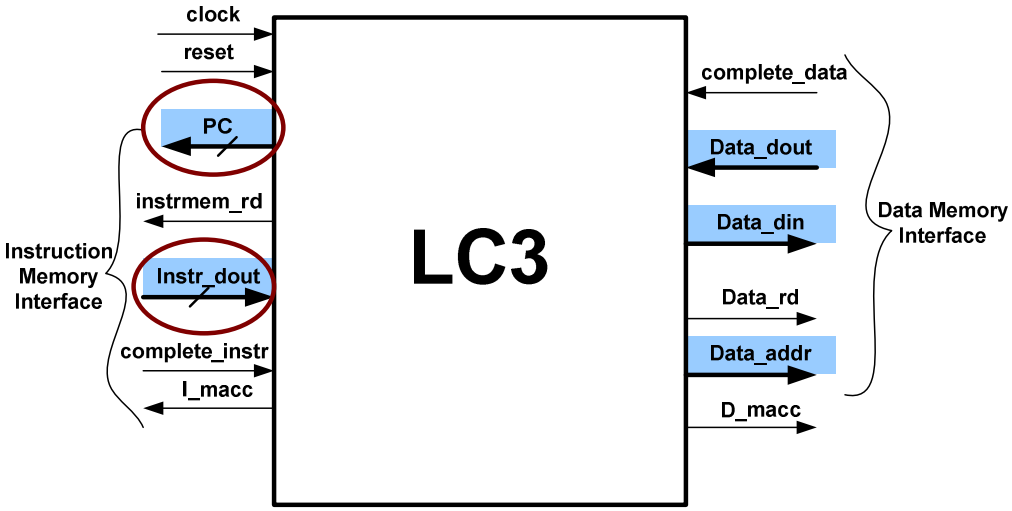
ECE 745 : ASIC VERIFICATION
PROJECT 1 LC-3 DESIGN SPEC

Introduction:
This project deals with the verification of the data and control path of an UNPIPELINED LC-3 microcontroller with a reduced instruction set. This document will take you through the implementation and specification of this microcontroller for the reduced instruction set you will be dealing with in this project. The aim is to provide the introductory first steps in dealing with the system that you would need to get familiar with as you progress in this class.

To begin, the top level block diagram for the Design Under Test (DUT) being considered is shown in Fig. 1.



`Instruction Register = IR <= IMem_dout`

Figure 1: Top Level Block Diagram of LC3

The inputs and outputs to this design are:

Inputs to DUT

- `clock` **(1 bit)**
- `reset` **(1 bit)**
- `complete_data` **(1 bit)** (this signal will be explained in later sections)
- `complete_instr` **(1 bit)** (this signal will be explained in later sections)
- `Instr_dout` **(16 bits)**Corresponds to the instruction from the Instruction Memory into the DUT i.e. IMem[PC].
- `Data_dout` **(16 bits)**Corresponds to the value read from the Data Memory into the DUT for loads. **Ignore for this project.**

Outputs from DUT

- `PC` **(16 bits)**This corresponds to an address to the Instruction Memory (given that the instruction being fetched corresponds to the PC in question).
- `instrmem_rd` **(1 bit)**This signal enables a read from the Instruction Memory for a fetch.
- `Data_addr` **(16 bits)**Corresponds to the address sent to the Data Memory for reads from it. **Ignore for this project.**
- `Data_din` **(16 bits)** Corresponds to the values that need to be written to Data Memory which would correspond to stores. **Ignore for this project.**
- `Data_rd` **(1 bit)**This signal enables a read from the Data Memory. If this signal is 0 then a write to Data Memory is enabled. **Ignore for this project.**
- I_macc and D_macc : These will be used in the future projects to distinguish between Instruction and Data memory access phases. **Ignore for this project.**

As stated earlier, we are going to deal with only a smaller subset of the possible LC3 instructions *i.e.* ALU Operations (AND, NOT, ADD) and the one of the Memory operations called LEA (Load Effective Address). The reason for the use of this subset is to provide an introduction to the signals of importance in the datapath of the LC3. Moreover, in this project we are going to be working exclusively with an un-pipelined version of the LC3. We will progress towards a fully pipelined version and a complete set of instructions as we go ahead in this course. Additionally, the instructions of interest for this project have been chosen such that each one ends in exactly 5 clock cycles.

Instructional Example:
The operation of a microcontroller is controlled by the contents of the instruction memory. The content read out, called an instruction, is a 16 bit value which causes the microcontroller to perform a specific function. To help perform the function, there would be a set of memory locations used to store values that can be shared between multiple instructions. In case of the LC3, we have 8 such locations, R0 – R7 which can be accessed for reading (using SR1 and SR2 say) and writing (using DR) . Let us assume, `SR1` = 3bits = Source register 1 address; `SR2` = 3bits = Source register 2 address and `DR` = 3bits = Destination register address;

Let us assume, we want to perform the following set of functions
`AND R0, R0, #0` → `ADD R2, R0, #2` → `ADD R1, R2, R0`

In case of the LC3, these instructions would be stored, starting at 3000 and would be addressed using something called the Program Counter (`PC`) seen above. The content from the Instruction memory corresponding to the location `PC` would be asserted on `Instr_dout`. Thus, the sequence of operations that would be performed in the LC3 in this case would be,

- `PC` = **3000** which leads to `Instr_dout` = `IMEM`[**3000**] = **5020**. Here, 5020 the way `AND R0, R0, #0` is encoded.
- `SR1` = **`R0`** ; Source 2 = **`Immediate(from IR) #0`**; Dest = **`R0`**
- Operation Performed = **`R0 ← R0 & 0 = 0`**

At this point, the first instruction is done with and the next one can be performed. This corresponds to

- `PC` = **3001** and hence `Instr_dout` = `IMEM`[**3001**] = **1422** (`ADD R2, R0, #2`)
- `SR1` = **`R0`** ; Source 2 = **`Immediate(from IR) #2`**; Dest = **`R2`**
- Operation Performed = **`R2 ← R0 + 2= 2`**

And finally we have,

- `PC` = **3002** which leads to `Instr_dout` = `IMEM`[**3002**] = **1280** (`ADD R1, R2, R0`)
- `SR1` = **`R2`** ; `SR2` = **`R0`**; Dest = **`R1`**
- Operation Performed = **`R1 ← R2 + 0 = 2 + 0 = 2`**

Thus, we notice that the operation of the LC3 can be controlled in terms of the PC value and the content of the instruction memory corresponding to the location `[PC]`.

To understand the operation of the system it becomes necessary to look at the operation of the different instructions available. The different instructions available can be identified on the basis of the value of the instruction read in from the instruction memory i.e. `IMem[PC]` i.e. for a particular PC.

The core operations based on the value of the incoming Instruction structure can be divided into:

ALU Operations: (AND, ADD, NOT)

Instruction	15	14	13	12	11 10 9	8 7 6	5	4	3 2 1 0
ADD	0	0	**0**	**1**	DR	SR1	0	0	0 SR2
	0	0	**0**	**1**	DR	SR1	1	imm5	
AND	0	1	**0**	**1**	DR	SR1	0	0	0 SR2
	0	1	**0**	**1**	DR	SR1	1	imm5	
NOT	1	0	**0**	**1**	DR	SR1	1	1	1 1 1 1 1

There are two variations for the ADD or AND operations:

1. Immediate (`[DR] ← [SR1] +/& imm5(sign extended)`)
2. Register (`[DR] ← [SR1] +/& [SR2]`)

The NOT operations works simply as `[DR] ← ~[SR1]`

For example, if instruction `(IR)` = **`16'h12BC`**, assume `R2 = 16'h0030 = 48` (decimal)

- **`IR[15:12] = 0001`** => operation is an ADD;
- **`DR = IR[11:9] = 001`** => we are writing to i.e. Destination Register = R1;
- **`SR1 = IR[8:6] = 010`** => We are reading from i.e. source register 1 = R2;
- since **`IR[5] = 1`** => Immediate mode, **`IR[4:0] = imm5 = 28 = -4`**
- Resulting operation (in hex): **`R1← R2 + sxt(-4)`**
  `R1 ← 16'h30 + 16'hFFFC = 002C` (ignoring carry out) = **`44`**

Note the extension with sign (`sxt()`) of the immediate to 16 bits in the above. This sign extension is performed to allow the execution to be performed in 2's complement.

Memory Operations: (LD, LDR, ST, STR, LDI, STI, LEA)

Memory operations have two variants: Load (**LD[x]**) and Store (**ST[x]**) instructions based on whether memory is being read from or written to. The stores to memory are done using values read from the register file (SR in the table below) and loads involve reading from memory into the register file (DR in the table below). The memory address to be read from or written to is created using the PC of the memory instruction and the offsets (`PCoffset9/PCoffset6`). The memory operations can be divided into different modes based on the way the memory addresses are created for loads and stores.

Instruction	15			12	11	9	8	6	5	4	3	0
LD	0	0	**1**	**0**	DR		PCoffset9					
LDR	0	1	**1**	**0**	DR		BaseR		PCoffset6			
LDI	1	0	**1**	**0**	DR		PCoffset9					
LEA	1	1	**1**	**0**	DR		PCoffset9					
ST	0	0	**1**	**1**	SR		PCoffset9					
STR	0	1	**1**	**1**	SR		BaseR		PCoffset6			
STI	1	0	**1**	**1**	SR		PCoffset9					

In the discussion below, we will assume that the PC of the memory instruction under analysis is **PC_{mem}.**

- *PC Relative* (LD/ST): Here the effective memory address for read or write is formed as

  Mem_Addr = **PC_{mem}** + 1 + sign-extended(*PCoffset9*)

  And the resulting read/write to memory is done as

  [DR] ← DMem[**Mem_Addr**] // For LD
  DMem[**Mem_Addr**] ← [SR] // For ST

- *Register Relative* (LDR, STR): Here the effective memory address for read or write is formed as

  Mem_Addr = [BaseR] + sign-extended(*PCoffset6*)

  And the resulting read/write to memory is done as

  [DR] ← DMem[**Mem_Addr**] // For LDR
  DMem[**Mem_Addr**] ← [SR] // For STR

- *Indirect* (LDI, STI): Here the effective memory address for read or write is formed as

  Mem_Addr1 = **PC_{mem}** + 1 + sign-extended(*PCoffset9*)
  Mem_Addr = DMem[**Mem_Addr1**] ;

  Therefore, we see that the initial read from Data Memory gives an address which is used to do a subsequent read from / write to Data Memory.
  And the resulting read/write to memory is done as

  [DR] ← DMem[**Mem_Addr**] // For LDI
  DMem[**Mem_Addr**] ← [SR] // For STI

- Load Effective Address (LEA): This operation does not deal with memory. It is essentially creates an address for a future register-based load or store.

  Mem_Addr = **PC_{mem}** + 1 + sign-extended(*PCoffset9*)
  [DR] ← **Mem_Addr**

For example, if instruction (IR) = **16'hA7E8**, **PC_{mem}** = **16'h310C**, **DMem[16'h30F5]** = **16'h301A**, **DMem[16'h301A] = 16'h000A**

- **IR[15:12] =1010** => operation is an LDI;
- **DR = IR[11:9] = 011** => we are writing to i.e. Destination Register = R3;
- **Mem_Addr1** = **PC_{mem}** + 1 + sign-extended(*PCoffset9*)
  Mem_Addr1 = 310C + 1 + sxt(-24) = 310C + 1 + sxt(1E8)
  = 310C + 1 + FFE8 = 30F5
- **Mem_Addr** = **DMem[30F5]** = **16'h301A**
- Resulting operation (in hex): **R3←DMem[16'h301A]**
  R1 ← 16'h000A

An example for LEA is given in the accompanying presentation.

Note the extension with sign (sxt()) of the immediate to 16 bits in the above. This sign extension is performed to allow the execution to be performed in 2's complement.

Control Instructions: (BR*x* Offset, JMP BaseR)

Instruction	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BR	0	0	0	0	N	Z	P	PCoffset9								
JMP	1	1	0	0	0	0	0	BaseR			0	0	0	0	0	0

The Branch is a conditional statement whose condition being met is dependant upon the checking of the conditional flags NZP (Negative, Zero and Positive). The checks are performed against the results of the previous register file affecting operation i.e. loads, ALU operations and LEA operation. The following combinations can be explored for Branch i.e. BR.

- `==0  (BRZ)` NZP = 010
- `!=0  (BRNP)` NZP = 101
- `>0  (BRP)` NZP = 001
- `>=0  (BRZP)` NZP = 011
- `<0  (BRN)` NZP = 100
- `<=0  (BRNZ)` NZP = 110
- `Unconditional jump  (BRNZP or simply BR)` NZP = 111

The branch shall take place if any of the conditions being sought (`N, Z or P` being 1) is satisfied. Assume now that the PC of the instruction that contains a branch is $\mathbf{PC_{branch}}$. On the satisfactory passing of the status flag check the branch leads to the updating of the `PC` to PC_{next} which changes the next instruction to be executed after a branch to be:
$\mathbf{PC_{next}} \leftarrow \mathbf{PC_{branch}} + \mathbf{1} +$ sign extended*(PCoffset9)*

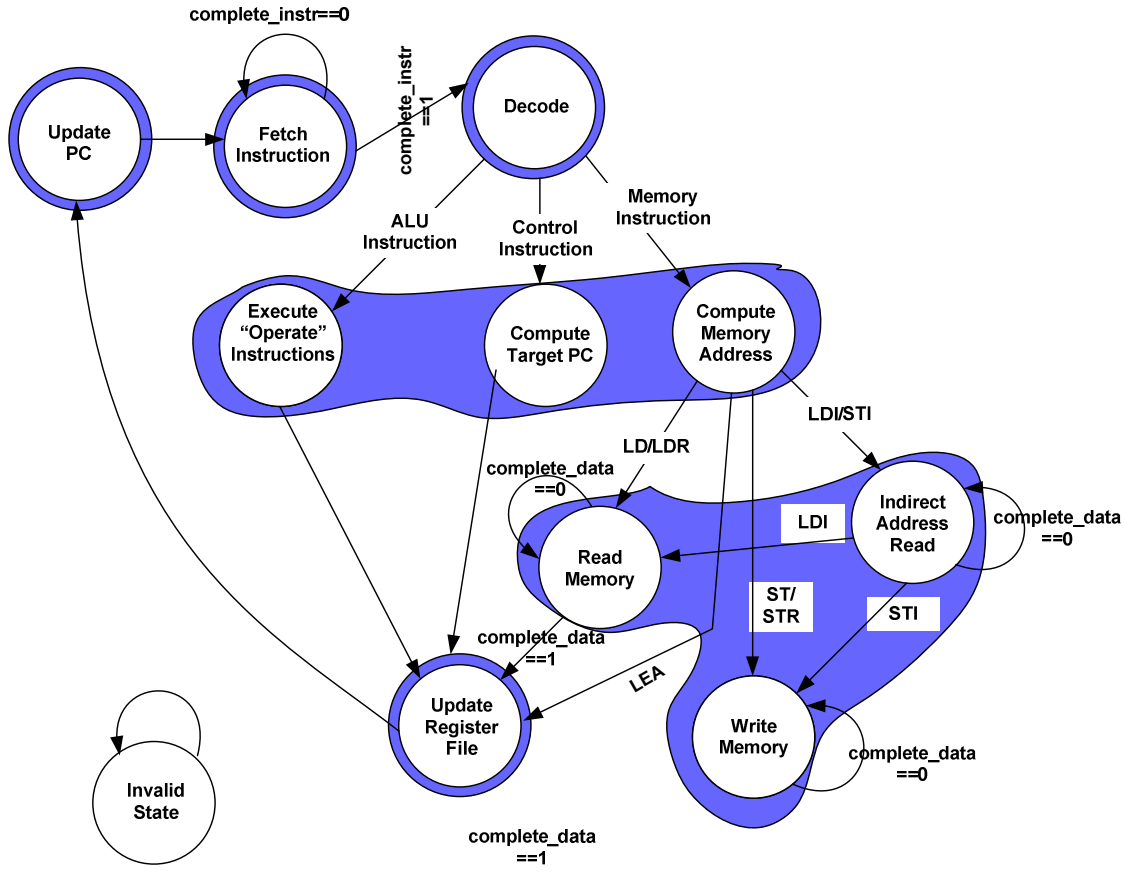
The JMP instruction is an unconditional branch statement. This leads to the change in $(\mathbf{PC_{next}})$ to
$\mathbf{(PC_{next})} \leftarrow \mathbf{[BaseR]}$

To determine whether the conditions for the branch are met or not, the result of the previous register file manipulating instruction is recorded in what is called the PSR register. The PSR register is written to in the following conditional manner:

```
if(register write negative)       // Negative
      psr     <=      3'h4;
else if(register write positive)  // Positive
      psr     <=      3'h1;
else                              // Zero
      psr     <=      3'h2;
end
```

For example, when we want to branch when the previous register modifying instruction gave a positive result we would need to do a BRP which would be encoded as: `IR = 16'h03FD` where we assume *`PCOffset9`* `= 1FD` and if the check for the `NZP = IR[11:9] = 3'b001` proves to be successful, the `PC` value to be used after the branch would be $\mathbf{PC_{next} = PC_{branch} + 1 + sxt(1FD) = PC_{branch} + 1 + (-3)}$

An important consequence of the above analysis is that we can now break up the entire process of servicing instructions of different types into their constituent steps. Based on the reusability of computation, we can then determine a means of pipelining the LC3 and providing computational structures that would service each incoming instruction in a sequence of clocked stages. This manifests itself as the FSM shown below. **Each state in the FSM deals with a certain operation for ONE instruction.** It must be noted that this state machine is used in this context to explain the typical steps that ONE instruction would follow. The operations that could be performed by a single clocked hardware unit in a pipeline are grouped in the state machine. "Execute Operate Instruction", "`Compute Target PC`" and "`Compute Memory Address`" can be done by one unit (here, Execute) given that they are mutually exclusive states.



The states that are followed are dependant upon the type of instruction coming in. We can look at each instruction in an individual sense for now. The states that should be followed for a given instruction based on opcode (function of `IR[15:12]`) are (keeping it generic for now)

a) ALU instructions: **(5 clock cycles not considering `complete_instr`)**
   a. (*Fetch Instruction*) Fetch Unit enables instruction load from memory. **Waits for complete_instr to go high before it transitions to next state.**
   b. (*Decode*) Decode Unit determines the operands and the operation type and asserts the right control signals for ALU operations to be executed down the line. The signals of importance will be come clear in the next section.
   c. (*Execute Operate Instructions*) Execute Unit applies operands to ALU and performs operations based on operation type. It is to be noted that we could have multiple variants a) the type of operation (**ADD, AND, NOT**) and the values being worked with [`(SR1 with imm5) / (SR1 with SR2)` / just `SR1`]
   d. (*Update Register File*) ALU Operation result stored in Register File
   e. (*Update PC*) `PC` incremented to `PC + 1`

b) Control Instructions: **(4 clock cycles not considering `complete_instr`)**
   a. (*Fetch Instruction*) Fetch Unit enables instruction load from memory. **Waits for complete_instr to go high before it transitions to next state.**
   b. (*Decode*) Decode operation type and the choice of the source for the creation of the new `PC` i.e. PC_{branch} `+ 1 or [BaseR]`
   c. (*Compute Target PC*) Execute computes new `PC` (PC_{new}) for either Branch or Jump and using sign extension.
   d. (*Update PC*) PC updated to either `PC + 1` or PC_{new}.

c) Memory Instructions:
   a. LD/LDR: **(6 clock cycles not considering `complete_(instr/data)` signals)**
      i. (*Fetch Instruction*) Fetch Unit enables instruction load from memory. **Waits for `complete_instr` to go high before it transitions to next state.**
      ii. *Decode* Decoding performed keeping in mind the need for recognition of variations in memory address type and the sources of offset (*`PCoffset6/ PCoffset9`)* and choices of either working with `PC` or `BaseR` and recognizing the need for a read from memory to a destination register `DR`.
      iii. (*Execute: Compute Memory Addresss*)Execute Unit computes address
      `Mem_Addr =` PC_{mem} `+ 1 +` *`PCoffset9`*`(sign-extended)` OR
      `[BaseR] +` *`PCoffset6`*`(sign-extended)`
      iv. (*Read Memory*) MemAccess Unit reads Data Memory ←`MEM[Mem_Addr].` **Waits for `complete_data` to go high before it transitions to next state.**
      v. (*Update Register File*)Write to Register File
      `[DR] ← DMem[Mem_Addr]`
      vi. (*Update PC*) PC incremented

b. ST/STR: **(5 clock cycles not considering `complete_(instr/data)` signals)**
   i. *Fetch* Fetch Unit enables instruction load from memory. **Waits for `complete_instr` to go high before it transitions to next state.**
   ii. *Decode* Decoding performed keeping in mind the need for recognition of variations in memory address type and the sources of offset (*`PCoffset6/ PCoffset9`)* and choices of either working with `PC` or `BaseR` and recognizing the need for a write to memory from a source register `SR`.
   iii. (*Execute: Compute Memory Addresss*)Execute Unit computes address
   `Mem_Addr =` PC_{mem} `+ 1 +` *`PCoffset9`*`(sign-extended)` OR
   `[BaseR] +` *`PCoffset6`*`(sign-extended)`
   iv. (*Write Memory*) MemAccess Unit writes Data Memory (`DMem[Mem_Addr] ← [SR]`). **Waits for `complete_data` to go high before it transitions to next state.**
   v. (*Update PC*)PC incremented

c. LDI **(7 clock cycles not considering `complete_(data/instr)` signals)**
   i. *Fetch* Fetch Unit enables instruction load from memory. **Waits for `complete_instr` to go high before it transitions to next state.**
   ii. *Decode* Same as above
   iii. (*Execute: Compute Memory Addresss*)Execute Unit computes address
   (`Mem_Addr1 =` PC_{mem} `+ 1 +` *`PCoffset9`*`(sign-extended)`)
   iv. (*Indirect Memory Access Read*) MemAccess Unit reads Data Memory (`Mem_Addr = DMem[Mem_Addr1`). **Waits for `complete_data` to go high before it transitions to next state.**
   v. (*Read Memory*). MemAccess Unit reads Data Memory (`←DMem[Mem_Addr]`). **Waits for `complete_data` to go high before it transitions to next state.**
   vi. (*Update Register File*)Write to Register File
   `[DR] ← DMem[Mem_Addr]`
   vii. (*Update PC*)PC incremented

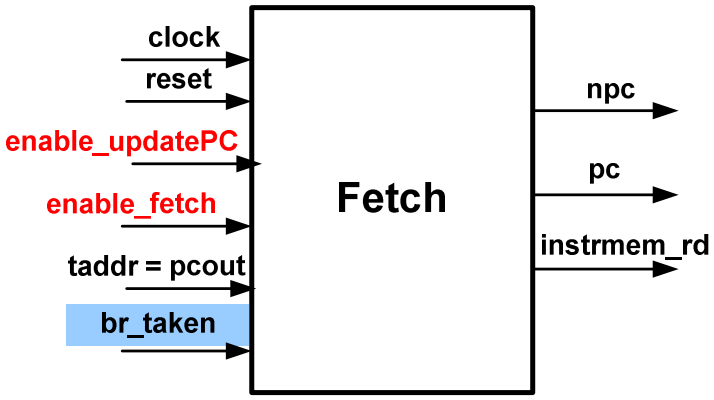
d. STI **(6 clock cycles not considering `complete_(data/instr)` signals)**
   i. *Fetch* Fetch Unit enables instruction load from memory. **Waits for `complete_instr` to go high before it transitions to next state.**
   ii. *Decode*. Same as above
   iii. (*Execute: Compute Memory Addresss*)Execute Unit computes address
   (`Mem_Addr1 =` PC_{mem} `+ 1 +` *`PCoffset9`*`(sign-extended)`)
   iv. (*Indirect Memory Access Read*) MemAccess Unit reads Data Memory (`Mem_Addr = DMem[Mem_Addr1`). **Waits for `complete_data` to go high before it transitions to next state.**
   v. (*Write Memory*). MemAccess Unit writes Data Memory (`DMem[Mem_Addr] ← [SR]`). **Waits for `complete_data` to go high before it transitions to next state.**
   vi. (*Update PC*)PC incremented

e. LEA **(5 clock cycles not considering `complete_instr` signal)**
   i. *Fetch* Fetch Unit enables instruction load from memory. **Waits for `complete_instr` to go high before it transitions to next state.**
   ii. *Decode*
   iii. (*Execute: Compute Memory Addresss*)Execute Unit computes address (`Mem_Addr = PC`$_{mem}$ `+ 1 +` *`PCoffset9`*`(sign-extended)`)
   iv. (*Update Register File*) : Store effective address into register file `[DR]` ← `Mem_Addr`
   v. (*Update PC*): PC incremented

It is important to note that the "`complete_data`" signal is used to wait for a successful read/write to memory. The "`invalid state`" should never be reached.

FETCH:

This block forms the interface to the testbench environment for accessing the instructions for the LC-3 to execute. The Top level block diagram of the Fetch is shown below:



Inputs

- `clock, reset`[1bit]:
- `br_taken`[1bit]: this signal tells the fetch block that a control signal has been encountered and hence the next instruction to be executed does not come from PC+1 but the target address computed by the execution of the control instruction (`taddr`). **Ignore for current project.**
- `taddr`[16 bits]: this is the value of the target address computed by a branch or jump instruction which would be loaded to the PC in case of a successful branch or Jump. **Ignore for current project.**
- **`enable_updatePC`** [1bit]: This signal enables the PC to change at the positive edge of the clock to either PC+1 or taddr based on br_taken. If zero, the PC should remain unchanged.

- **`enable_fetch`**[1bit]: This signal allows for fetch to take place i.e. IMem[PC] to happen. If this is low, then reading of the Instruction Memory should not be allowed.

Outputs

- `instrmem_rd` [1 bit] signal to indicate to the Memory that a read is to be performed, rather than a write. This signal should be high when fetch is enabled and is asynchronous.
- `pc`[16 bits]**:** the current value of the program counter
- `npc` [16 bits] should always be `pc+1` **(asynchronous)**.

On reset, at the positive edge of the clock, `pc = 16'h3000` and hence asynchronously `npc = 16'h3001`.

Signals of Importance in the fetch block for this project are
a) `enable_fetch` and `enable_updatePC` given that they control the execution of the fetch block and the instruction read b) `PC` = Program Counter of instruction being fetched and hence `npc = PC + 1` c) `instrmem_rd` = enable for Instruction Memory Read.

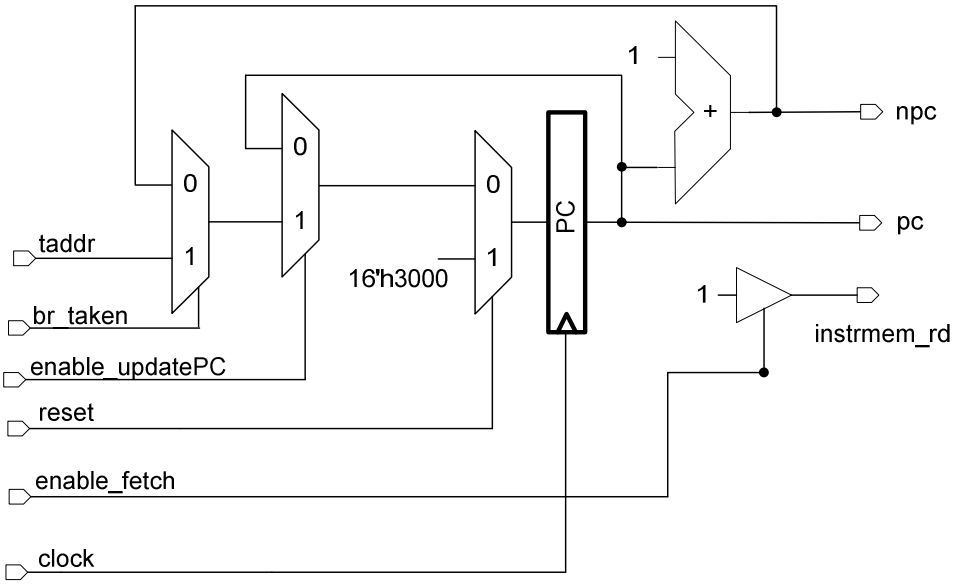
For this project please ignore the following signals

- `br_taken` = if `1 PC = taddr` else `PC = PC+1`
- `taddr` = new `PC` if branch is taken

Important operational notes:

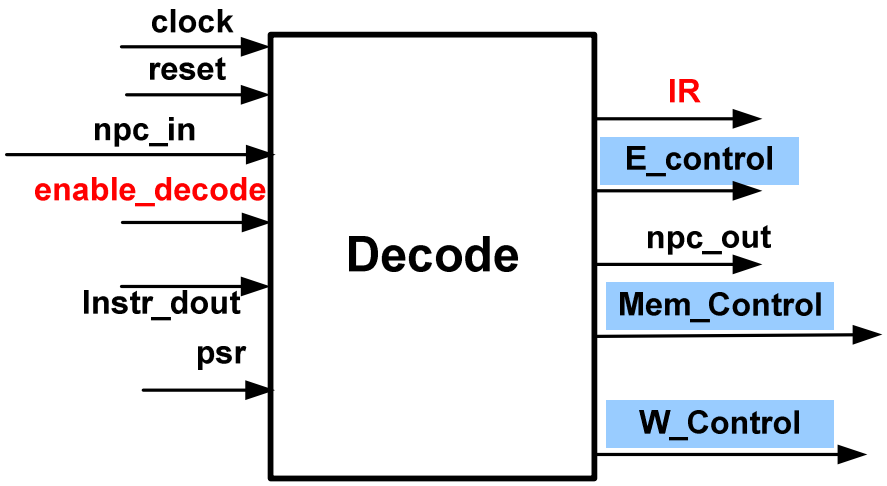
- if `enable_fetch = 1 instrmem_rd = 1` else `instrmem_rd = Z (HIGH IMPEDANCE)`
- `PC` and hence (asynchronously) `npc` updated only when `enable_updatePC = 1`

The details of this block are shown below:



DECODE

The aim of the decode block is to create the relevant control signals for a given instruction i.e. the contents of the data read out of the instruction memory. These control signals flow through the pipeline to influence the configuration of the datapath at each pipeline stage. The top level block diagram of this block is shown below:



Inputs

- `clock, reset` [1 bit]**:**
- `Instr_dout` [16 bits]**:** this signal comes from Instruction memory and contains the instruction to be used to do the relevant operations in "*Decode*" state.
- `npc_in`**[**16 bits**]:** This corresponds to the `npc` value from the Fetch stage which needs to be passed along the pipeline to its final consuming block i.e. the Execute block.
- `psr`**[**3 bits**]:** The PSR values reflect the status (Negative, Zero, Positive) of the value written (or to be written in case write back to the register has not been issued yet) into the register by the most recent register varying instruction (ALU or Loads). **Ignore as input for current project.**
- `enable_decode` [1 bit]: When 1, this signal allows for the operation of the decode unit in normal mode where it creates the relevant control signals at the output based on the input from the Instruction Memory. If 0, the block stalls with no changes at the output.

Outputs: **All outputs are created at the positive edge of the clock when** `enable_decode = 1`

- `IR`[16 bits] : This is equal to `Instr_dout`
- `npc_out`[16 bits]: This signal reflects the value of `npc_in`
- `E_control` [6 bits] : This signal controls the Execute unit. It allows for the choice of the right input values for the ALU within the Execute and also controls the type of operation that is going to be performed.

- `W_control` [2 bits] This signal determines the right choice between the flowing for a write to the register file
    - the output from an ALU operation (for alu operations)
    - the output from a PC relative operation (for LEA) and
    - the output from memory (for loads)
- `Mem_control` [7 bits]: this enables the selection of the right set of states for memory based operations. **Ignore for current project.**

Signals of Importance in the Decode block for this project are
a) `enable_decode`: master enable b) `E_Control` = Execute Block Control c) `W_Control` = Writeback Block Control d) `IR` = Instruction Register: Reflects contents of `Instr_dout` e) `npc_out`: reflects contents of `npc_in`

For this project, ignore `Mem_Control` given that this control signal is of consequence only for memory related operations.

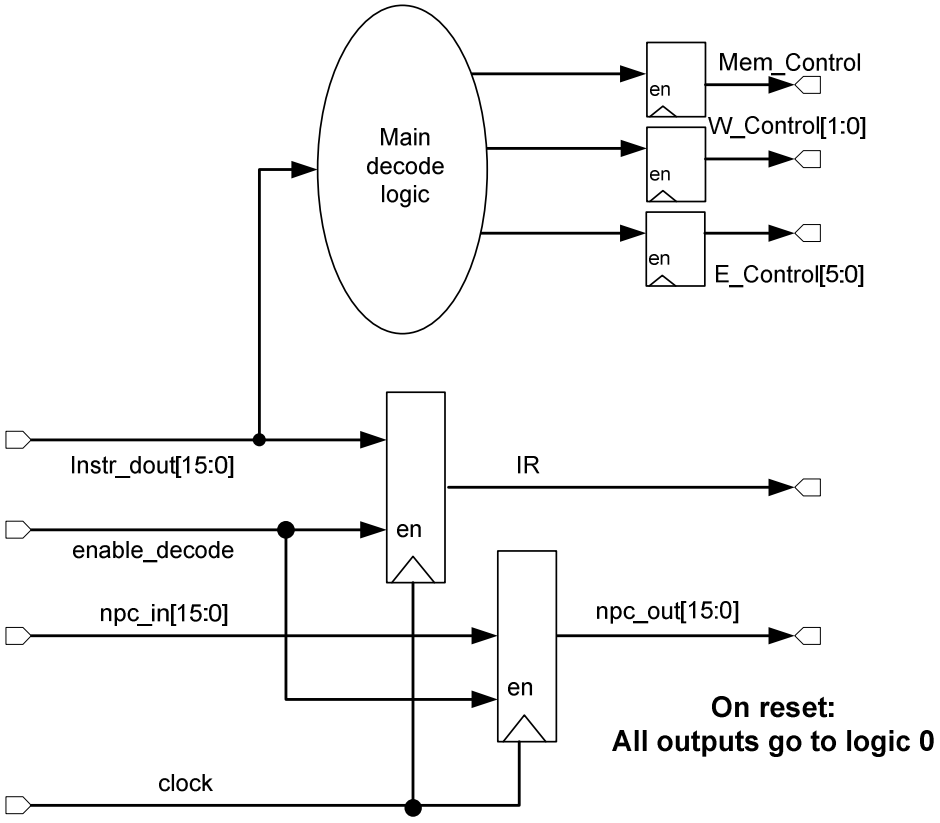
The `W_Control` signal, as stated earlier, controls the Writeback and is a function of `IR[15:12]`. We shall focus only on ALU and LEA instructions and hence `W_control` would either be `0` (ALU) or `2` (LEA). For the sake of completion, the comprehensive table of values for the `W_Control` signal is shown below:

Operation	mode	**W_Control**
ADD	0	**0(aluout)**
	1	**0(aluout)**
AND	0	**0(aluout)**
	1	**0(aluout)**
NOT		**0(aluout)**
BR		0
JMP		0
LD		**1(memout)**
LDR		**1(memout)**
LDI		**1(memout)**
LEA		**2(pcout)**
ST		0
STR		0
STI		0

The `E_Control` signal is the concatenation of {`alu_control, pcselect1, pcselect2, op2select`} in that order and takes the following values.

IR[15:12]	IR[5]	E_Control			
		alu_control [2]	**pcselect1 [2]**	**pcselect2 [1]**	**op2select [1]**
ADD	0	**0**	-	-	VSR2 **(1)**
	1	**0**	-	-	imm5 **(0)**
AND	0	**1**	-	-	VSR2 **(1)**
	1	**1**	-	-	imm5 **(0)**
NOT		**2**	-	-	-
BR		-	offset9 **(1)**	npc **(1)**	-
JMP		-	0 **(3)**	VSR1 **(0)**	-
LD		-	offset9 **(1)**	npc **(1)**	-
LDR		-	offset6 **(2)**	VSR1 **(0)**	-
LDI		-	offset9 **(1)**	npc **(1)**	-
LEA		-	offset9 **(1)**	npc **(1)**	-
ST		-	offset9 **(1)**	npc **(1)**	-
STR		-	offset6 **(2)**	VSR1 **(0)**	-
STI		-	offset9 **(1)**	npc **(1)**	-

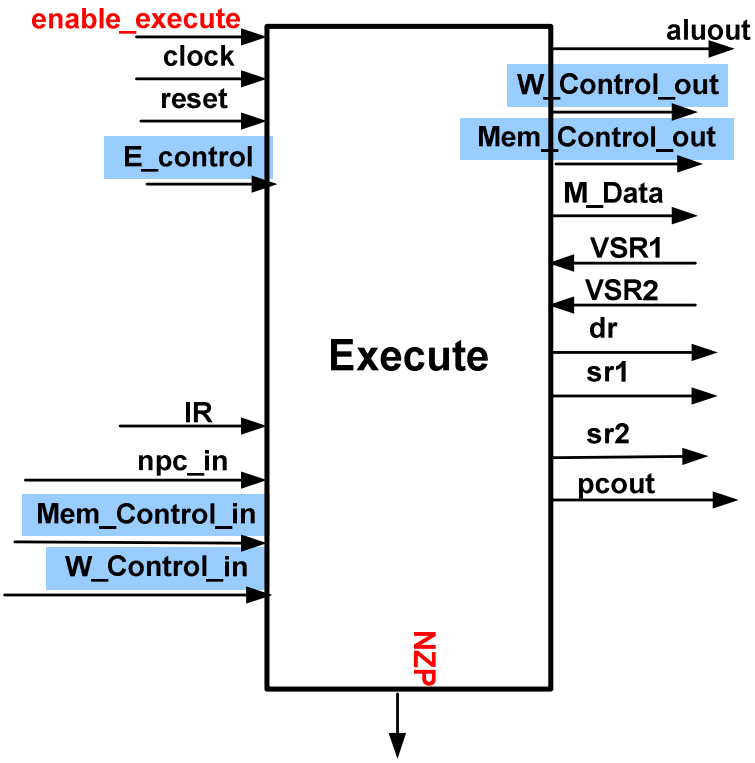
The internal details of the Decode block are shown below:



EXECUTE

This block forms the heart of the LC3 microcontroller where data corresponding to a given instruction is manipulated. The type of manipulation and the type of data to be used is a function of the E_Control signal. Moreover, this block is very closely coupled with the Writeback unit where it gets all its data from i.e. the contents corresponding to SR1 &/ SR2. Also, the manipulations corresponding to PC related operations for LEA (and other Memory and Control operations for that matter) are also performed in this block. The top level block diagram and the inputs and outputs are listed below.

The inputs and outs of the block are:



Inputs:

- `clock, reset`[1 bit]
- `E_control`: [6 bits]This signal has already been explained in the Decoder section.
- `IR:` [16 bits]The instruction register that is used to create the offset values for PC based operations.
- `npc_in`: [16 bits]The `npc` that was passed along the pipeline from the Decode stage corresponding to `1+ PC` that gave the `IR`.
- `VSR1, VSR2:` [16 bits] These values come from the Writeback unit based on the `sr1` and `sr2` outputs. **These values are asynchronously read from the register**

file in the writeback unit. Therefore, if the outputs `sr1 = 4` and `sr2 = 5` for the execute unit then `VSR1 = RegFile[4]` and `VSR2 = RegFile[5]`.

- `W_Control_in`[2 bits], `Mem_Control_in`[7 bits]: These are the control signals created at the Decoder stage that need to be passed along the pipeline to the Writeback and Memory Access blocks (with controller). **Ignore** `Mem_Control_in` **for current project.**
- `enable_execute:` [1 bit]This signal is the master control for the entire Execute block. All the outputs are created only when this signal is enabled. If zero, all the outputs should hold their previous value.

Outputs: **except for SR1 and SR2 all outputs are created synchronously only when enable_execute = 1. SR1 and SR2 are asynchronously created as a function of IR[15:12].**

- `W_Control_out`[2 bits], `Mem_Control_out`[7 bits]: They equal the values of the `W_Control_in` and `Mem_Control_in` . **Ignore** `Mem_Control_out` **for current project.**
- `aluout`: [16 bits]This is the result of ALU operations which take the form
  - ← `SR1 (OPERATION) SR2/sxt(imm5)` for AND and ADD or
  - ← `(NOT) SR1` (NOT operation)

  Synchronously created.
- pcout: [16 bits] This is the result of either
  - `pc + 1 + sxt(offset)` (Branch, LD, ST, LDI, STI, LEA) or
  - `[BaseR] + sxt(offset)` (LDR, STR)

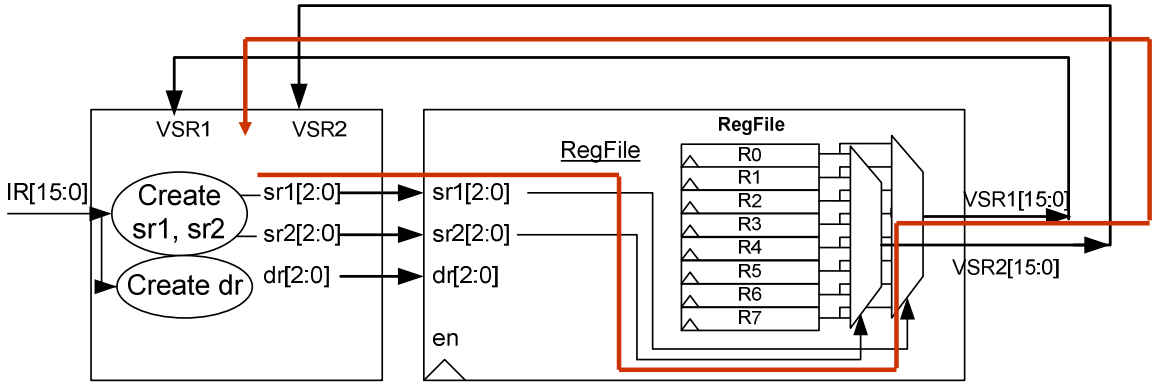
  Synchronously created.
- `dr`: [3 bits] The destination address for the instruction that came into the Execute which is of relevance for loads and ALU operations. It should be zero for all other types of incoming instructions. **Synchronously created.**
- `sr1`: [3 bits]This reflects the contents of `IR[8:6]` for all instruction types. **Asynchronously created.**
- `sr2`: [3 bits] This reflects the contents of `IR[2:0]` for ALU instructions, `IR[11:9]` for stores and is zero for all other types. **Asynchronously created.**
- `NZP`[3 bits]: This signal reflects the NZP requirements of the Control instructions at locations IR[11:9]. The aim is to use it to determine whether the branch conditions is satisfied. **Ignore for current project.**
- `M_data`[16 bits]: This is the value that needs to be written to Memory for Loads and Stores. **Ignore for current project.**

The type of operation executed to create `aluout` and `pcout` is based on the `E_Control` value which in itself is based on the type of instruction. **Also, on reset, all synchronous outputs of the Execute block go to 0.**

Again, please ignore `M_data, NZP, Mem_Control_in` **and** `Mem_Control_out` **for this project.**

Given that this is, at present, an un-pipelined microcontroller, we have it a little easy. Traditional pipelined processing units have to deal with dependencies between instructions. We are going to deal with this later in the course. For now, we can assume complete independence.

An important aspect of the design that needs to be appreciated is the asynchronous interaction between the Execute and Writeback blocks. This interaction is based on the creation of `SR1` and `SR2` values asynchronously from the `IR` signal that comes into the Execute. The SR1 and SR2 signals then go to the Writeback block which sends the relevant `VSR1 = RegisterFile[SR1]` and `VSR2 = RegisterFile[SR2].` This is pictorially represented in the figure below:



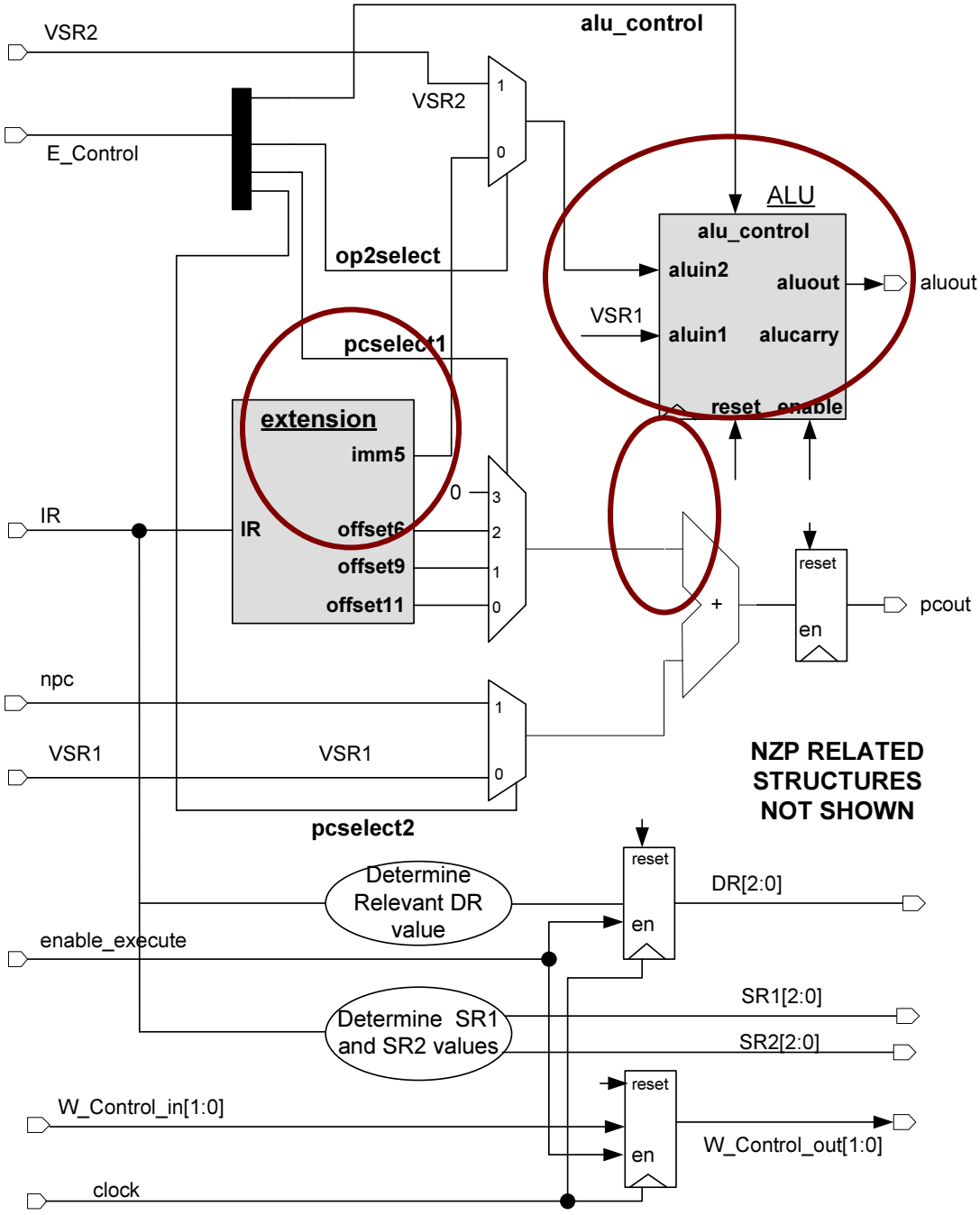
The internal details of the Execute block are shown below:

The extension within the Execute creates the following sign-extended signals in a combinational manner to be used with the relevant instructions:

- `imm5 = {11{IR[4], IR[4:0]}`
- `offset6 = {10{IR[5], IR[5:0]}`
- `offset9 = {7{IR[8], IR[8:0]}`
- `offset11 = {5{IR[10], IR[10:0]}`

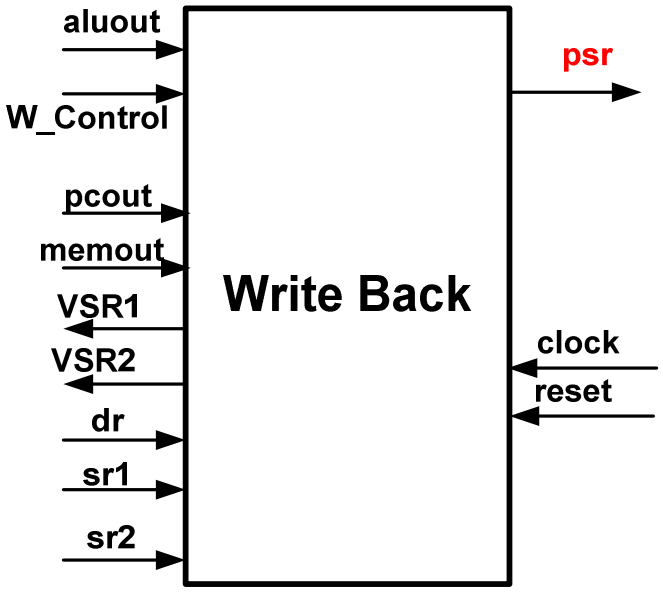
The importance of the `E_Control` comes from the recognition that the `pcselect1`, `pcselect2` and `opselect` signals reconfigure the signal flow within the execute block. This reconfiguration causes the relevant inputs to be sent into the ALU and the computation unit for `pcout`. Also, the `alu_control` part of the `E_Control` signal controls the type of operation that will be performed on the data coming into the ALU (`aluin1 and aluin2`). Again, it is good to remember that we are only going to be dealing with following variations in operations for this project.

ALU ([DR]←[SR1] **aluop** [SR2])
ALU ([DR]←[SR1] **aluop** **sxt**(Imm5))
LEA ([DR] ← PC_{mem}+1+**sxt**(PCOffset9))


VSR2
alu_control
VSR2
1
0
E_Control
ALU
alu_control
op2select
aluin2
aluout
aluout
VSR1
pcselect1
aluin1
alucarry
reset
enable
extension
imm5
0
3
IR
IR
offset6
2
offset9
1
offset11
0
reset
+
pcout
en
npc
1
VSR1
VSR1
0
NZP RELATED
STRUCTURES
NOT SHOWN
pcselect2
reset
Determine
Relevant DR
value
DR[2:0]
en
enable_execute
SR1[2:0]
Determine SR1
and SR2 values
SR2[2:0]
reset
W_Control_in[1:0]
W_Control_out[1:0]
en
clock

WRITEBACK

The Writeback unit contains the register file which provides the relevant data values for register based operations. Hence it also controls the values that need to be written into the Register file. As already stated in the previous section this block works closely with the Execute Unit to provide the `VSR1` and `VSR2` signals needed to perform most operations in the Execute Unit. The top level block diagram and the inputs and outputs of this block are shown below:



Input and outputs of the design are:

Inputs

- `clock, reset`
- `npc`[16 bits]
- `W_control_in` [2 bits]
- `aluout` [16 bits]: value from Execute for ALU operations
- `pcout` [16 bits]: value from Execute corresponding to PC based operations for LEA
- `memout`[16 bits]: Values read from memory **Please ignore for this project.**
- `enable_writeback`[1 bit]: Enable signal to allow for operation of the Writeback block. If zero, the register file is not written to.
- `sr1, sr2` [3 bits]: Source register addresses 1 and 2 for Execute operations to be performed.
- `dr`[3 bits]: Destination register address where the data chosen using `W_Control` value is written to in the Register File.

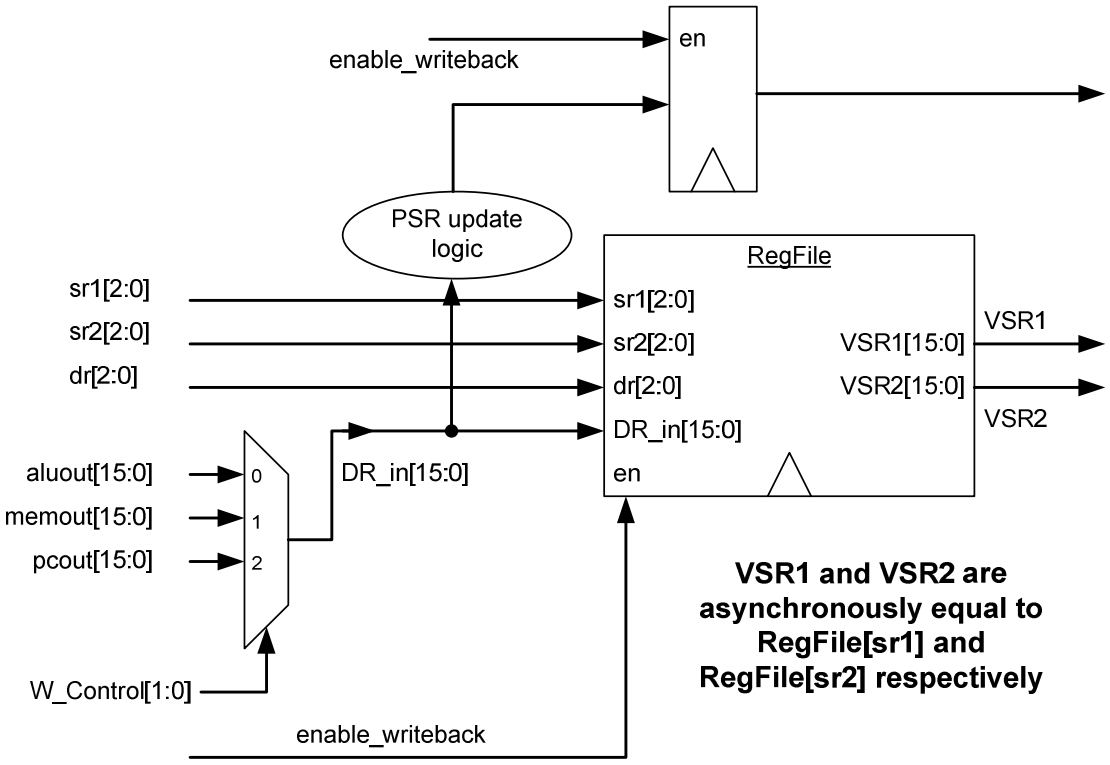
Output:

- `VSR1, VSR2` [16 bits] Value returned from the register file **(ASYNCHRONOUS)** corresponding to `RegFile[sr1] and RegFile[sr2]`
- `psr` [3 bits] The status register which provides the negative, zero and positive flags for the latest value written to the Register File in the Writeback block.

The `psr` register is encoded based on the value being written to the register file and follows the encoding `psr[2]` = 1 for negative values, `psr[1]` = 1 for values equal to 0 and `psr[0]` = 1 for positive values. Thus

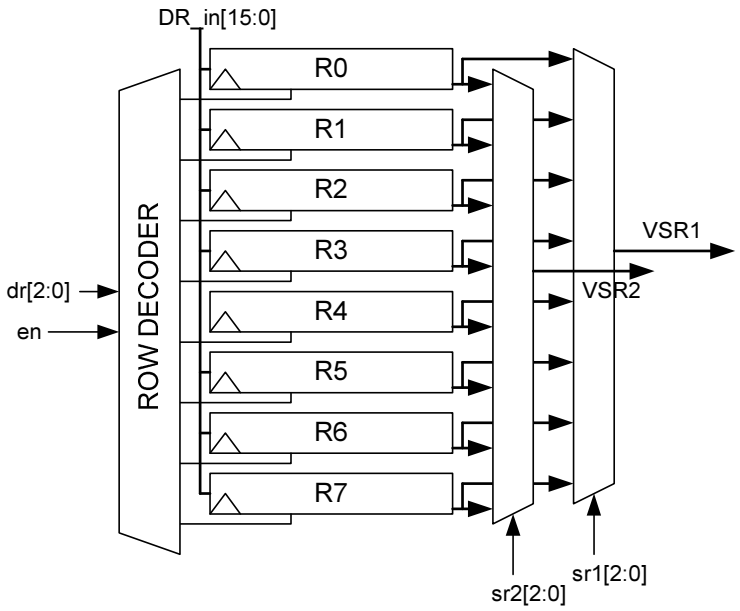
- if we are writing `16'hfff2` to the register file we would have `psr = 3'b100`,
- if we are writing `16'h00f2` to the register file we would have `psr = 3'b001`,
- if we are writing `16'h0000` to the register file we would have `psr = 3'b010`,

The internal details of this block are shown below:



Note the use of the W_Control signal to determine the data that needs to be written into the register file. The write to the register file would take the form `RegisterFile[dr] ← DR_in`

The Register File takes the form shown below:



On reset IR = 0 and hence sr1 = sr2 = 0. This implies that VSR1 and VSR2 would have xx's initially. This needs to be kept in mind during testing. It is for the very reason that it is suggested that any testing begin by initialization of the register to a known state.

CONTROLLER:

Inputs:

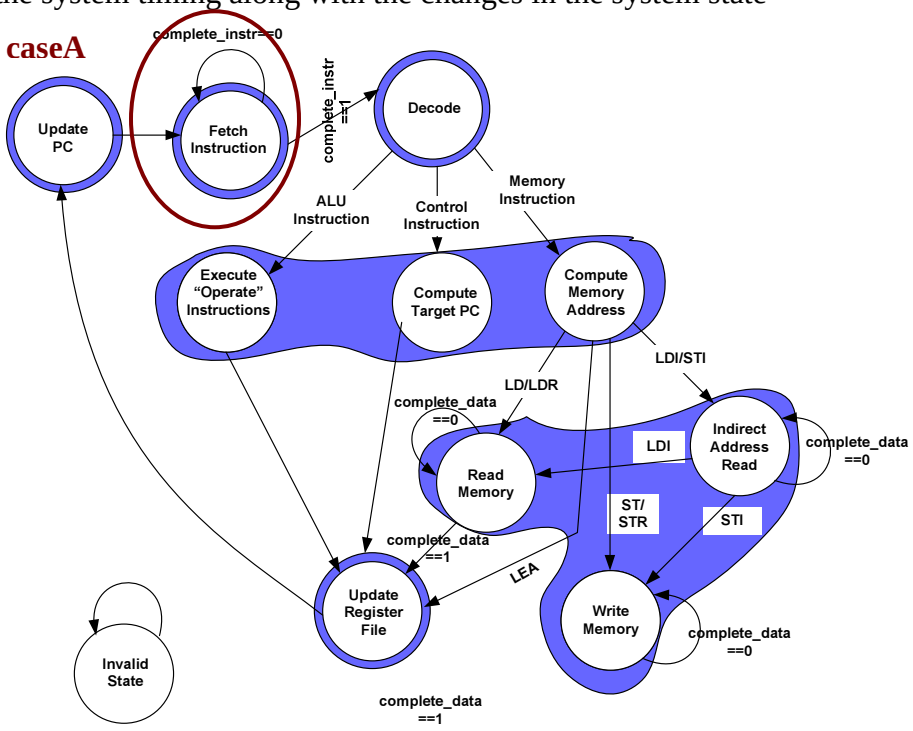
- `complete_data`: This signal comes from the Data Memory which tells the controller that the Memory data that the memory access block is waiting on is present at output of DMem. **Please ignore for this project.**
- `complete_instr`: This signal comes from the Instruction Memory which tells the controller that the instruction that the memory access block is waiting on is present at the output of IMem.
- `clock, reset:`
- `IR`: This signal comes from the Decode and allows the controller to detect the instruction type and the presence of dependencies and stall requirements. **Please ignore for this project.**
- `psr, NZP:` These signals come from the Execute and Writeback block and provide the necessary information to determine the result of branch conditions if required and hence the determination of the `taddr` and `br_taken` signals which would update the `PC`.

Outputs

- `enable_updatePC`: Enables the updating of the PC to its next value (present `PC + 1` or `taddr` @ Fetch block)
- `enable_fetch:` Enables the reading of the Instruction Memory by the Fetch unit. If this signal is high, `instrmem_rd` should also go high asynchronously.

- `enable_decode`: Enables the Decoder and the creation of all of the decoder outputs corresponding to the value at `Instr_dout`
- `enable_execute`: Enables the Execute and the creation of all of but `sr1` and `sr2` signals at the execute outputs.
- `enable_writeback`: Enables writing back to the register file and the creation of the PSR values.
- `br_taken:` This is used to choose between `taddr` or `PC+1` for the result of a control instruction. **Please ignore for this project.**
- `mem_state`: Enables proper operation of the memory access block by moving between the right read and write memory states such that variants of loads and stores are performed correctly. **Please ignore for this project.**

There are other signals which will be introduced to provide bypasses for pipeline dependencies and to deal with stalls. Signals also need to be introduced to deal with instructions that take longer or less than 5 clock cycles. We shall be dealing with these variations in the coming projects. To appreciate the operation of this block, it is important to view the system timing along with the changes in the system state

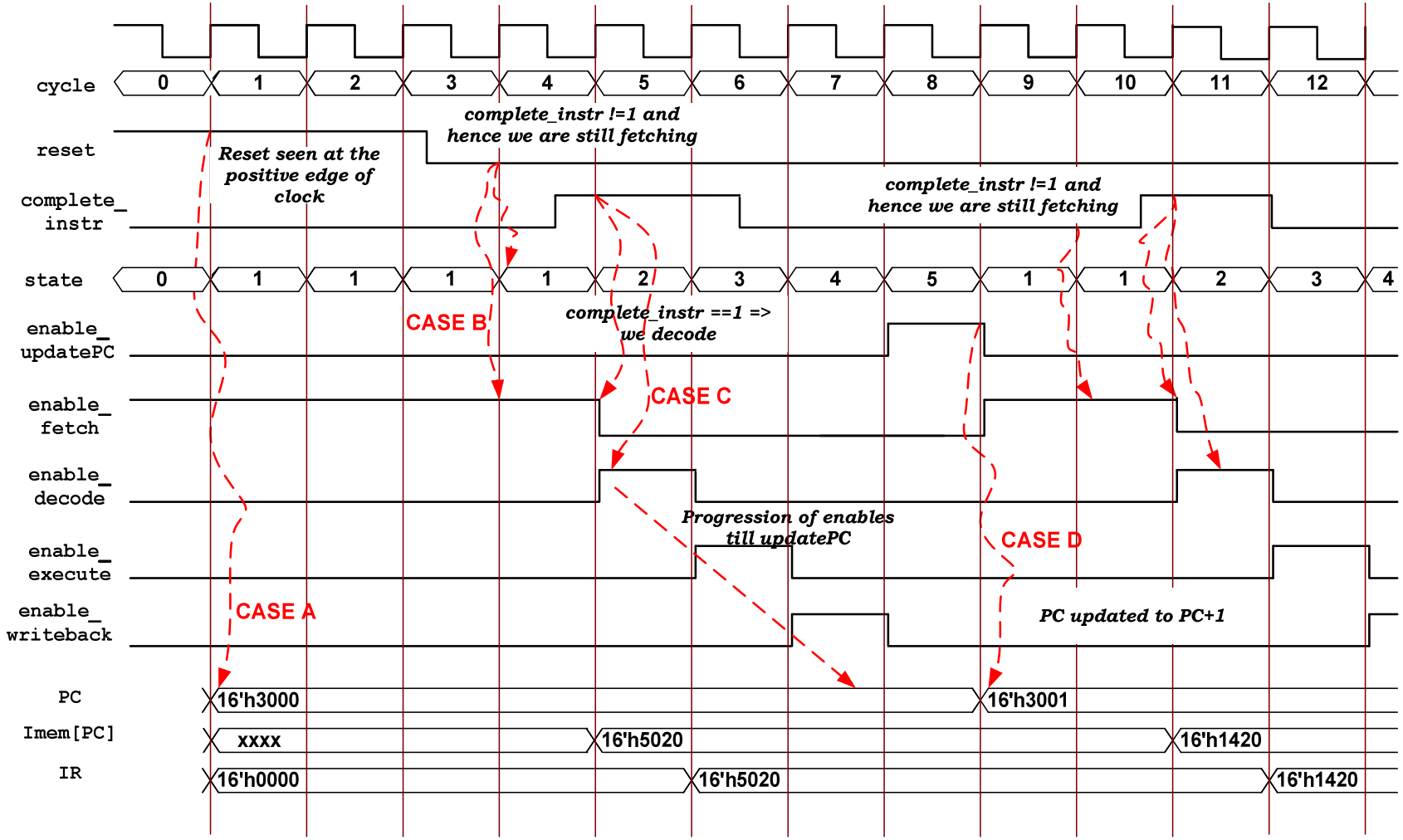


For this project, we are going to ignore the `complete_data` signal but we are still interested in the `complete_instr` signal that controls the fetch completion as shown in case A. Also, it is to be noted that for the ALU operations and the LEA instruction, the states that will be visited are

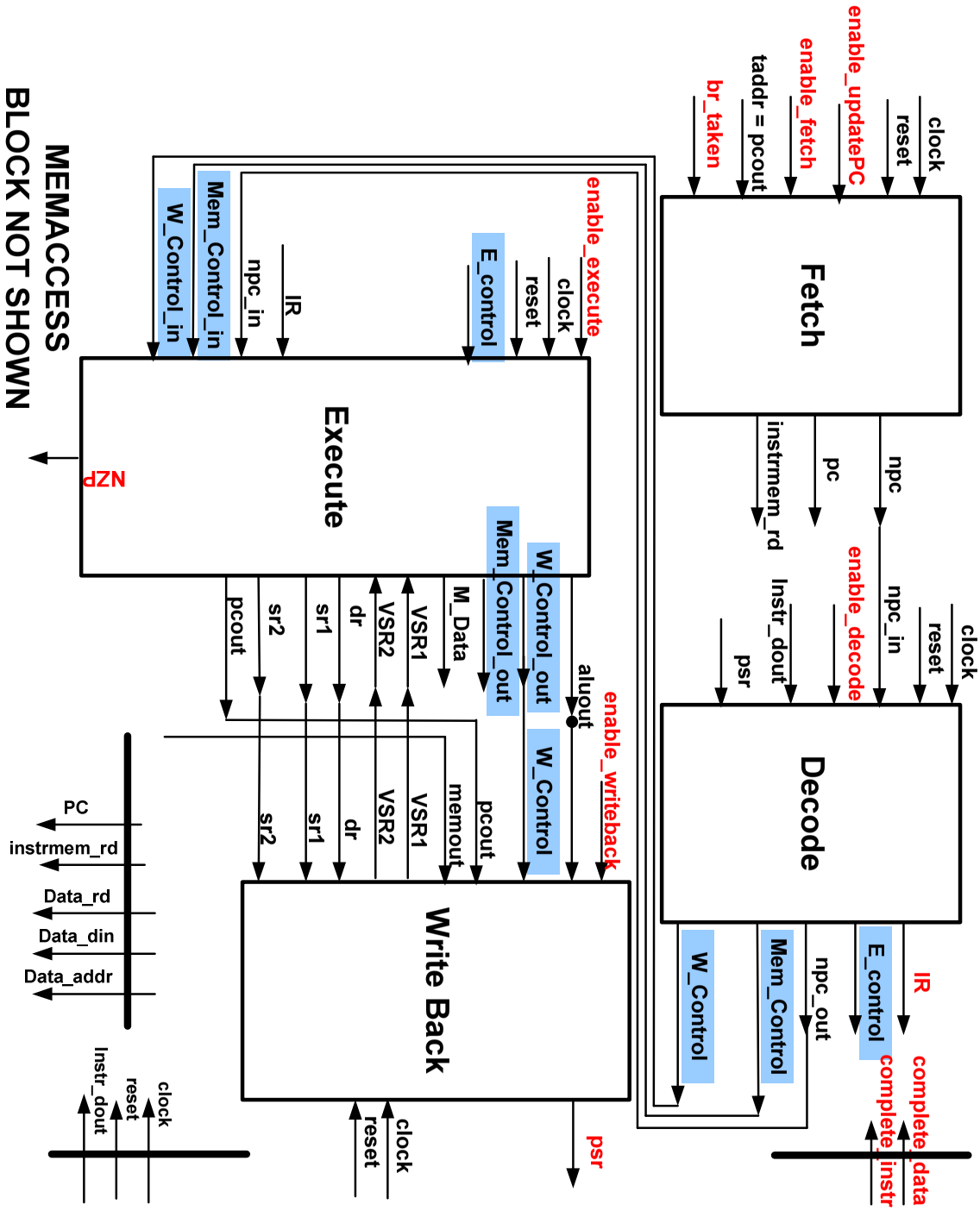
ALU: Fetch → Decode → **Execute "Operate" Instructions** → Update Register File → Update PC

LEA: Fetch → Decode → **Compute Memory Address** → Update Register File → Update PC

The reset operation, the flow of enables and the sensitivity to the `complete_instr` signal are best captured by the timing diagram shown below:


cycle
0
1
2
3
4
5
6
7
8
9
10
11
12
reset
complete_instr !=1 and hence we are still fetching
Reset seen at the positive edge of clock
complete_instr
complete_instr !=1 and hence we are still fetching
state
0
1
1
1
1
2
3
4
5
1
1
2
3
4
CASE B
complete_instr ==1 => we decode
enable_ updatePC
CASE C
enable_ fetch
enable_ decode
Progression of enables till updatePC
CASE D
enable_ execute
CASE A
PC updated to PC+1
enable_ writeback
PC
16'h3000
16'h3001
Imem[PC]
xxxx
16'h5020
16'h1420
IR
16'h0000
16'h5020
16'h1420

The controller transitions from one state to the next based on the values of the incoming signals, here `complete_instr`, at the positive edge of the clock. This becomes very important in terms of testing not just the controller but the entire system since it is the control that determines the operation of each. The top level stitching of the blocks is shown below. **More details after the figure ….**



In the controller timing diagram, **Case A** corresponds to the response of the system at the positive edge of the clock to a positive reset. It is seen that the `PC` goes to `16'h3000` and the `IR` goes to 0. **Case B** represents the response of the system to the lack of a `complete_instr` signal after the reset is de-asserted. The system stalls on a fetch until the `complete_instr` signal is a 1 at the positive edge of the clock as seen in **CaseC**. At this point the enables ripple through the system to perform decoding, execution, writeback and a PC update. **CaseD** shows the update of the PC to PC+1 after which the system again waits, in this case, for a `complete_instr` and the whole process is repeated. It is important to note that there would no stall in fetch if the `complete_instr` was 1 when the LC3 was fetching i.e. at the clock edge when we transition from cycle 9 to 10.

To better understand the progression of the signals within the LC3, it is best to look at the execution of a few instructions in greater detail to know what to expect. In the examples below we are going to be looking at the inputs and outputs of interest in each block. Special focus must be kept on the control signals and the result from the Execution unit. Moreover, important signal changes are shown in bold. The 4 instructions are:

```
@3000:      5020  (AND  R0    R0    #0);
@3001:      1422  (ADD  R2    R0    #2);
@3002:      1280  (ADD  R1    R2    R0);
  @3003:      EDFE  (LEA  R6    #-2);
```

INSTRUCTION 1 @3000: 5020 (AND R0 R0 #0);

FETCH (after reset has gone to 0)
in: enable_fetch = 1 ; pc = **3000** ; npc = **3001** ;
out: instrmem_rd = 1 ;

DECODER
in: Instr_dout = **IMem**[3000] = **5020** ; npc_in = **3001** ;
enable_decoder = 1 ;
out: npc_out = **3001**; IR = **5020** ; W_Control = **0** (aluout) ;
alu_control = **1**; pcselect1 = 0; pcselect2 = 0; op2select = **0** (imm5)
=> E_Control = **[6'b01_0000]**

EXECUTE
in: E_Control = **[6'b01_0000]**; W_Control_in = 0 ; npc_in = **3001** ;
IR = **5020** ; enable_execute = 1 ;
VSR1 = **R0 = xxx** ; VSR2 = **R0 = xxx** ;
out sr1 = 0; sr2 = 0 ;
dr = 0 ; W_Control_out = 0 ; aluout = **VSR1 & sxt(Imm5) = R0&0 = 0**

WRITEBACK
in: dr = **0** ; W_Control = 0 ; aluout = 0 ; enable_writeback = 1 ;
out: **RegFile[dr] = aluout = 0** => **R0 = 0**

Combinational relationship between sr1, sr2 , VSR1 and VSR2

UPDATEPC
in: enable_updatePC = 1 ; **out:** pc = **3001** ; npc = **3002**

INSTRUCTION 2 @3001: 1422 (ADD R2 R0 #2);

FETCH
in: enable_fetch = 1 ; pc = **3001** ; npc = **3002** ;
out: instrmem_rd = 1 ;

DECODER
in: Instr_dout = IMem[**3001**] = **1422** ; npc_in = **3002** ;
enable_decoder = 1 ;
out: npc_out = **3002**; IR = **1422** ; W_Control = **0** (aluout) ;
alu_control = **0**; pcselect1 = 0; pcselect2 = 0; op2select = **0** (imm5)
=> E_Control = **[6'b00_0000]**

EXECUTE
in: E_Control = **[6'b00_0000]**; W_Control_in = 0 ; npc_in = **3002** ;
IR = **1422** ; enable_execute = 1 ;
VSR1 = **R0 = 0** ; VSR2 = **R2 = xxx** ;
out: sr1 = 0; sr2 = 2 ;
dr = **2** ; W_Control_out = 0 ; aluout = **VSR1 + sxt(Imm5) = 0 + 2 = 2**

WRITEBACK
in: dr = **2** ; W_Control = 0 ; aluout = **2** ; enable_writeback = 1 ;
out: **RegFile[dr] = aluout = 2** => **R2 = 2**

UPDATEPC
in: enable_updatePC = 1 ; **out:** pc = **3002** ; npc = **3003**

INSTRUCTION 3 @3002: 1280 (ADD R1 R2 R0);

FETCH
in: enable_fetch = 1 ; pc = **3002** ; npc = **3003** ;
out: instrmem_rd = 1 ;

DECODER
in: Instr_dout = IMem[**3002**] = **1280** ; npc_in = **3003** ;
enable_decoder = 1 ;
out: npc_out = **3003**; IR = **1280** ; W_Control = **0** (aluout) ;
alu_control = **0**; pcselect1 = 0; pcselect2 = 0; op2select = **1** (VSR2)
=> E_Control = **[6'b00_0001]**

EXECUTE
in: E_Control = **[6'b00_0001]**; W_Control_in = 0 ; npc_in = **3003** ;
IR = **1280** ; enable_execute = 1 ;
VSR1 = **R2 = 2** ; VSR2 = **R0 = 0** ;
out: sr1 = 2; sr2 = 0 ;
dr = **1** ; W_Control_out = 0 ; aluout = **VSR1 + VSR2 = 2 + 0 = 2**

WRITEBACK
in: dr = **1** ; W_Control = 0 ; aluout = **2** ; enable_writeback = 1 ;
out: **RegFile[dr] = aluout = 2** => **R1 = 2**

UPDATEPC
in: enable_updatePC = 1 ; **out:** pc = **3003** ; npc = **3004**

INSTRUCTION 4 @3003: EDFE (LEA R6 #-2);

FETCH
in: enable_fetch = 1 ; pc = **3003** ; npc = **3004** ;
out: instemem_rd = 1 ;

DECODER
in: Instr_dout = IMem[**3003**] = **EDFE**; npc_in = **3004** ;
enable_decoder = 1 ;
out: npc_out = **3004** ; IR = **EDFE**; W_Control = **2** (pcout) ;
alu_control=0; pcselect1 = **1(offset9)**; pcselect2 = **1(npc)**; op2select = 0
=> E_Control = **[6'b00_0110]**

EXECUTE
in: E_Control = **[6'b00_0110]**; W_Control_in = 2 ; npc_in = **3004** ;
IR = **EDFE**; enable_execute = 1 ;
VSR1 = **R7 = xxx** ; VSR2 = **R6 = xxx** ;
out: sr1 = **7**; sr2 = **6** ;
dr = **6** ; W_Control_out= **2** ; pcout = **npc + sxt(offset9) =3004+ FFFD = 3002**

WRITEBACK
in: dr = **6** ; W_Control = **2** ; **pcout** = **3002** ; enable_writeback = 1 ;
out: **RegFile[dr] = pcout = 3002** => **R6 = 3002**

UPDATEPC
in: enable_updatePC = 1 ; **out:** pc = **3002** ; npc = **3003**

Testing Considerations

- Basic Checks
  - ALWAYS INITIALIZE REGISTERS. It is not possible to determine correctness when all the data involved is going to be don't cares. Known states are always desirable.
  - Reset Behavior (Each block has a particular behavior)
  - Completion of instructions and time taken. We must make sure that all the instructions complete and in the right number of cycles.
  - Operational Correctness i.e. results are right when an instructions are executed.
  - Register File correctness. Be sure to test different values when working with internal memories like register files and other storage elements.

- Further thoughts
  - Smart use of data values: use inputs that give you maximum information. Do not just send in random values. Always make sure you can get as much checked with a set of input values/ data values as possible.
  - Sequence of instructions: ADD → NOT → AND → ADD where the destination of one instruction is the source register for the next or another future instruction.