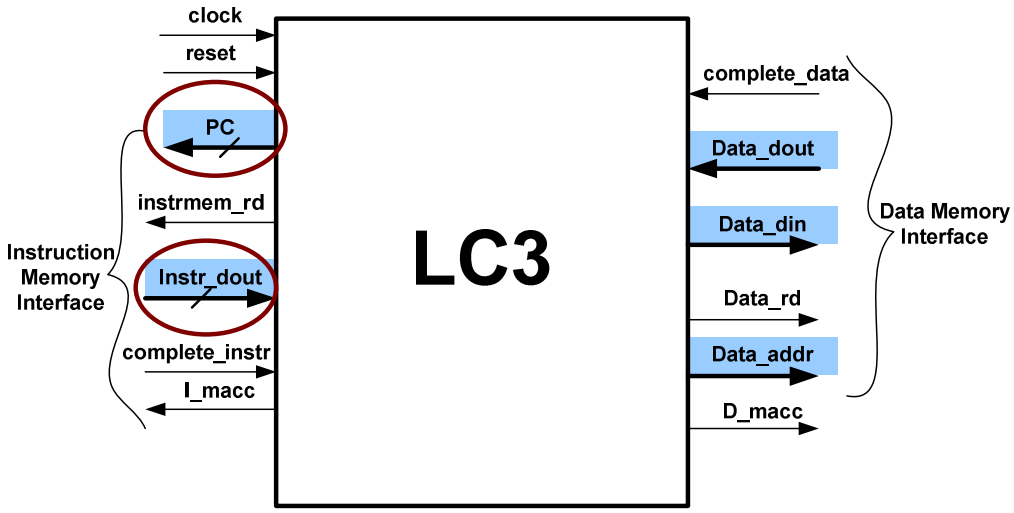
ECE 745 : ASIC VERIFICATION
PROJECT 2 LC-3 DESIGN SPEC

Introduction:
This project deals with the verification of the data and control path of a PIPELINED LC-3 microcontroller with a comprehensive instruction set. This document will take you through the implementation and specification of this microcontroller. 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.

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.
- `Data_din` **(16 bits)** Corresponds to the values that need to be written to Data Memory which would correspond to stores.
- `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.
- I_macc and D_macc : These will be used in the future projects to distinguish between Instruction and Data memory access phases.

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 12	11 9	8 6	5	4	3 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		

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 $\mathbf{PC_{mem}}$.

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

  `Mem_Addr` = $\mathbf{PC_{mem}}$ + 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 = $\mathbf{PC_{mem}}$ + 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 = $\mathbf{PC_{mem}}$ + 1 + sign-extended(*PCoffset9*)

  [DR] ← **Mem_Addr**

For example, if instruction (IR) = **16'hA7E8, $\mathbf{PC_{mem}}$ = 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 = R3f;
- **Mem_Addr1** = $\mathbf{PC_{mem}}$ + 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			12	11		9	8	6	5	4	3	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:
$\mathbf{PC_{next}} \leftarrow \mathbf{PC_{branch}} + 1 +$ sign extended*(PCoffset9)*

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

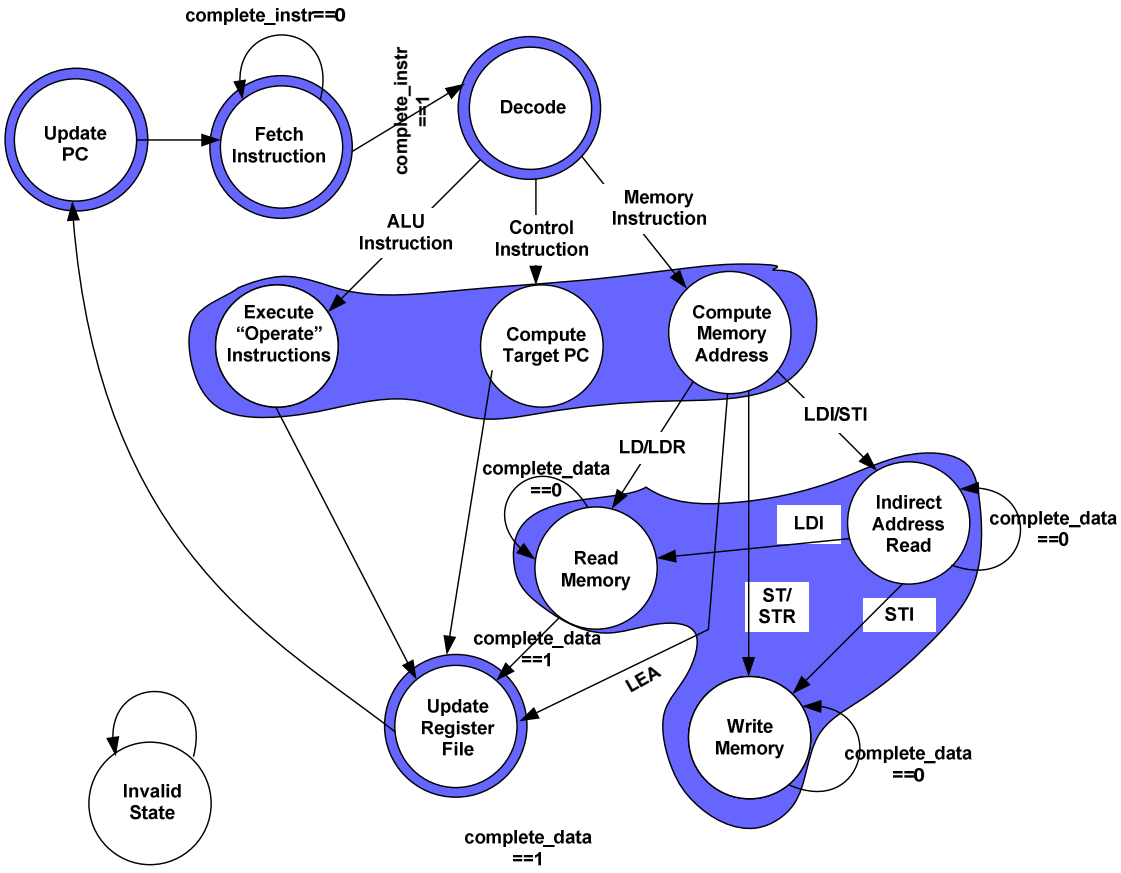
To determine whether the conditions for the branch are met or not, the result of the previous register file manipulating instruction is recorded in what is called the PSR register. The PSR register is written to in the following conditional manner:

```
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

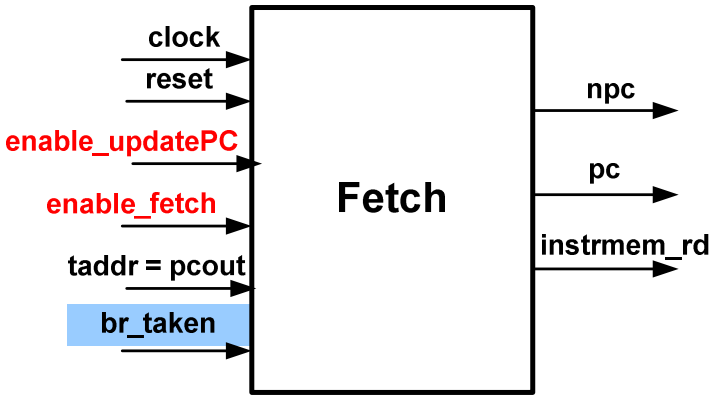
d. STI **(6 clock cycles not considering `complete_(data/instr)` signals)**
i. *Fetch* Fetch Unit enables instruction load from memory. **Waits for `complete_instr` to go high before it transitions to next state.**
ii. *Decode*. Same as above
iii. (*Execute: Compute Memory Addresss*)Execute Unit computes 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`).
- `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.
- `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`.

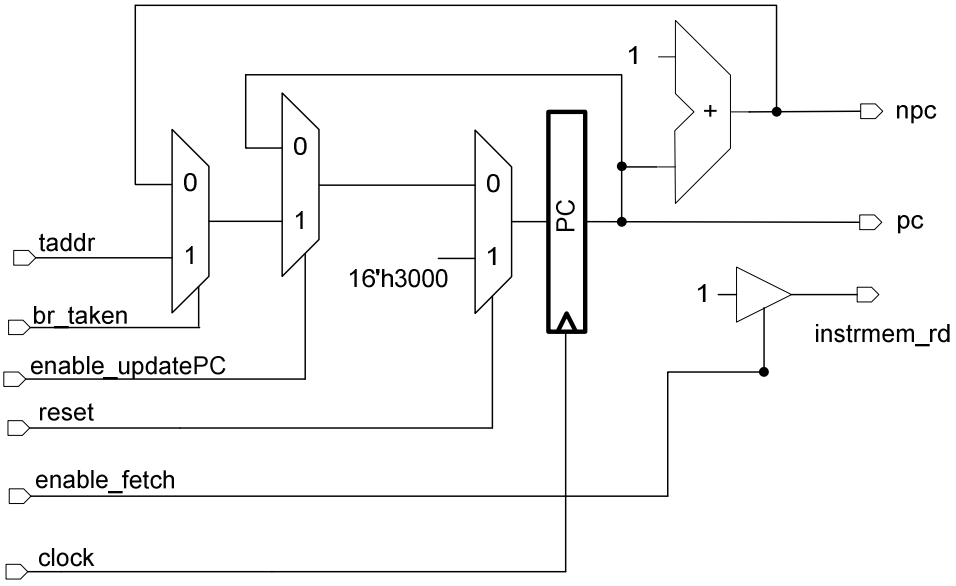
Signals of Importance in the fetch block for this project are
a) `enable_fetch` and `enable_updatePC` given that they control the execution of the fetch block and the instruction read b) `PC` = Program Counter of instruction being fetched and hence `npc = PC + 1` c) `instrmem_rd` = enable for Instruction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

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

Important operational notes:

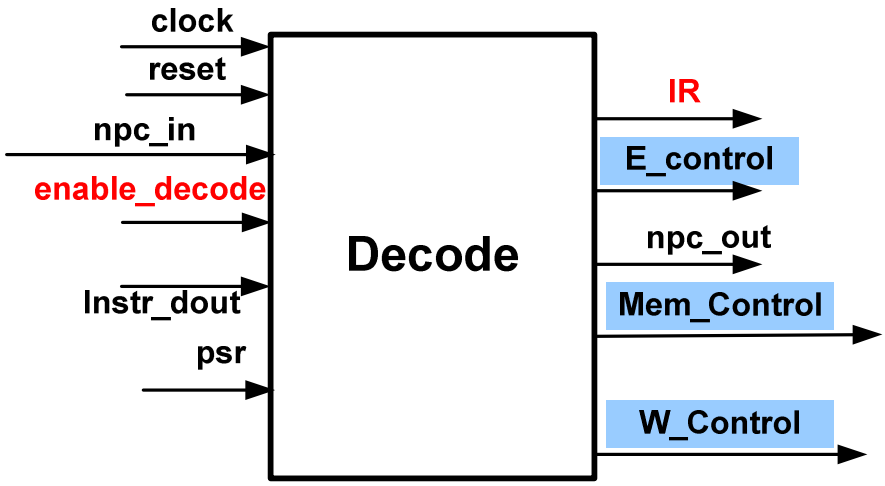
- if `enable_fetch = 1 instrmem_rd = 1` else `instrmem_rd = Z (HIGH IMPEDANCE)`
- `PC` and hence (asynchronously) `npc` updated only when `enable_updatePC = 1`

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).
- `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` [1 bit]: this enables the selection of the right set of states for memory based operations.

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`

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:

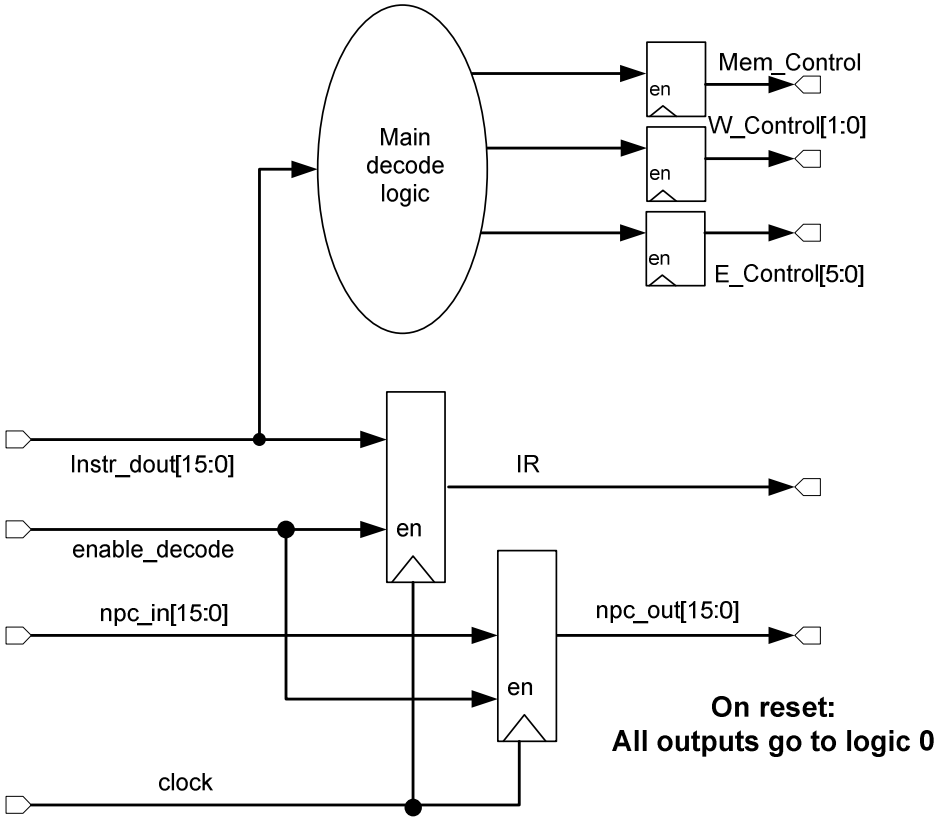
Operation	mode	**W_Control**
ADD	0	**0(aluout)**
	1	**0(aluout)**
AND	0	**0(aluout)**
	1	**0(aluout)**
NOT		**0(aluout)**
BR		0
JMP		0
LD		**1(memout)**
LDR		**1(memout)**
LDI		**1(memout)**
LEA		**2(pcout)**
ST		0
STR		0
STI		0

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)**	-

Operation	mode	**Mem_Control**
ADD	0	-
	1	-
AND	0	-
	1	-
NOT		-
BR		-
JMP		-
LD		0
LDR		0
LDI		1
LEA		-
ST		0
STR		0
STI		1

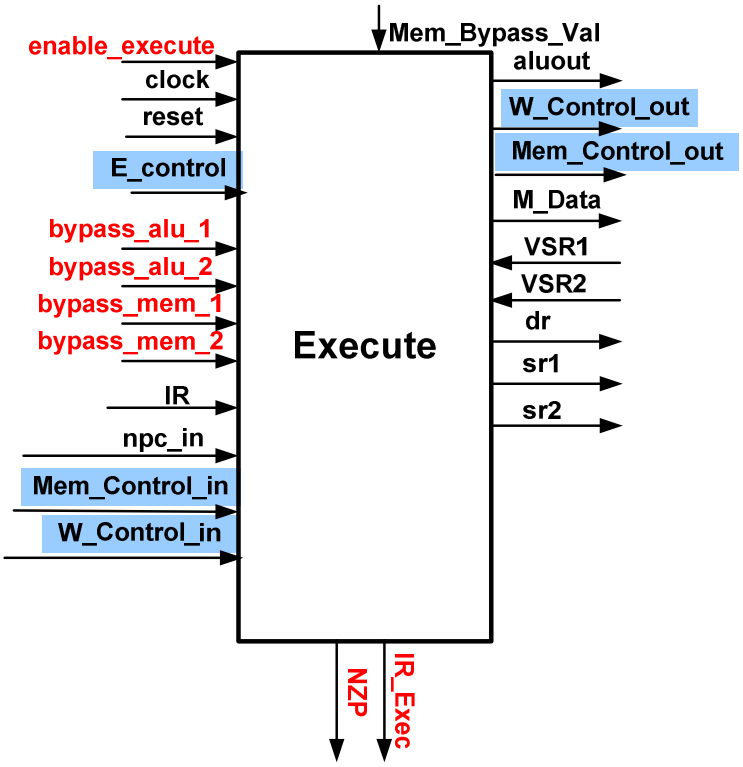
The internal details of the Decode block are shown below:



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 outputs 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`.
- `bypass_alu_1`[1 bit], `bypass_alu_2`[1 bit]: These signals allow for the use of the bypass from aluout for [SR1] and [SR2] respectively of the instruction coming into the Execute. This is better explained in a little while
- `bypass_mem_1`[1 bit], `bypass_mem_2`: [1 bit]These signals allow for the use of the bypass from `Mem_Bypass_Val` for [SR1] and [SR2] respectively of the instruction coming into the Execute. This is better explained in a little while
- `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`[1 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).

- `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.
- `Mem_Bypass_Val`: This corresponds to the Data Memory output value from the Memory Access block. This functions as a bypass value when the result is needed before it is written to the register file. A typical example is when you have a Load writing to, say, R5 and the next instruction is an ALU instruction that uses R5 as either SR1 and SR2.
-

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].**

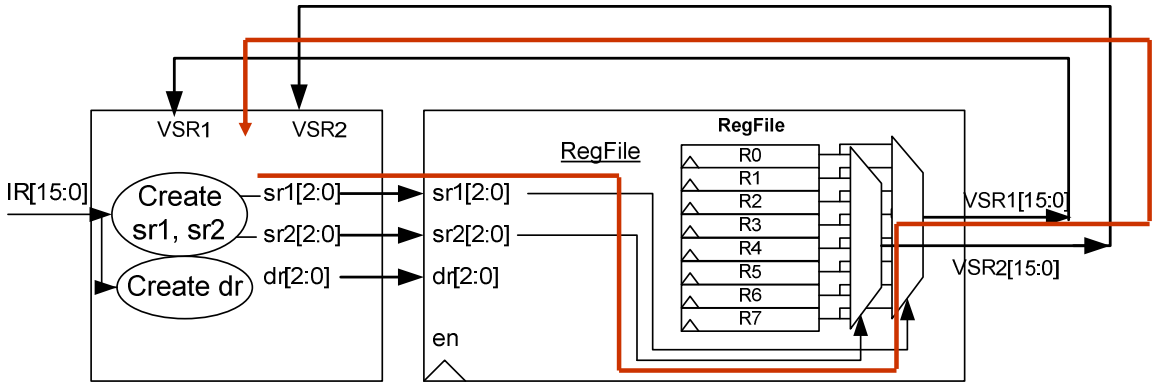
- `W_Control_out`[2 bits], `Mem_Control_out`[1 bits]: They equal the values of the `W_Control_in` and `Mem_Control_in` .
- `aluout`: [16 bits]This is the result of ALU operations which take the form
  - ← `SR1 (OPERATION) SR2/sxt(imm5)` for AND and ADD or
  - ← `(NOT) SR1` (NOT operation)

  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. For branch, JMP and load instructions, `sr2` = 0. **Asynchronously created.**
- `IR_Exec`: [16 bits] This reflects the contents of the input Instruction Register at the at the output and is synchronously created.
- `NZP`: [3 bits] This is synchronously created and reflects the contents of the NZP flags for control instructions only. For non-control instructions at the input IR it should reflect 3'b000. The aim is to use it to determine whether the branch conditions is satisfied. When `enable_execute` is low, the `NZP` goes to 000 synchronously.
- `M_Data`: [16 bits] should reflect the contents of `RegFile[SR]` synchronously (bypasses need to be considered).

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.** When `enable_execute` is low, the `NZP` goes to 000 synchronously.

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`).

Timing and Reset Behavior:

On reset = 1, the synchronous signals `dr, aluout, W_Control_out` and `Mem_Control_out` go to 0 in a synchronous reset. `sr1` and `sr2` are combinationally created.

Understanding Dependencies:

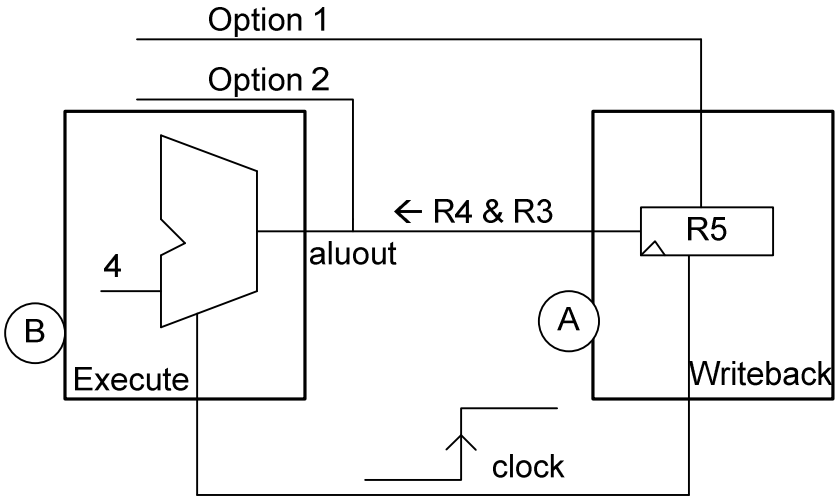
Let us consider two instructions

A: `@PC = 16'h3005, Instruction = AND R5, R4, R3 (16'h5B03)`

B: `@PC = 16'h3006, Instruction = ADD R6, R5, #4 (16'h1D64)`

A dependency exists here because A produces a value [DR = R5] which is needed by B [SR1 = R5] . This is also possible for, say, A being a load to R5.

As these two instructions flow through the pipeline flow through the pipeline (5 stages), at one clock edge, A would be in the writeback stage and waiting to write to the Register File (R5←) while B is ready to consume the value needed to do (← R5 + 4) at the Execute stage. At this point when B is executed it does not have the right value of R5. This is shown in the figure below for this specific example:



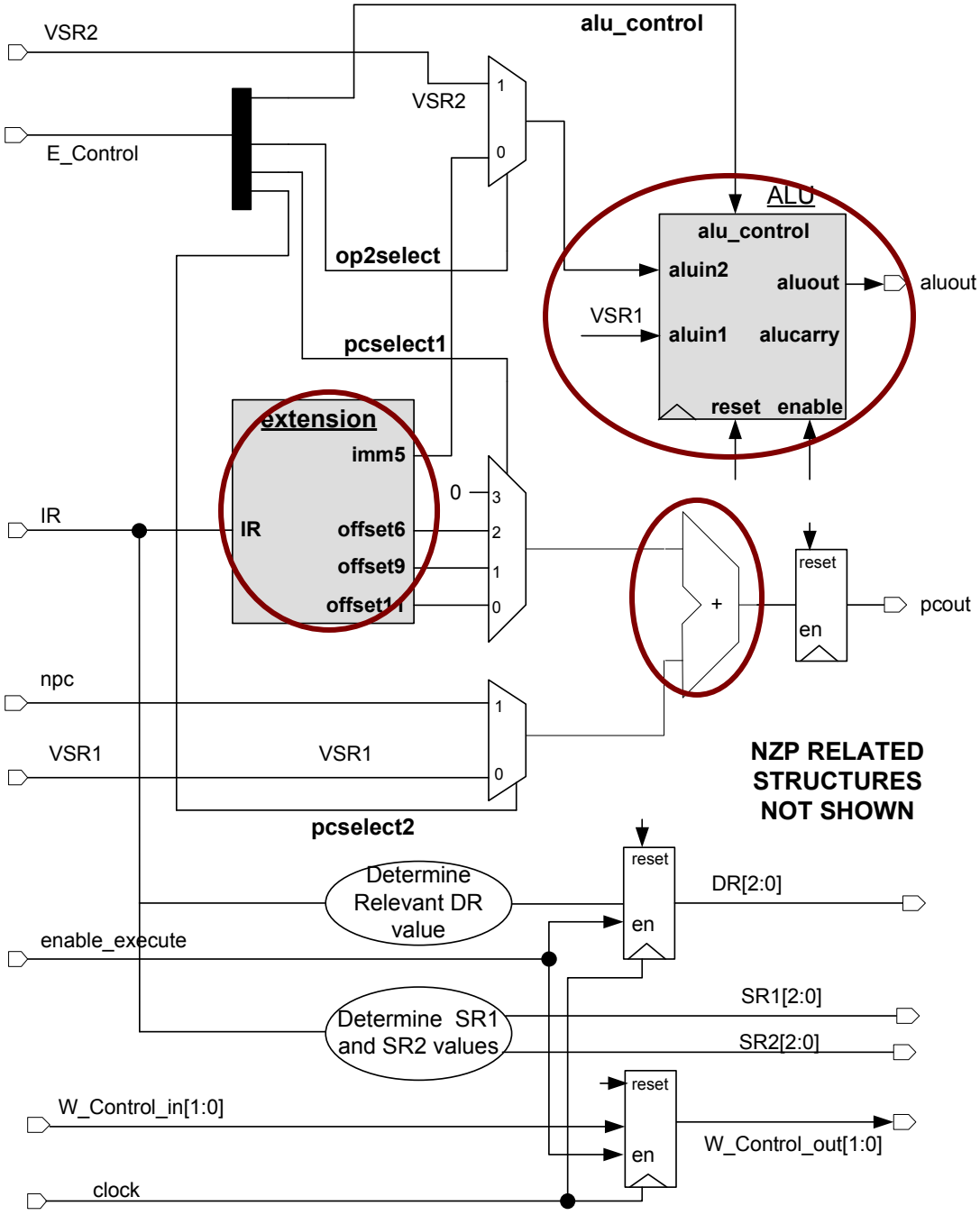
The above state of the pipeline is when A has been executed and is waiting to be written to R5 while B is waiting to be executed. Under normal operation, if B did not have a dependency on A, option 1 (value from register file) would be used. But in this case, we would have to use option B (which is the bypass from aluout) for computations.

We can have the dependency exist for either SR1 or SR2 and for

(case a) the bypass coming from the ALU output or

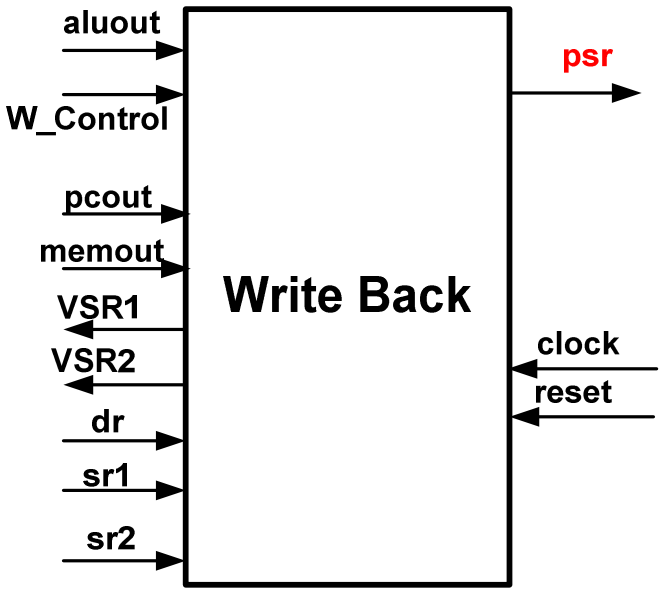
(case b) the bypass value coming from Memory (for load followed by dependent instruction).

Therefore, we have `bypass_alu_1` (SR1 and case a), `bypass_alu_2` (SR2 and case a), `bypass_mem_1` (SR1 and case b) and `bypass_mem_2` (SR2 and case b) to cater to all possibilities in case of the Execute Unit.


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

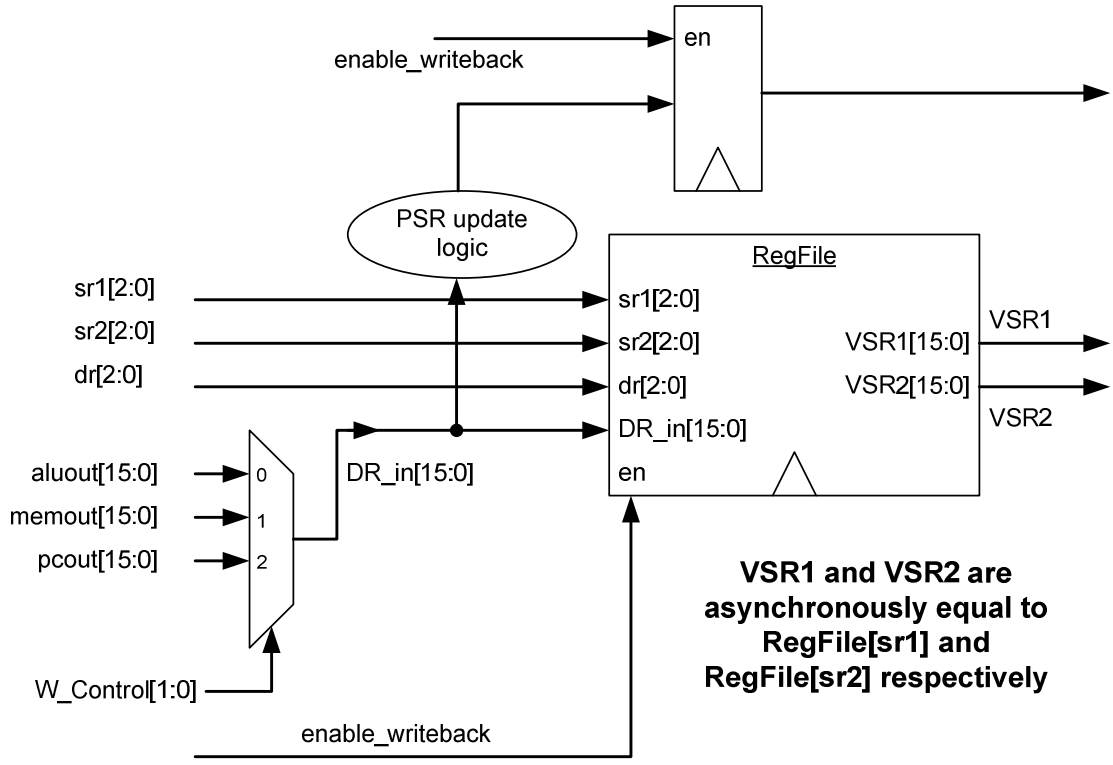
- `VSR1, VSR2` [16 bits] Value returned from the register file **(ASYNCHRONOUS)** corresponding to `RegFile[sr1] and RegFile[sr2]`

- `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

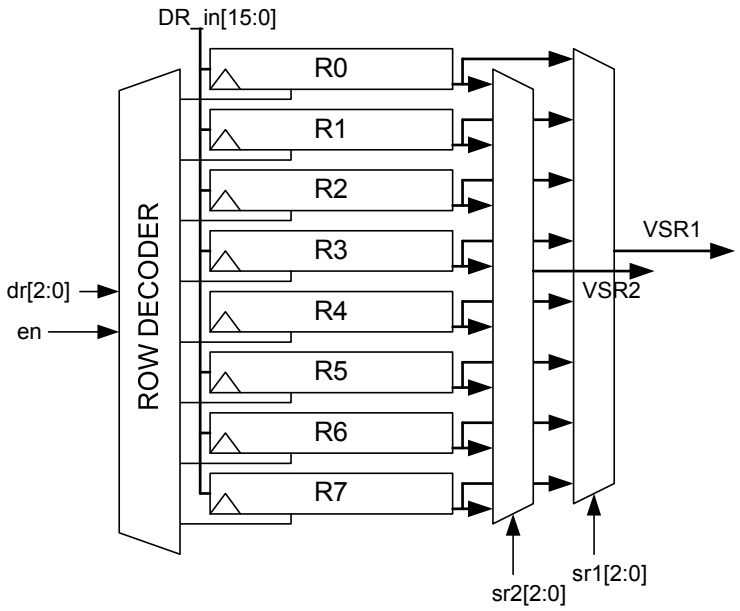
- if we are writing `16'hfff2` to the register file we would have `psr = 3'b100`,
- if we are writing `16'h00f2` to the register file we would have `psr = 3'b001`,
- if we are writing `16'h0000` to the register file we would have `psr = 3'b010`,

The internal details of this block are shown below:



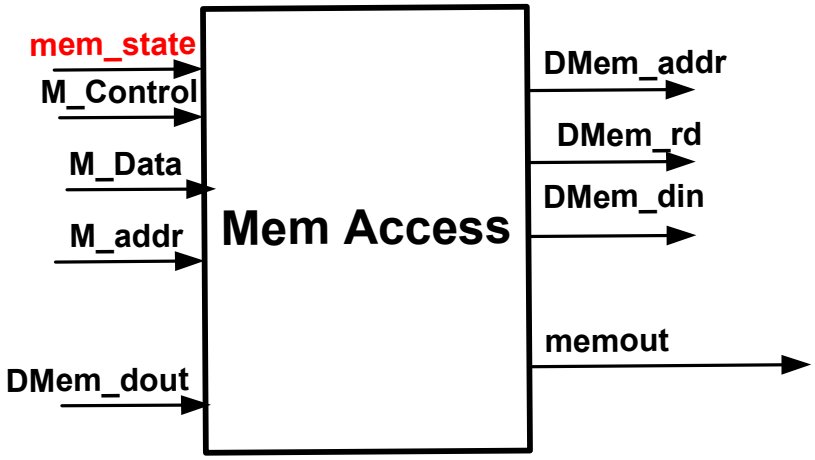
Note the use of the W_Control signal to determine the data that needs to be written into the register file. The write to the register file would take the form `RegisterFile[dr]` ← `DR_in`

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.

MEMACCESS BLOCK:



Inputs:

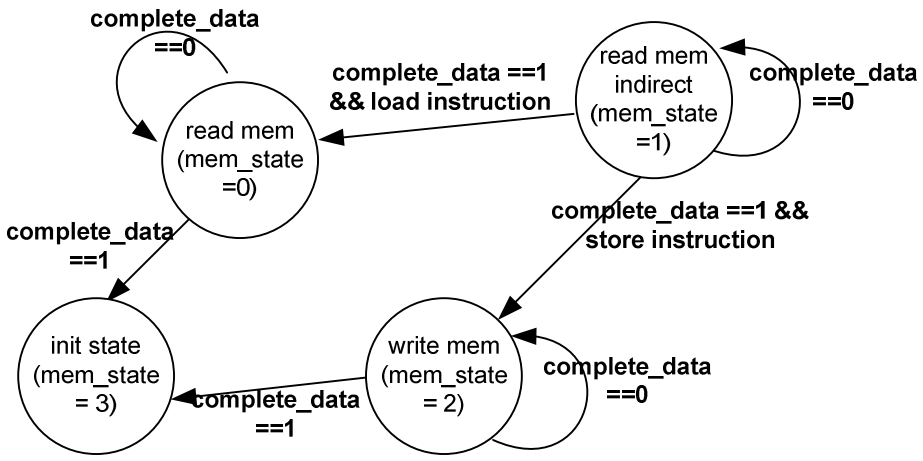
- `M_Data`: [16 bits] This signal comes from the Execute block and is the value that is going to be written to Data Memory for stores.
- `M_Addr`: [16 bits] This signal is connected to the pcout signal from the Execute block and is used as the address for reads and writes to memory when appropriate.
- `M_Control`: [1 bit] This signal is connected to the `Mem_Control_out` signal from the Execute is used to indicate a indirect addressing mode per the table in the decode block description.

- `mem_state`**:** [2 bits] This signal is sent in by the controller to control the operations within the Mem Access block. This will be described in greater detail shortly.
- `DMem_dout` [16 bits]. This is the value read from the data memory which can either be an address for indirect addressing or data for a normal memory load operation.

Outputs:

- `DMem_addr`: [16 bits] This is the address that is sent out to the Data Memory for reads from or writes to it. Would be z's (high impedance) for `mem_state` = 3.
- `DMem_din`: [16 bits] Reflects the values that need to be written to memory for stores. Would be 0 for loads. Would be z's (high impedance) for `mem_state` = 3.
- `DMem_rd`: [1 bit] This signal is 1 for reads from and 0 for writes to memory. Would be z's (high impedance) for `mem_state` = 3.
- memout [16 bits] Always reflects the contents of `DMem_dout` asynchronously and is valid when loads are done.

The state transitions for the `mem_state` value is as shown below when a Memory based operation is seen at the output of the Execute block. For the duration that these transitions are going on, the rest of the pipeline should be stalled waiting for the completion of the memory related operations.



When

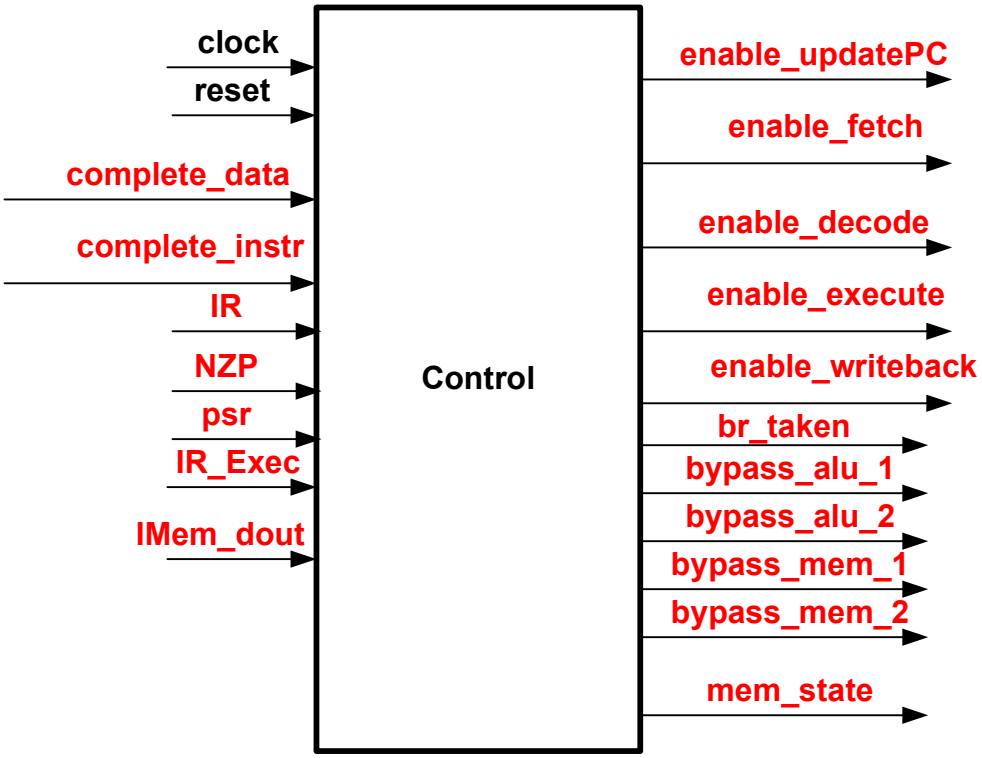
i) `mem_state` = 0 (reading memory for loads) `DMem_addr = M_addr` for `LDR`, `LD` and `DMem_dout` for `LDI` (prev value read in is used as address ; `DMem_din` = 0;

ii) `mem_state` = 2 (writing memory for stores) `DMem_addr = M_addr` for `STR`, `ST` and DMem_dout for STI (prev value read in is used as address); `DMem_din = M_data`;

iii) `mem_state` = 1 (reading from memory for indirect addressing) `DMem_addr` = `M_addr; Dmem_din` = 0;
`DMem_rd` = 0 for writes and 1 for reads.
For `mem_state = 3, DMem_rd = z ; DMem_dout = z; DMem_addr = z;`

Timing behavior:
This block does not have a clock. Instead, the block is controlled by the value of the mem_state value and its transitions. The mem_state values are controlled by the Controller.

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
- `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 which will be used to determine the `bypass_(mem/alu)_(1/2)` values.
- `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).

- `IR_Exec`: [16 bits] This signal comes from the Execute which is used to determine if a Memory operation is detected.
- `IMem_dout`: [16 bits] This the signal from the IMem which is used by the controller to determine the presence of control instructions.
- `NZP`[3 bits]: From the Execute block and is used with the psr register to create `br_taken` as described below.

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 `IMem_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 IMem_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.
- `bypass_alu_1, bypass_alu_2`: These signals are explained in the execute unit. They are created on the basis of the analysis of the sequence of IR values seen at the input of the controller.
- `bypass_mem_1, bypass_mem_1`: These signals are explained in the execute unit. They are created on the basis of the analysis of the sequence of IR values seen at the input of the controller.
- `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. This signal goes to the MemAccess block.
- `br_taken` [1 bit]. When 1 it implies that the Control instruction being serviced has a branch taken.

The `psr and NZP` 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`. The `br_taken` is created for the control instructions using the logic `br_taken = |(psr & NZP)`

In case of stores, there is a need for bypasses to be considered for `SR` for all store types and for `BaseR` for STR. For purposes of mapping these registers to `sr1` and `sr2` from the Execute to the Writeback:

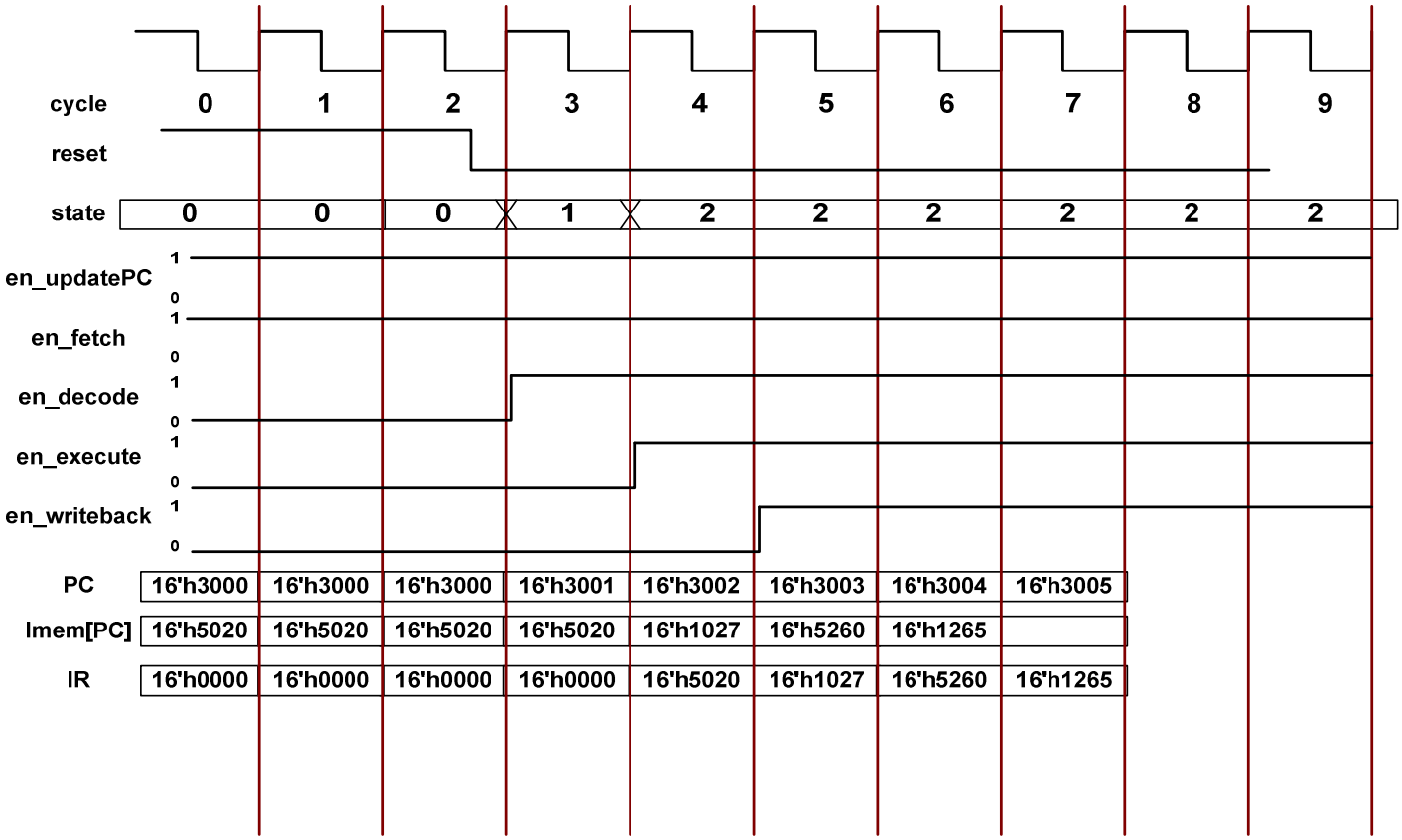
- `sr1 = IR[8:6] = BaseR` (valid for STR only) and
- `sr2 = IR[11:9] = SR` (valid for all stores)

Again, bypasses need to be considered for both `SR` (all stores) and `BaseR` (for STR only).

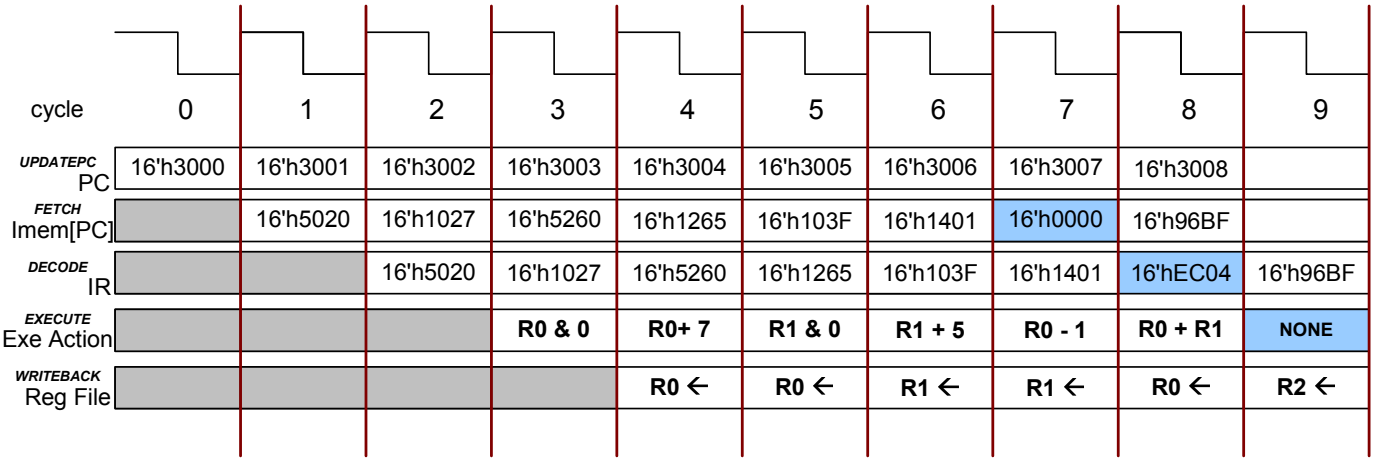
Timing and Reset behavior:

On reset all enables and bypass signals go to 0 but `mem_state` = 3. Reset is synchronous. The enables and bypass values are sensitive to the changing values of the IR from the decode and the Execute Unit, the completes, Instr_dout, prev_enables, psr, NZP.

Startup Timing:



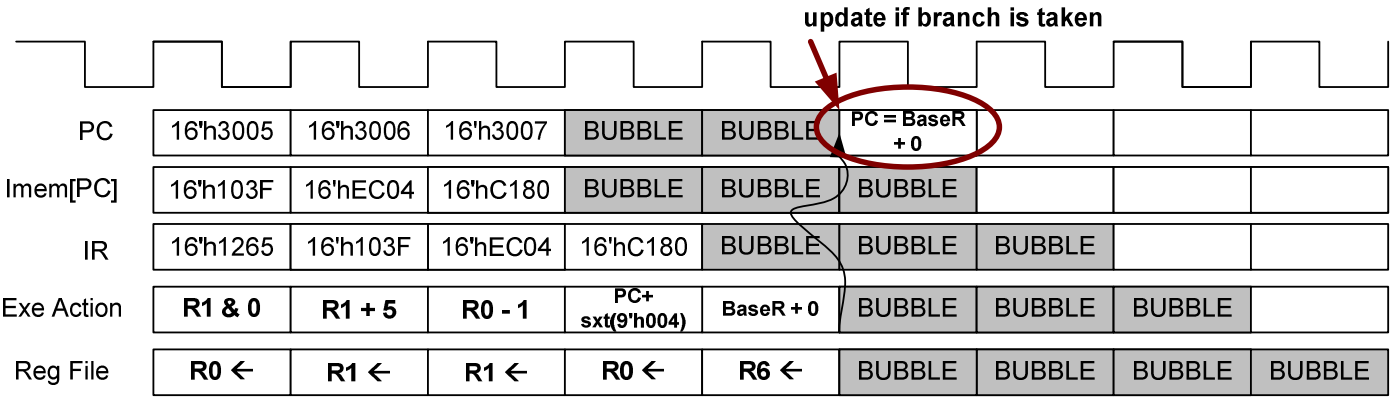
Timing for ALU Operations:



cycle	0	1	2	3	4	5	6	7	8	9
UPDATEPC PC	16'h3000	16'h3001	16'h3002	16'h3003	16'h3004	16'h3005	16'h3006	16'h3007	16'h3008	
FETCH Imem[PC]		16'h5020	16'h1027	16'h5260	16'h1265	16'h103F	16'h1401	16'h0000	16'h96BF	
DECODE IR			16'h5020	16'h1027	16'h5260	16'h1265	16'h103F	16'h1401	16'hEC04	16'h96BF
EXECUTE Exe Action				R0 & 0	R0+ 7	R1 & 0	R1 + 5	R0 - 1	R0 + R1	NONE
WRITEBACK Reg File					R0 ←	R0 ←	R1 ←	R1 ←	R0 ←	R2 ←

Timing for Control Operations

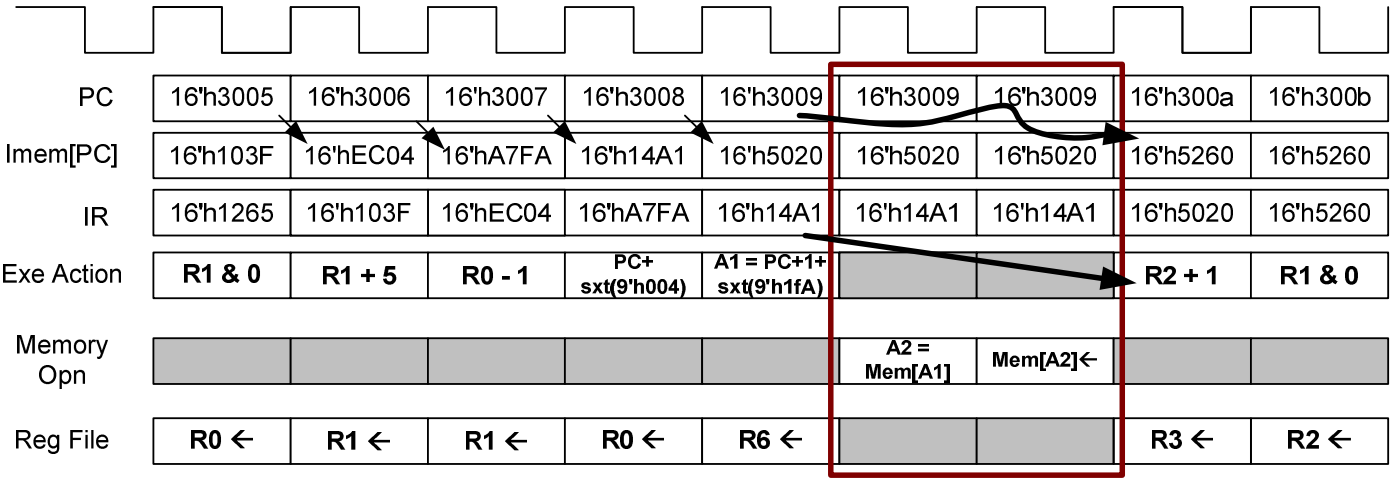
Consider for this case Instruction = `C180 // JMP R6`



As can be seen, when a control instruction is encountered, the pipeline does not service in any new instructions until the Execute creates a new PC value which will be used as long as the br_taken signal is high to indicate a taken branch or JMP signal. Also there should be no writeback for control instructions

Timing for Memory Operations

Consider for this case Instruction = `A7FA // LDI R3,-6 ; R3 = Mem[Mem[PC+1 -6]]`



In this case i.e. for LDI, we are going to have to stall the entire pipeline for 2 cycles that the Mem_Access block is busy getting data from the memory. We of course assume that there is no wait for complete in this case**. Similarly, we would need a 1 cycle stall for LDR, LD, STR, ST and 2 cycles stall for STI. It must also be noted that there should be no writeback for stores. Remember that you would stall for longer if the complete_data is not high i.e. data is not ready.**

It must be noted that all of the above effects of stalling and sending in bubbles would be achieved by the assertion of the enables to 0/1. This is explained in greater detail below.

Behavior affected by `complete_instr`: If the `complete_instr` signal is low, it implies that the value from the memory is not valid yet. Therefore for the general case, the `enable_ updatePC` and `enable_decode` stay low until the `complete_instr` goes high and the instruction is ready to use. The pipeline continues to function correctly for the instructions that came in before the instruction that is being waited on. Remember that the complete signal here is only supposed to affect the pipeline when a fetch is being performed.

Behavior for Control Operations: The detection of the Control (BR/JMP) Operations is done early by the analysis of the `Instr_dout` signal from the Instruction Memory. The control instruction is processed only after the validity is proved by the presence of the `complete_instr` going high. Beyond this point, the control instruction is sent through the pipeline while nothing is fetched / decoded / executed until the result of the execute unit for the control provides the requisite NZP and PC_{next} value to make a decision on whether the branch is taken or not. If the branch is taken, then the new PC value that is created at the output of the execute block is used to update the PC and is used for the fetch in the clock cycle after the PC is updated.

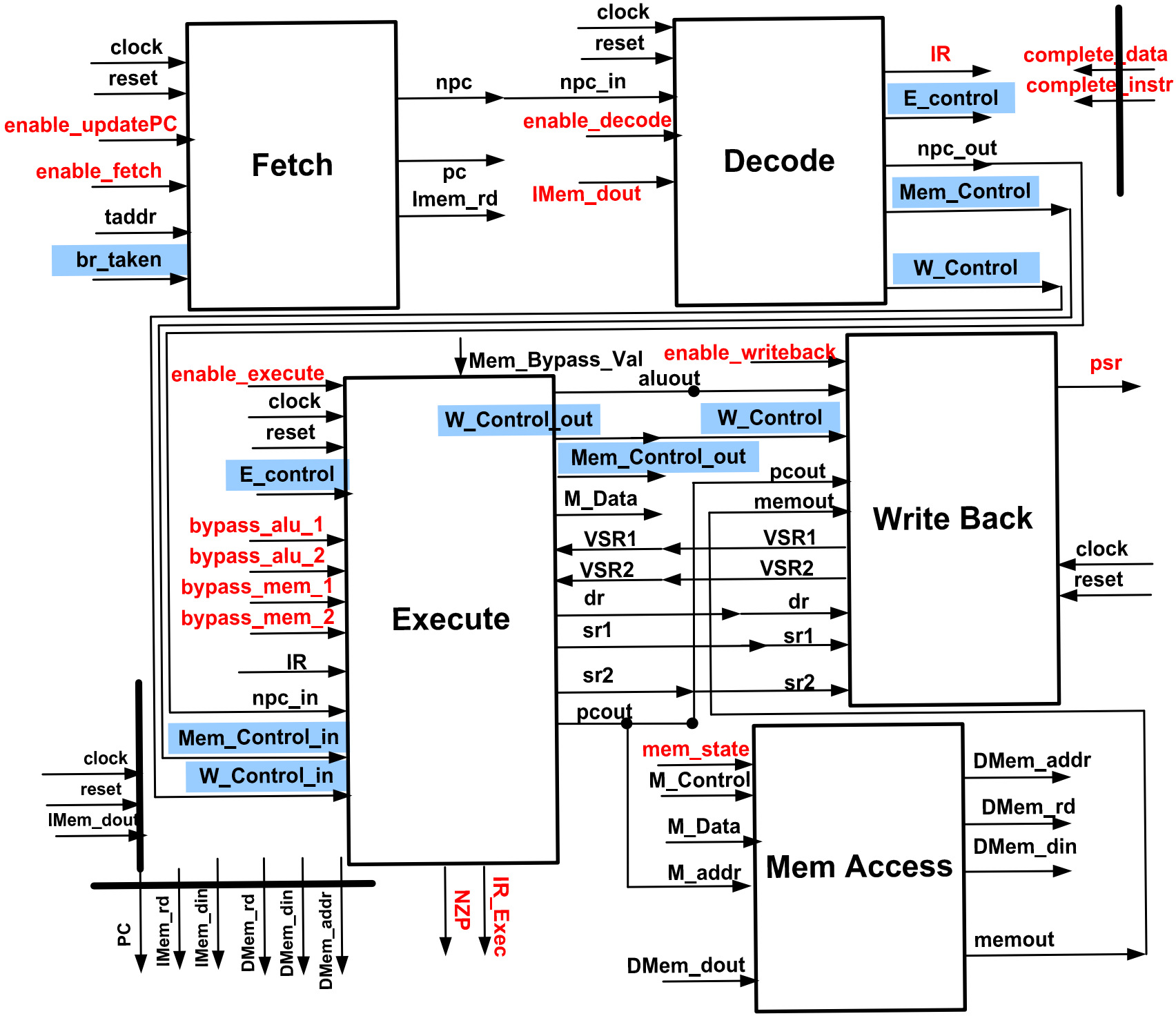
Detailed `br_taken` operation: `br_taken` uses the values `NZP` (from Execute) and `psr` (from Writeback) after the relevant values are created at the output of the Execute for control instructions. The br_taken will have the value `= |(NZP & psr)`. For all other times it should be 0 to ensure `pc` is updated to `pc+1`. Thus, for the case when there should be a branch: a) BR (when the branch condition is satisfied) or b) JMP (unconditional branch), br_taken = 1 after 3 clock cycles for the control instructions to reach the Execute output. This makes, the PC change in the next clock cycle to the `taddr` value.

Behavior for Memory Operations(except LEA): The Memory operations are detected at the output of the Execute block by analyzing the `IR_Exec` signal. When a Memory operation, except LEA which is treated as an ALU type of instruction, is seen, the entire pipeline is stalled by making all the enables move down to zero until the memory read or write is complete. Again, the validity of the completion is determined by a)The `complete_data` going high and `memout` having the value of read operation for Loads and b) the `complete_data` going high for stores. The important thing to note for stores is that the writeback stage must be disabled. Thus, a store would result in a bubble for the writeback.

Other operational instructions:

- As you will note from the timing the control instructions, the presence of a BR/JMP at `IMem_dout` will cause the `enable_fetch` & `enable_updatePC` to go low. In the next clock cycle, the `enable_decode` would go low (after the BR/JMP has gone through it) and then `enable_execute` will go low in the cycle after that and so on until the bubble passes through the pipeline. Note that the bubbles in the figure are achieved by de-asserting the relevant enables.
- `pc` gets updated to `pc+1` as the BR/JMP passes through the pipeline leaving a bunch of bubbles in its wake. But nothing is fetched. When the BR/JMP address is resolved and we know whether the branch is taken or not, we either keep the PC+1 value already present by de-asserting `enable_updatePC` or we assert `enable_updatePC` and take the `taddr` value for the `pc`.
- **For all instructions, the next instruction SHOULD NOT be sent in until `Imem_rd` is high. Only when a `Imem_rd = 1` is seen should the instruction memory (i.e. you as the testbench) send in a new instruction at the next positive edge of the clock cycle.**

The final top level view of this system sans the controller takes the form shown below. In this figure, the signals to and from the controller are in red.


Fetch
clock
reset
enable_updatePC
enable_fetch
taddr
br_taken
npc
pc
Imem_rd
Decode
clock
reset
npc_in
enable_decode
IMem_dout
IR
E_control
npc_out
Mem_Control
W_Control
complete_data
complete_instr
Execute
enable_execute
clock
reset
E_control
bypass_alu_1
bypass_alu_2
bypass_mem_1
bypass_mem_2
IR
npc_in
Mem_Control_in
W_Control_in
Mem_Bypass_Val
W_Control_out
Mem_Control_out
M_Data
VSR1
VSR2
dr
sr1
sr2
pcout
aluout
enable_writeback
W_Control
pcout
memout
VSR1
VSR2
dr
sr1
sr2
Write Back
psr
clock
reset
Mem Access
mem_state
M_Control
M_Data
M_addr
DMem_dout
DMem_addr
DMem_rd
DMem_din
memout
clock
reset
IMem_dout
PC
IMem_rd
IMem_din
DMem_rd
DMem_din
DMem_addr
NZP
IR_Exec