Building a Basic RV32I Pipeline
Report 4- Integration & Hazard Handling

Group 2
E/16/203 - Lakshan S.A.I
E/16/332 - Samaraweera A.A.R.V

1. Instruction Fetch Stage

Modules in this Stage

i). Instruction fetch module
ii). Instruction Cache module
iii). Instruction Cache Controller module
iv). Instruction Memory module

1.1 Instruction fetch module

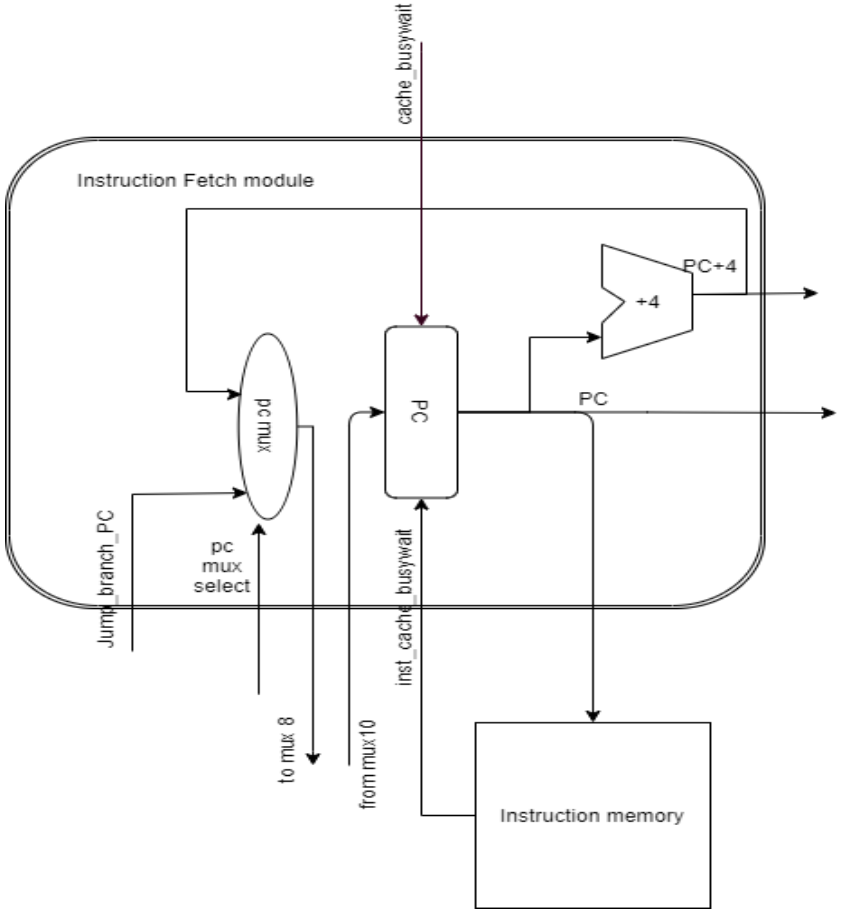


Figure 1: Instruction Fetch module

This module is a combination of

PC module,
+4 adder,
PC mux module in the datapath.

I/O signals

Inputs:

CLK : 1bit input - Clock signal
RESET : 1 bit input - reset signal - This is used to Reset the PC
Instruction_mem_busywait : 1 bit input - busywait signal from instruction memory
Data_mem_busywait : 1 bit input - busywait signal from the cache memory
Jump_branch_signal : 1 bit input - as the select bit as PC mux select
Jump_Branch_PC : 32 bit input - modified PC for branch or jump instructions
From_mux10 : 32 bit input from mux10

Outputs:

PC : 32 bit output - PC value for instruction cache :
INCREMENTED_PC_by_four : 32 bit output - PC + 4 value
To_mux8 : 32 bit output for mux 8

Implementation & Timing

PC is held whenever data memory or instruction memory suffered a miss, therefore or operation is done between those Instruction_mem_busywait and data_memory_busywait.

When RESET, the PC value is set to -4 so that the CPU will restart from the next positive clock edge as PC = 0.

Every positive clock edge, PC is updated to PC+4 value from the adder when busywait is de-asserted. For the adder, we introduced #2 time unit delay. This is the same delay of the addition operation of the main ALU.

jump_branch_signal is used to select between modified PC from 3rd stage and typical PC+4 value

PC register write is incurred #2 delay.

1.2 Instruction cache module

The instruction cache module has a cache control module within it.

I/O signals

Inputs:

Clock : clock signal
Reset : reset signal
PC : PC value as address

mem_Readdata : 127 bits of Read data from instruction memory
mem_BusyWait : busywait signal from instruction memory

Outputs:

readInstruction : 32 bit instruction output from the cache
Busywait : busywait signal of instruction cache

mem_Read : read signal for instruction memory
mem_Address : address of the required block to the cache

Hit : instruction hit signal this will be used by regfile and data memory

Implementation & Timing

Our Instruction cache can store 4 instructions per block. After 4 instructions cache holds the CPU and goes to the instruction memory and fetch the next four instructions from the instruction memory. At the time it may seem that instruction cache size is not enough. We will be increasing it in the next part.

Cache has total of 8 blocks each block is 128 bits and byte addressed. since cache fetches 4 bytes per time least significant 2 bits of the PC are ignored. The offset is 2 bits wide so that we can differentiate between 4 instructions total. The index is 3 bits wide so that a total of 8 blocks can be stored within the cache. Therefore TAG is 25 bits wide.

Whenever a new PC value is received the PC value is decoded according to index, offset, and TAG. Here we insert a check to disregard -4 value at the very beginning when the CPU resets.
This decode costs #1 time unit.

Then the TAG matching and hit state resolving occur asynchronously. This costs #5 unit times.
Overall #6 units delay for instruction cache.

While hit status resolving data extraction from the relevant block happens with #1 time unit delay. Hit status resolving and data extraction happen simultaneously.

Cache memory will flag a busywait as soon as a new PC comes, to signal the CPU that the cache is busy fetching the instruction.

If the current PC value is a hit, this Busywait signal will be deasserted in the following clock edge. The instruction is fetched before this clock edge.
If it's not a hit cache will not deassert the busywait at this clock edge but a garbage instruction may already be fetched since it is asynchronous cache reading. This garbage instruction may be a register write or data memory write instruction. So this has a vulnerability that these garbage instructions can corrupt data in register file or data memory. This is where the "*instrHit"* signal is needed. When there's a miss the hit signal is deasserted. This is used as a signal in the register file and Data memory before doing any write operation to make sure the fetched instruction is not garbage.

When there's a miss, the cache controller module does the part to generate relevant control signals to get the required data from the instruction memory.

After the required data is fetched from the memory that 127-bit block will be stored in the cache with TAG and setting VALID bit to 1. This register writing operation takes #2 time delay.

1.2.1 Instruction cache controller module

This module is responsible for generating relevant control signals for instruction memory when there is a miss occurred in the Instruction cache.

I/O signals

Inputs:

Clock : clock signal
Reset : reset signal
Tag,Index : These two signals used to set the read address to instruction memory
mem_Readdata : 127 bits of Read data from instruction memory
mem_BusyWait : busywait signal from instruction memory

Outputs:

Busywait : this is for cache controller to say that it is busy fetching data from instruction memory. This is for cache memory.

mem_Read : read signal for instruction memory
mem_Address : address of the required block to the cache

Implementation & Timing

We implemented cache controller as a 2 state FSM.
Two states are,

i). IDLE
ii).MEM_READ

This FSM can be easily understandable by following FSM diagram

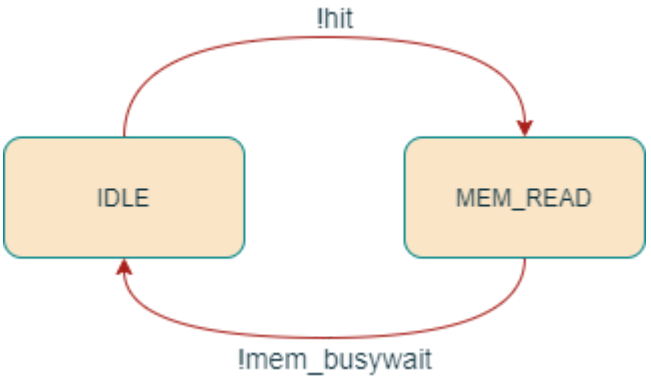


Figure 2: FSM diagram

State changes are happening at the positive clock edge while next-state logic is combinational (asynchronous).

Instruction memory control signal generation is happening according to the state of the FSM.

At RESET the FSM goes to default IDLE state.

State changing time delay is negligible

1.3 Instruction memory module

I/O signals

Inputs:

Clock : clock signal
Reset : reset signal
Address : block address (combination of required TAG and Index)

Outputs:

busywait : this is for the cache controller to say that instruction memory is busy fetching data. The cache controller waits until this signal to de-asserted to change state from MEM_READ to IDLE.

readdata : 127 bit block read data

Implementation & Timing

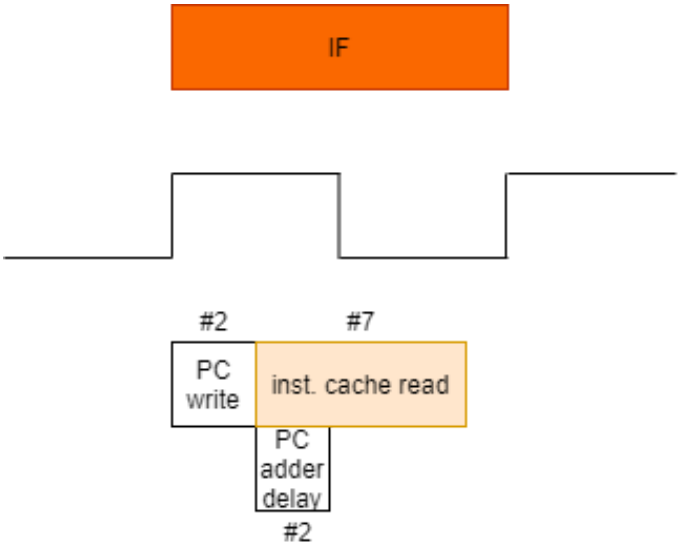
The instruction memory is byte addressed. Total of 1024 registers of 8bit wide. Since Instruction memory and cache communicate in blocks each read request from cache involves a 127-bit block.

The block address is 27 bit wide.

Instruction memory fetches data byte by byte until the whole block is fetched. Each byte fetch is costs #40-time units. Meaning for the whole block to be fetched there will be a total of 40 x 16 time units delay.

Encoded instructions should be hardcoded into the module

Instruction Fetch stage timing diagram



2. Regfile Access stage and control signal generating stage

Modules in this Stage

i). Control Unit module

ii). Reg FIle module

iii). Extension wire module

iv). Dedicated adder

2.1 Control unit module

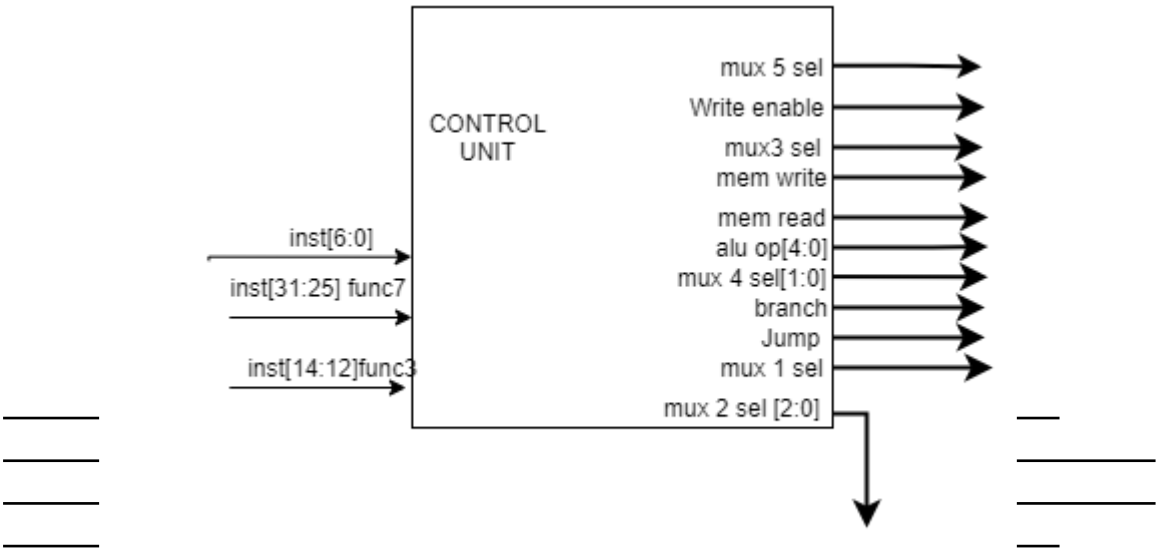


Figure 3: Control Unit

I/O signals

Inputs:

Inst[6:0] : The opcode field in the instruction

Inst [31:25] : the funct7 field - here since we only consider base instruction set and standard M extension, we only needed two bits from funct7 field. Though we inserted the whole funct7 field as an input.

Inst [14:12] : the funct3 field - this field is needed to generate instruction-specific control signals for instructions in the same type.

Ex: to generate control signals into addi and slti.Both instructions have same Opcode. The difference between these two instructions is in the funct3 field.

Note: here in the figure we showed which parts of the decoded instruction is used in the control unit to generate relevant control signals. But in the implementation, we included instruction decoding within the control unit module. Therefore you may notice we have given the whole 32 bit instruction into the control unit.

Outputs:

Mux 5 sel: 1 bit select signal for mux 5 - multiplexer mux_5 is used to select between reg file output (Out2) and immediate value (mux 2 out) which direct into the ALU module.

Write enable: 1 bit - this signal is checked whenever a write occurs to refile at the last stage of the pipeline. When this signal is set to one, the write occurs in the reg file.

Mux_3_sel : 1 bit - this is the select signal for multiplexer mux_3 which selects between ALU output and data memory output. After this, relevant data is direct to regfile.

Mem_write and mem_read: 1 bit signals - these signals are checked when accessing data memory to distinguish whether it's a write or read.

Alu_op : 5 bit signal for ALU - this signal tells the ALU which operation it should in the two data inputs to it. Whether it's addition subtraction etc.

Mux_4_sel: 2 bit signal - this signal is multiplexer mux 4 which have 3 inputs namely ALU output, current pc + 4 value, and immediate value from the mux 2 (mux 2 used to select the appropriate immediate from the set of immediate)

Branch and jump: 1 bit signals -two separate signals to distinguish whether the instruction is a jump or a branch. These signals are inputs to the jump and branch unit. (please see the data path).

Mux_1_sel: 1 bit signal - a select signal to multiplexer mux 1, which selects between reg file output and PC value before giving them to the ALU for its operations

Mux_2_sel: 3 bit signal: mux 2 is the multiplexer that selects between immediates according to the instruction. There are 5 such immediates so we needed 3 bits to distinguish between them.

(we numbered multiplexers for easy identification. Please have a look at our datapath)

(all the control unit control signals are indicated in red in the data path.)

Implementation & Timing

First, we decoded instruction and extracted opcode, funct3, and funct7 field 0th and 7th bits.
Note*: it is observed that with the instructions we are considering, we only need the 0th and the 7th bit from funct7.*
For instruction decoding, we removed the delay we added before because it is just asynchronous wire operation so time delay would be very low.

Then according to the opcodes, we generated common control signals to each instruction type.
We have introduced #1 time unit for generating common control signals to represent the combination logic delays here.
To identify each type we generate an internal 4 bit control signal named instr_type.

Then we create an instruction-specific opcode concatenating funct7 two bits, funct3, and instr_type.
Using this instruction-specific opcode, we generated the instruction-specific control signals.
For this, we introduced #1 time unit delay to represent combinational logic delays here.

Design Decisions

First, Instruction is decoded into opcode, funct3, and funct7 2 bits
Namely funct7_A and funct7_B.

Generating basic control signals:
Mux select signals, memRead, memWrite, WriteEnable signals,Branch or Jump are identified as basic control signals
These signals are unique to a particular OPCODE. Therefore these control signals are generated just by comparing opcode

Generating instruction specific control signals:
To generate instruction specific ALUop, first control unit resolve which type of instruction it is (to identify type we used instr_type signal within the control unit), and then depending on the funct3 and funct7 2 bits it generates 5-bit ALUop.

Load and store instructions:
The control unit generates the same control signals for all Load instructions and the same control signals for all Store instructions. Whether the instruction is a full word, half word, byte is considered in the memory access stage using funct3 field.

The following table consists of all the control unit signals for each specific instruction.

Some abbreviations used in the following table:

WE: write enable signal
MR : memread signal
MW: memwrite signal

control unit signal values

Instruction	aluOP 5bit	mux 1	mux 2	mux 3	mux4	mux 5	WE	MR	MW	Branch	Jump
LUI	**x**	**x**	**011**	**0**	**00**	**x**	**1**	**0**	**0**	**0**	**0**
AUIPC	**ADD**	**1**	**011**	**0**	**01**	**0**	**1**	**0**	**0**	**0**	**0**
JAL	**ADD**	**1**	**100**	**0**	**10**	**0**	**1**	**0**	**0**	**0**	**1**
JALR	**ADD**	**0**	**100**	**0**	**10**	**0**	**1**	**0**	**0**	**0**	**1**
BEQ	**SUB**	**0**	**000**	**x**	**xx**	**1**	**0**	**0**	**0**	**1**	**0**
BNE	**SUB**	**0**	**000**	**x**	**xx**	**1**	**0**	**0**	**0**	**1**	**0**
BLT	**SUB**	**0**	**000**	**x**	**xx**	**1**	**0**	**0**	**0**	**1**	**0**
BGE	**SUB**	**0**	**000**	**x**	**xx**	**1**	**0**	**0**	**0**	**1**	**0**
BLTU	**SUB**	**0**	**000**	**x**	**xx**	**1**	**0**	**0**	**0**	**1**	**0**
BGEU	**SUB**	**0**	**000**	**x**	**xx**	**1**	**0**	**0**	**0**	**1**	**0**
LB	**ADD**	**0**	**001**	**1**	**01**	**0**	**1**	**1**	**0**	**0**	**0**
LH	**ADD**	**0**	**001**	**1**	**01**	**0**	**1**	**1**	**0**	**0**	**0**
LW	**ADD**	**0**	**001**	**1**	**01**	**0**	**1**	**1**	**0**	**0**	**0**
LBU	**ADD**	**0**	**001**	**1**	**01**	**0**	**1**	**1**	**0**	**0**	**0**
LHU	**ADD**	**0**	**001**	**1**	**01**	**0**	**1**	**1**	**0**	**0**	**0**
SB	**ADD**	**0**	**001**	**1**	**01**	**0**	**0**	**0**	**1**	**0**	**0**
SH	**ADD**	**0**	**001**	**1**	**01**	**0**	**0**	**0**	**1**	**0**	**0**
SW	**ADD**	**0**	**001**	**1**	**01**	**0**	**0**	**0**	**1**	**0**	**0**
ADDI	**ADD**	**0**	**010**	**0**	**01**	**0**	**1**	**0**	**0**	**0**	**0**
SLTI	**SLT**	**0**	**010**	**0**	**01**	**0**	**1**	**0**	**0**	**0**	**0**

SLTIU	SLTU	0	010	0	01	0	1	0	0	0	0
XORI	XOR	0	010	0	01	0	1	0	0	0	0
ORI	OR	0	010	0	01	0	1	0	0	0	0
ANDI	AND	0	010	0	01	0	1	0	0	0	0
SLLI	SLL	0	010	0	01	0	1	0	0	0	0
SRLI	SRL	0	010	0	01	0	1	0	0	0	0
SRAI	SRA	0	010	0	01	0	1	0	0	0	0
ADD	ADD	0	xxx	0	01	1	1	0	0	0	0
SUB	SUB	0	xxx	0	01	1	1	0	0	0	0
SLL	SLL	0	xxx	0	01	1	1	0	0	0	0
SLT	SLT	0	xxx	0	01	1	1	0	0	0	0
SLTU	SLTU	0	xxx	0	01	1	1	0	0	0	0
XOR	XOR	0	xxx	0	01	1	1	0	0	0	0
SRL	SRL	0	xxx	0	01	1	1	0	0	0	0
SRA	SRA	0	xxx	0	01	1	1	0	0	0	0
OR	OR	0	xxx	0	01	1	1	0	0	0	0
AND	AND	0	xxx	0	01	1	1	0	0	0	0
MUL	MUL	0	xxx	0	01	1	1	0	0	0	0
MULH	MULH	0	xxx	0	01	1	1	0	0	0	0
MULHSU	MULHSU	0	xxx	0	01	1	1	0	0	0	0
MULHU	MULHU	0	xxx	0	01	1	1	0	0	0	0
DIV	DIV	0	xxx	0	01	1	1	0	0	0	0
DIVU	DIVU	0	xxx	0	01	1	1	0	0	0	0
REM	REM	0	xxx	0	01	1	1	0	0	0	0
REMU	REMU	0	xxx	0	01	1	1	0	0	0	0
nop	xxxxx	x	xxx	x	xx	x	0	0	0	0	0

Note: Above the table is colored (grouped) according to opcodes for easy identification.

The First 4 instructions are different types but they have unique opcodes so we keep it white.

The nop instructions are there to insert bubbles to the pipeline when control and data hazards are there. This is inserted from the assembler. Our nop instruction consists of 32 bit 0s. For this instruction, no data memory write, read or reg file read write, jump or branch should not happen. Therefore these control signals are 0ed and other mux values are “don’t care”.

Other colour codes are

B type : `1100011` **Load :** `0000011` **Store :** `0100011`

I type : `0010011` **R type and M extention :** `0110011`

All the basic control signals are the same for a particular opcode.

Following encodings are used in control unit,

ALU_op :

- **ADD : 00000**
- **SUB: 00001**
- **AND : 00010**
- **OR : 00011**
- **XOR : 00100**
- **SLL : 00101**
- **SRL : 00110**
- **SRA : 00111**
- **MUL : 01000**
- **MULH : 01001**
- **MULHU : 01010**
- **MULHSU : 01011**
- **DIV : 01100**
- **DIVU : 01101**
- **REM : 01110**
- **REMU : 01111**
- **SLT : 10000**
- **SLTU : 10001**

2.2 REG FILE module

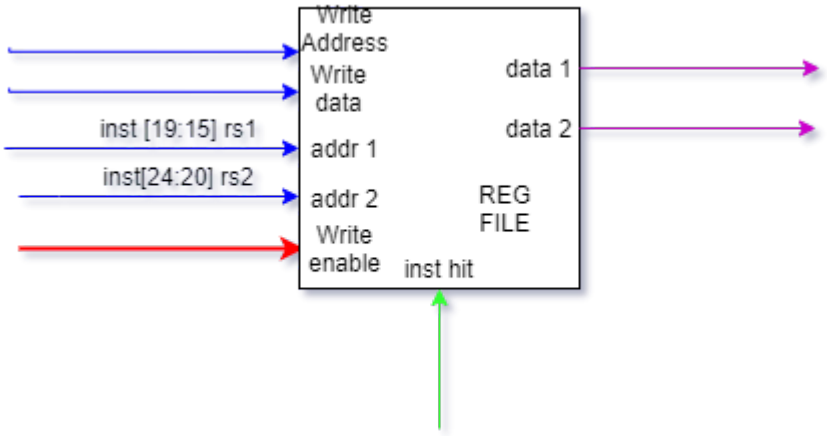


Figure 4: register file

I/O Signals

Inputs:

Write Address : 5 bit address which the register file write happen
Write data : 32 bit data that should be written in to the reg file
Addr1 and addr2 : 5 bit register read addresses
Writeenable : enable signal for write operation
Inst hit : 1 bit signal to identify whether the fetched instruction is garbage or not. This is Described below

Outputs:

data1 and data2: 32 bit register values corresponding to addr1 and addr2 addresses respectively.

Implementation & Timing

Regfile is a synchronized device according to the clock given. This is the dominating module taking the largest delay in the second stage. This is because register writing and register reading is time-consuming tasks. For Register reading, we introduced #1 time delay and for register writing #2 time delay. This is because compared to register reading register writing take a larger time. Typically register writing happens at the beginning of the clock cycle(first half of the clock), while register reading happens at the second half of the clock cycle. This way most recent updated data would be fetched when reading registers. For our implementation, we modeled the register file read operation as an asynchronous operation with a delay.

Note: clock signal and reset signals of the modules are not included in the data path. Having them too would make the diagram untidy and hard to read.

Design decisions

Asynchronous register read and synchronous register write.

*Why we used an **instrhit** signal from instruction cache memory?*
When a miss occurred in the instruction cache, it goes to the instruction memory and fetches the next set of instructions to the cache. To do this, it takes a delay of several clock cycles. Within this time there may be a garbage instruction fetched from the instruction memory and in the next stage, it is served as a legit instruction. If this garbage instruction contains a register write or a data memory write, it could corrupt the data in the register file and/or data memory. To avoid this we introduced a **instrhit** signal which is set to one when there's a hit within the cache. In other words, it means whether the fetched instruction in this clock cycle is a valid one. When there's a miss, then this signal is set to zero so that the regfile and data memory write won't occur(we check this as a condition within data memory and regfile memory before write operation). After a miss, When the cache update with the correct instructions the **instrhit** signal will set to 1 again. Actually **instrhit** signal is the same hit signal in the cache memory which resolves whether access is a hit or not, we just give it as an output.

Why didn't we use the typical busywait signal for this purpose?
Busywait means stalling the whole CPU.
which means while the instruction cache getting the new instruction from instruction memory,
The rest of the instructions which fetched BEFORE the instruction-cache miss, have to wait right where they are until the instruction cache gets updated. This leads to heavy inefficiency in the CPU.
For example, if miss occurs at the 5th instruction, 4th instruction is in stage two. Let's say Fetching 5th instruction from memory takes 3 cycles. So if we stall the cpu, the 4th instruction is still stuck in stage two even after 3 cycles while it should have been in 2+3 = 5th stage by then.
If it is implemented in our way, all the instructions fetched BEFORE the instruction cache miss, can go on without any disturbance while the instruction cache fetches from the instruction memory.
This would be very good in situations where data memory miss and instruction cache miss occur consecutively. Most of the clock cycles of accessing data memory and instruction memory would be overlapped saving a lot of time.

Configuring regfile write to the negative edge of the writeback clock cycle

In the write back stage if we start writing to the register file at the positive clockedge at the end of writeback stage. The writing operation should finish before the pipeline register updates. If the pipeline register updates before the writing operation finish in the register file, there may be garbage values get written to the register file. This possibility can be avoided if we synchronize the writing operation to the negative clock edge of the write back stage. So that writing operation takes place within the write back clock cycle. Since only the time-consuming module in the write back stage is the pipeline register it takes only #3 time of delay which is less than halfcycle #5. Therefore rather than wasting the rest of the half cycle doing nothing if we could use it for the write operation, we can avoid the possibility of data corruption.

Note:- We did not configure the register file to the negative edge. State this here as a suggestion.

2.3 Wire Extension Module

This is the module where we generate relevant extensions for input immediate values according to requirements.

I/O Signals

Inputs:

Instruction : 32 bit instruction as input

Outputs:

B_imm : 32 bit generated immediate value for B type
J_imm : 32 bit generated immediate value for J type
S_imm : 32 bit generated immediate value for S type
U_imm : 32 bit generated immediate value for U type
I_imm : 32 bit generated immediate value for R type

Implementation & Timing

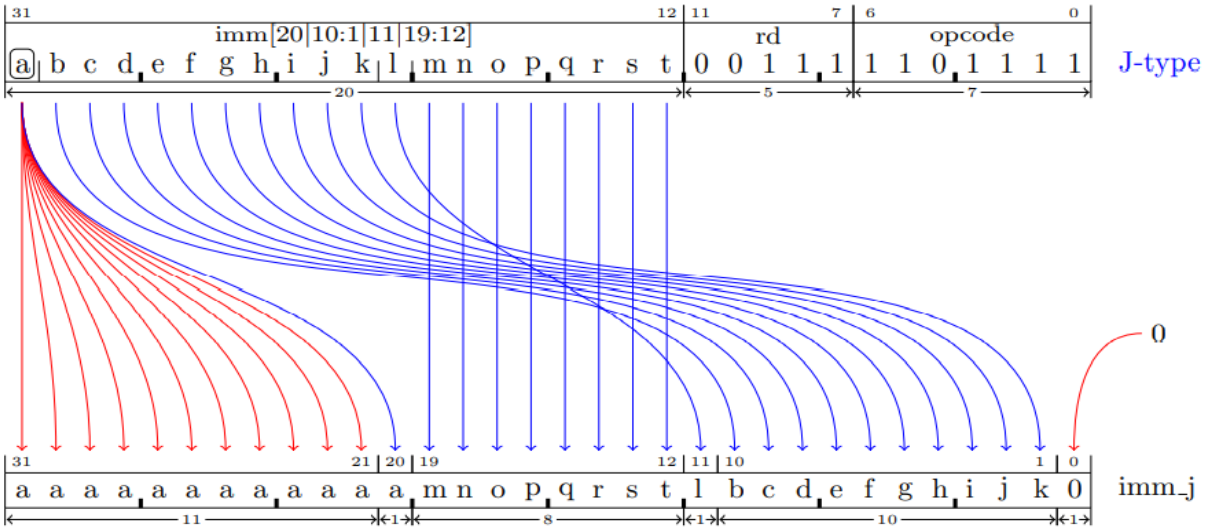
Since this is only a wire operation. Time delay is negligible

Fully asynchronous module.

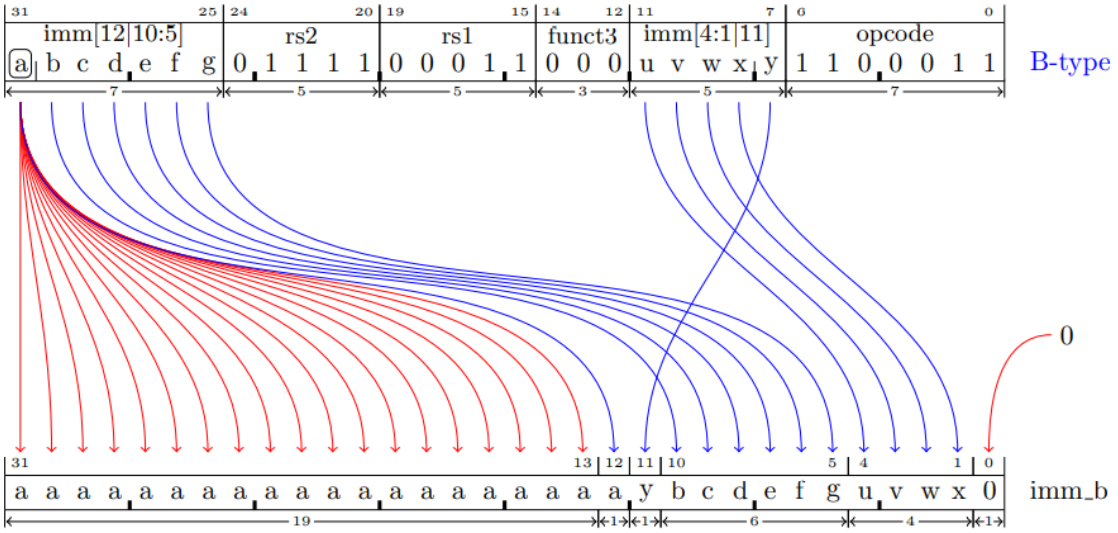
Mux(mux 2) is introduced to select between each generated immediate.

Design decisions

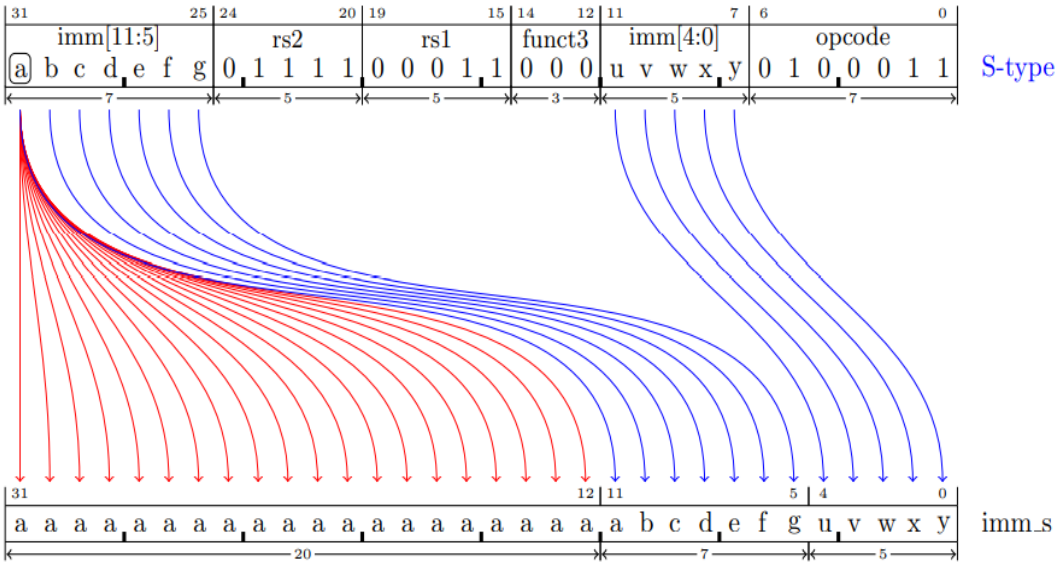
5 types of immediate value extensions are identified, namely
i) Sign extended J_imm :-



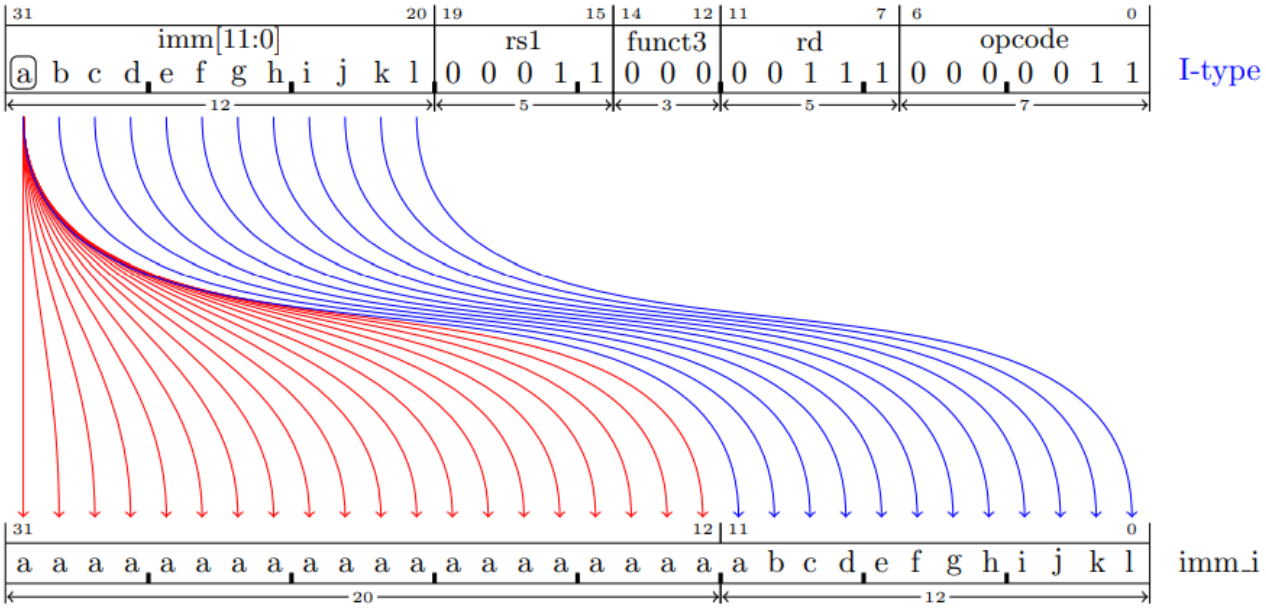
ii)Sign extended B_imm :-



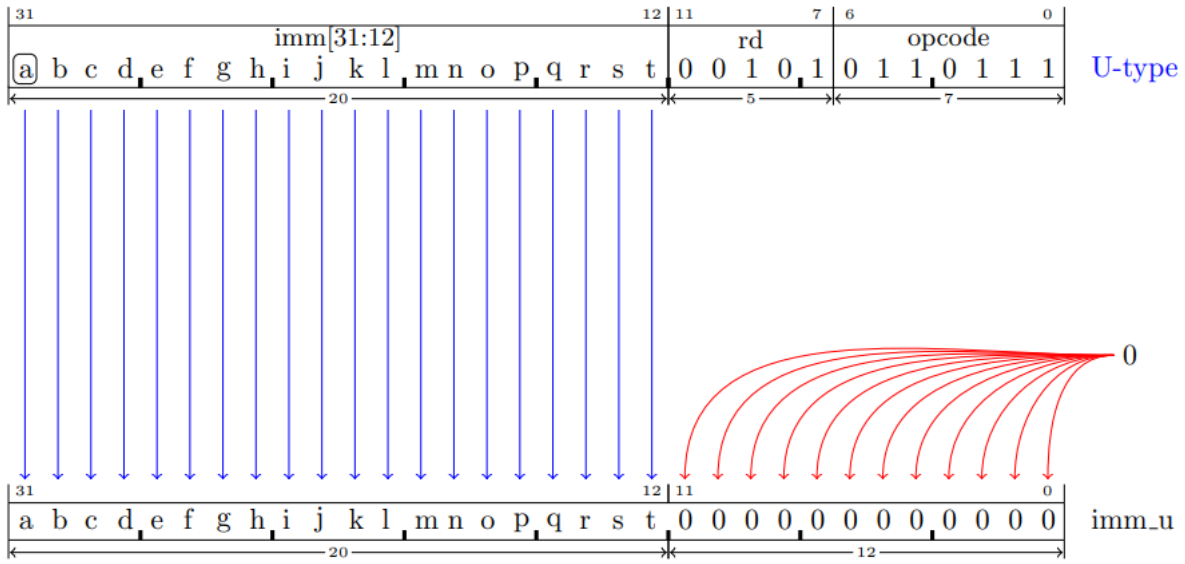
iii)Sign extended S_imm :-



iv)Sign extended I_imm :-



V)Zero extended U_imm :-



2.4 Dedicated Adder Module

This adder is used to add B_imm and current PC to generate branch address. This adder takes #2 delay.

Stage 2 time diagram

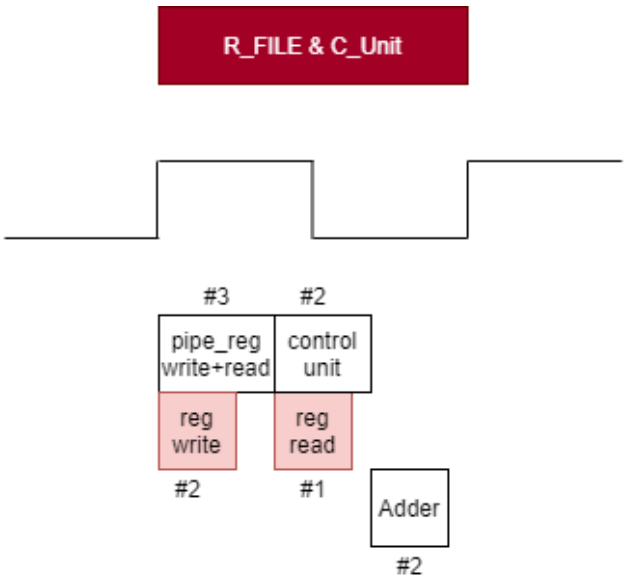


Figure 05

3. ALU stage

Modules in this Stage

i). ALU module

ii). Branch and jump module

3.1 ALU Module

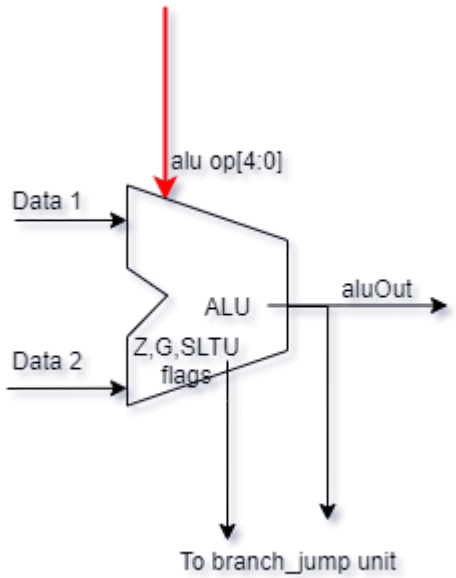


Figure 06: Alu module

I/O Signals

Inputs:

Data 1 and Data 2 : two 32 bit inputs as operands for calculations

ALUop : 5 bit control signal from control unit to resolve the operation

Outputs:

aluOut : 32 bit output from the alu

Z flag : 1 bit zero flag

G flag : 1 bit sign signal- set to 1 when aluOut is negative and set to 0 when aluOut is positive

K flag : 1 bit signal generated from sltu operation

Implementation & Timing

Fully asynchronous module.

For each operation, a separate opcode is there from the control unit. For addition and subtraction, we introduced #2 time unit delay. For logic operations, it would be less delay than add and sub. So we introduced #1 time delay. The most time-consuming operations are multiplication and division because the operation is complex. So we introduced #3 time units for this multiplication and division. Shift operation also does not consume much time so given #1 time unit delay for that.

Design Decisions

For every branch type instruction, ALU will do SUB operation in two operands given. Doing this will set the Z, G, K flags appropriately. These flags are used to resolve condition operations in *branch and jump unit*.
Z flag is set to one when SUB operation returns 32 bit 0s.
G flag is just to get the sign bit of the AluOut.
K flag is the zeroth bit of AluOut of SLTU operation

Workload of the ALU,

1) For jump instructions ALU calculates the jump address.
2) For load & store instructions ALU calculates data memory address.
3) For Branch instructions ALU does the SUB operation and set Z, G, K flags
4) For I type, R type, and M type instructions ALU does the relevant operation for each instruction. Such as ADD, SUB, SLL, etc.

For each ALU operation, we have given different ALUop from the control unit. Those are,

ADD : 00000
SUB: 00001
AND : 00010
OR : 00011
XOR : 00100
SLL : 00101
SRL : 00110
SRA : 00111
MUL : 01000
MULH : 01001
MULHU : 01010
MULHSU : 01011
DIV : 01100

DIVU : 01101
REM : 01110
REMU : 01111
SLT : 10000
SLTU : 10001

Note: forward operation is taken out from the ALU and forward operation is handled by multiplexer number 4 (mux 4 in the datapath) not complicating inner hardware in the ALU.

3.2 Branch Jump Module

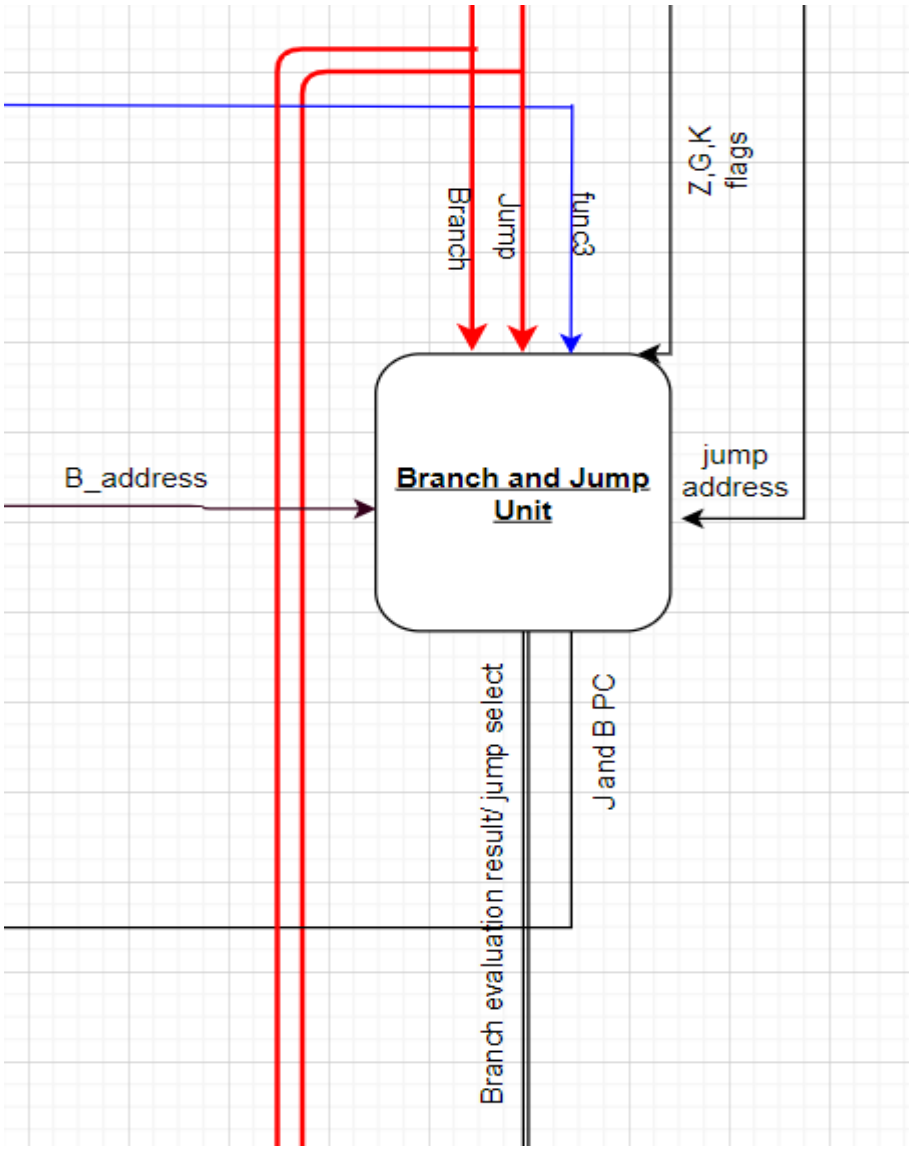


Figure 07: Branch jump module

I/O Signals

Inputs:

RESET : Reset bit
B_address : 32 bit branch address calculated in previous stage
Alu_Jump_imm : calculated jump address from ALU
Func_3 : Funct 3 field from the instruction to determine branch conditions
Branch_signal : to identify the instruction as branch
Jump_signal : to identify the instruction as jump
Zero_signal, sign_bit_signal, sltu_bit_signal : signals from ALU to determine conditions in branch signals

Outputs:

Branch_jump_PC_OUT : 32 bit modified PC to the instruction fetch module

Branch_jump_mux_signal : select signal to PC mux in instruction fetch module. This tells the fetch module to get the modified PC instead of PC + 4

Implementation & Timing

This module workload,

i). for jump instructions : send the modified PC to the instruction fetch module

ii). for branch instructions : evaluate the condition and send the modified PC from the adder to the instruction fetch module.

PC modification

PC modification is done within the ALU for jump instructions

In this Branch instructions, ALU's work is to set the flags that can be used in the branch jump module to evaluate conditions.

Handling Jump instructions

For every jump instruction this module will forward the alu out to the instruction fetch module.

Condition evaluation

For branch instructions condition evaluation is an asynchronous operation with #1 delay.

Also, funct 3 field is used to check which condition should be satisfied before taking the branch.

For this, following naming structure used in the ALU,

Zero signal : Z flag

Sign bit signal : G flag

Sltu bit signal : K flag

These flags are generated by ALU by doing SUB operation in DATA 1 and DATA 2 values.

Following table shows how these flags used to evaluate condition this,

Instruction Type	Zero bit flag	Sign bit flag	SLTU bit flag	Func3 field value
BEQ	1	X	X	000
BGE	X	0	X	101
BNE	0	X	X	001
BLT	0	1	X	100
BLTU	0	X	1	110
BGEU	X	X	0	111

Note: above table shows what should be the flags in order the condition to be true and valid.
For example, in order to BEQ instruction's condition to be true only zero flag is used and it should be 1. Equality is ok but to change the PC the instruction should BEQ. this is checked using funct 3 field value. If all is ok we set Branch_jump_mux_signal(PC mux select) so that next PC would be the modified PC.

When RESET, Branch_jump_mux_signal is set to 0 so that PC+4 value will take effect after.

The output "Branch_jump_PC_OUT" is set asynchronously with #2 time unit delay considering whether the instruction is a jump or branch instruction.
Therefore overall #3 unit time delay is needed by branch jump unit to generate signals.

Stage 3 time diagram

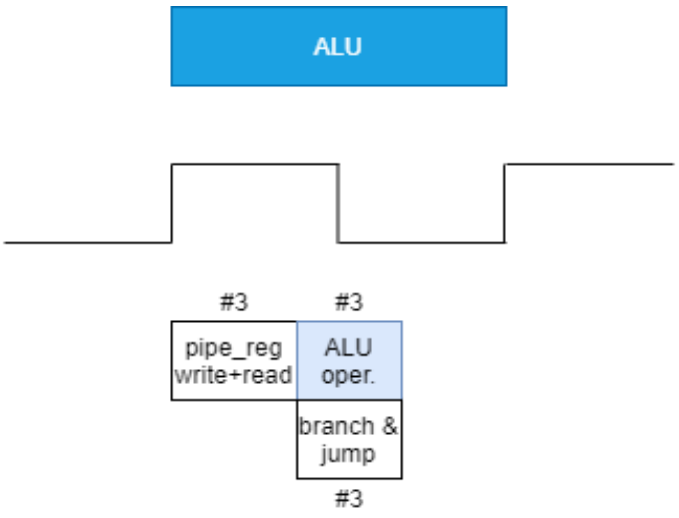


Figure 08

4. Data memory Access stage

Modules in this Stage

i). Cache module
ii). Cache controller module
iii). Data memory module
iv). Data refine module

4.1 Cache module

I/O Signals

Inputs:

Clock : clock signal
Reset : reset signal
read : read signal
Write : write signal
Address : 32 bit address from ALU
Writedata : 32 bit writedata
Inst_hit : hit signal from the instruction cache to identify the current instruction is legit.

mem_Readdata : 127 bit read data from data memory
mem_BusyWait : Data memory busywait

Outputs:

Busywait : cache busywait
Readdata : 32 bit readdata
mem_Read : read signal for data memory
mem_Write : write signal for the data memory
mem_Address : block address of the required block
mem_Writedata : 127 bit write data to the memory

Implementation & Timing

Our cache can hold 4 words per block. Each block is 127 bits wide. Therefore the offset is 2 bits wide.

Total 8 blocks are there in the cache. Therefore INDEX is 3 bits wide.

Since the cache is communicating with the CPU with words, the least significant 2 bits of the 32-bit address is ignored.

Tag is (32 - 2(ignored) - 2(offset) - 3(index) = 25), 25 bits wide.

When an address is received to the cache address is decoded according to INDEX,TAG, and OFFSET. This decode cost #1 time delay.
Another #1 delay is given to set the read,write and busywait registers.
After that, TAG comparison takes place incurring #5 time delay. And evaluate hit status. This is an asynchronous operation

While the hit status evaluation going on data is extracted from the relevant block with #1 delay. This is also an asynchronous operation. Also, this operation is overlapped with hit status evaluation.

At the positive clock edge(end of the memory access clock) if the instruction is a readacces and its hit, the cache memory busywait will be de-asserted. If the instruction is a writeaccess and a hit, busywait deasserted and the write operation takes place from there onwards with #2 time delay..

If the address is resolved as a miss, the rest of the workload goes to the cache controller. It is responsible for generating control signals for the data memory

Overall cache needs #7 time unit delay to fetch data for a hit address.

Since this stage have largest delay #3 + #7 = #10 time units. We set the clock to #10

4.2 Cache Controller module

I/O Signals

Inputs:

Clock : clock signal
Reset : reset signal
read : read signal
Write : write signal
Address : 32 bit address from ALU
Writedata : 32 bit writedata
Inst_hit : hit signal from the instruction cache to identify the current instruction is legit.

mem_Readdata : 127 bit read data from data memory
mem_BusyWait : Data memory busywait
Tag1,Index : need to create the block address of writeback state
writedata1: write data that should be written back to data memory at write back state
Tag : this is needed to create block address in memory read state
Hit,dirty : these signals used to determine state changes

Outputs:

Busywait : cache controller busywait
mem_Read : read signal for data memory
mem_Write : write signal for the data memory
mem_Address : block address of the required block
mem_Writedata : 127 bit write data to the memory

Implementation & Timing

Whenever a miss is sensed by the cache memory module, our cache controller triggers.

The Cache controller is designed as a 3 stage Finite State Machine. Stages Namely

1.IDLE:- State where cache controller lies when hits are handled by cache memory.no controls signals generated to trigger data memory.

2.MEM_READ:- State where the controller goes to when a miss occurred and additionally dirty bit deassertion is checked before changing state to MEM_READ at the upcoming clock posedge. In this state control signals are generated in order to receive a corresponding data block from data memory. state machine stays here until data memory busywait dessert.

3.WRITE_BACK:-When there's a miss AND the particular block is dirty, state machine goes to this state. Here data memory write-back operation takes place updating control signals (TAG and WRITEDATA) according to THAT particular block. Right after it finishes state changes to MEM_READ to get the required data block from memory. updating the TAG extracted from the address received from the CPU.

After MEM_READ state is done with its job, the state changes to IDLE at the next posedge of the clock. Then hit is again checked automatically since it is asynchronous. After HIT is resolved, data out or write occurs accordingly as same as the way when a hit occurs.

State machine diagram

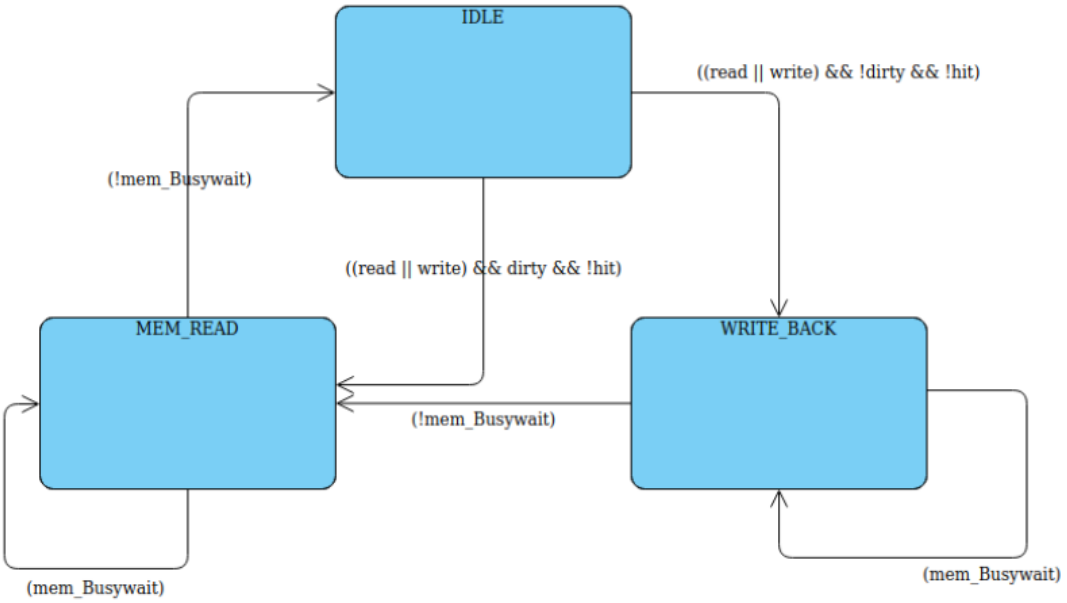


Figure 09: FSM diagram

4.3 Data memory module

I/O signals

Inputs:

Clock : clock signal
Reset : reset signal
Read : Read signal
Write : write signal
Address : block address (combination of required TAG and Index)
Writedata : 127 bit block of data that should be written to the datamemory

Outputs:

busywait : this is for the cache controller to say that instruction memory is busy fetching data. The cache controller waits until this signal to de-asserted to change state from MEM_READ to IDLE.

readdata : 127 bit block read data

Implementation & Timing

The Data Memory is byte addressed. Total of 256 registers of 127bit wide.
Since data memory and cache communicate in blocks each read or write request from cache involves a 127-bit block.

The block address is 27 bit wide.

Data memory fetches data byte by byte until the whole block is fetched.
Each byte fetch is costs #40-time units. Meaning, for the whole block to be fetched there will be a total of 40 x 16 time units delay.
Also when writing to the data memory each byte write costs #40 time units.
Therefore 40x16 time unit delay is there too.

4.4 Data refine module

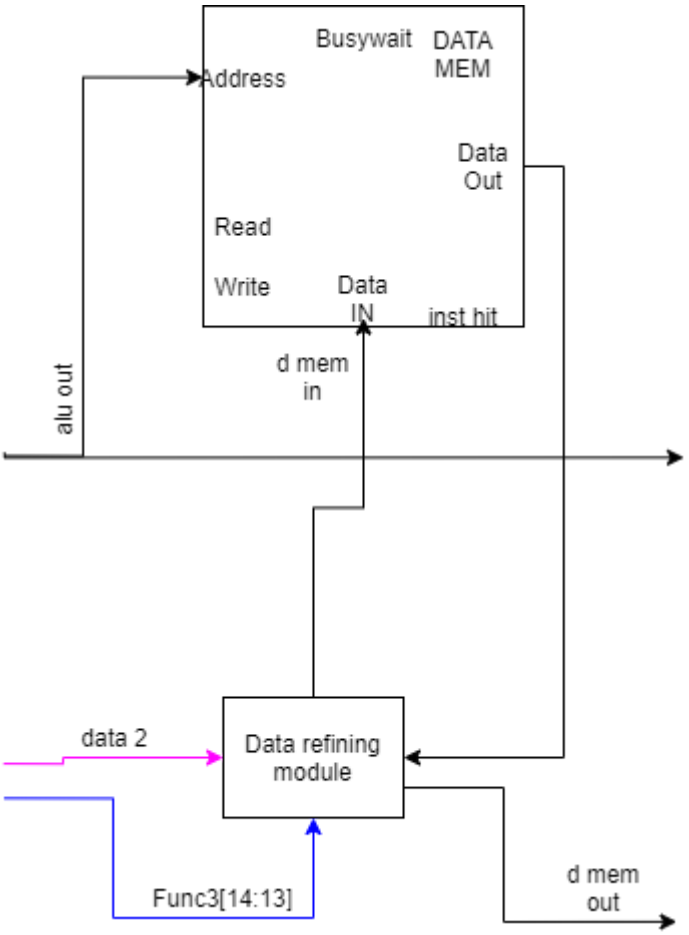


Figure 10: Data refine module

I/O Signals

Inputs:

Func3 : 3 bits from function 3 field to determine byte, half word or full word
Data_mem_in : 32 bit data from data memory
DATA2 : 32 bit DATA received from stage 3

Outputs:

To_data_memory : 32 bit data to the data memory to write in memory
Data_ref_out : 32 bit output from data refine module to the next pipeline register

Implementation & Timing

This module refines writedata to the data memory and readdata from the data memory before giving it to the CPU.

Since this is only wire operations delay is negligible.

This module operates in the CPU side.

By introducing a separate module for this we ease out the workload of the cache memory improving performance of the CPU.

In “STORE” instructions,

SB is refined as 0 extend to most significant 24 bits of the DATA2 value

SHW is refined as 0 extend to most significant 16 bits of the DATA2 value.

SW is not refined just forwarded to the Cache memory

In Load instructions,

LBU is refined as 0 extend to most significant 24 bits of the output of data memory

LHU is refined as 0 extend to most significant 16 bits of the output of data memory.

LW is not refined just forwarded to the to the next pipeline register.

LB is refined as sign extend to most significant 24 bits by the seventh bit of the output of data memory.

LH is refined as sign extend to most significant 16 bits by the 15th bit of the output of data memory.

Stage 4 time diagram

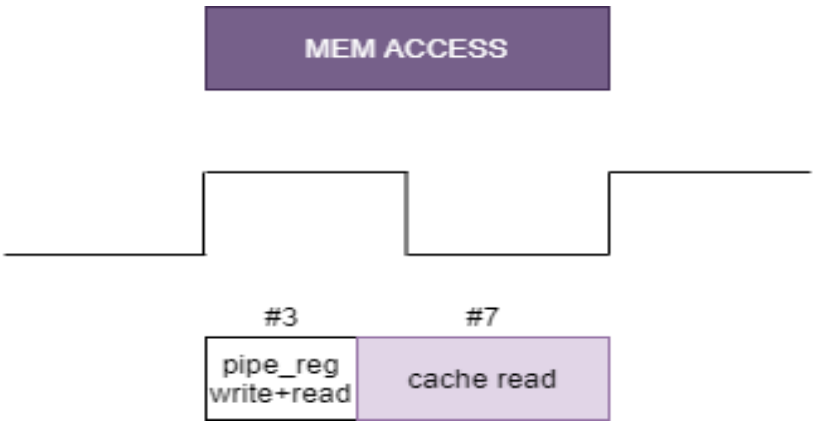


Figure 11

5. Write Back Stage

In this stage, only the multiplexer is there(Mux3).
Multiplexer delay is negligible compared to the other modules

Stage 5 time diagram

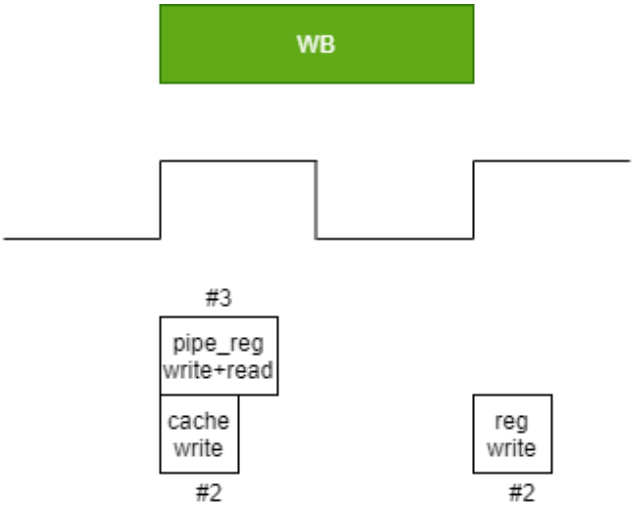


Figure 12

6. Multiplexers

There are three types of multiplexers,

Type 1 :- two 32 bit inputs, 1 bit select signal, 32 bit output
mux 1, mux 5, mux 3 and pc mux are this type of multiplexers (in the datapath)

Type 2 :- five 32 bit inputs, 3 bit select signal, 32 bit output
mux 2 is this type of multiplexer (in the datapath)

Type 3 :- three 32 bit inputs, 2 bit select signal, 32 bit output
mux 4 is this type of multiplexer (in the datapath)

Multiplexers incur negligible delay when operating.

Mux4 is selecting between ALUout, forwarding operation, and PC+4 value.

Forwarding is extracted out from the ALU so that ALU electronics will be simple.

7. Pipeline registers

A total of 4 pipeline registers were used. Each pipeline read operation costs #1 time unit delay and write operation costs #2 time delay. Therefore total #3 time units are used to get data from pipeline registers to the current stage.

When resetting only the "*instHit*" bit is set to 0 which invalidates any instruction already stored in the pipeline register. Changing only 1 bit is considered negligible time delay operation.

8. Assembler

Our assembler is coded using C++.
Every Assembly code instruction should be in capital form.
Every immediate value should be given in the hexadecimal form.
Here Nope instruction is a 32 bit 0.

Code rescheduling functionality is not included yet. But we are planning to have rescheduling functionality in future in order to avoid the unavoidable bubble in load-use hazards.

Hazard Handling

Mainly 2 types of Hazards identified,

1. Data hazard : Hazards occur due to use of data from registers before updating most recent updated values to registers

2. Control hazard : Hazards occur due to branching and jumping

Data Hazard Handling

There are 2 types of Data hazards identified,

1. ALU data hazard : Hazards occur due to use of data calculated by ALU before updating to register file

2. Load-Use data hazard : Hazards occur due to use of read data from data memory, but before updating the register file.

For each type we introduce a separate module. These modules are not in the pipeline.

ALU data Hazard Handling Module

We introduced a separate module for this.

I/O signals

Inputs:

Clock : clock signal
Reset : reset signal
destination_address_mem_stage : rd retrieved from stage 4(mem access stage)
Destination_address_alu_stage : rd retrieved from stage 3(alu stage)
Rs1_address_id_stage : rs1 retrieved from stage 2
Rs2_address_id_stage : rs2 retrieved from stage 2

Outputs:

Forward_enable_to_rs1_from_mem_stage_signal : select signal for mux
Forward_enable_to_rs2_from_mem_stage_signal : select signal for mux
Forward_enable_to_rs1_from_wb_stage_signal : select signal for mux
Forward_enable_to_rs2_from_wb_stage_signal : select signal for mux

Functionality

This module have two functionalities,

1. Hazard detection
2. Hazard handling

Hazard detection

Hazard detection is done by comparing destination registers and the source registers from different stages.

Mainly two hazards are detected by this module,

1).Hazard occur when use the calculated value from the ALU in the next instruction

Detection done by comparing rd register in alu stage(stage 3) and 2 source registers in instruction decode stage(stage 2)

Sample Assembly code 1 :

```
ADD R1, R2, R3
SUB R4,R1,R2
```

Here the hazard is in rs1 register, it is modified by ALU by adding R2 and R3. But it is used right next instruction for SUB operation before register file updates it with most recent updated value.

Important : In SUB operation R1 is used in rs1 field of the instruction.

Sample Assembly code 2 :

```
ADD R1, R2, R3
SUB R4,R2,R1
```

Here R1 is reused in the rs2 field of the instruction. This difference is important in generating select signals for muxes.

2).Hazard occur when use the calculated value from the ALU in one after next instruction.

Detection comparing rd register in memory stage(stage 4) and source register in instruction decode stage(stage 2)

Sample Assembly code 3 :

```
ADD R1, R2, R3
MUL R5,R6,R7        //some other instruction
SUB R4,R1,R2
```

Here the hazard is also in R1 register, it is modified by ALU by adding R2 and R3. But it is used right after 2 instructions for SUB operation before register file updates it with most recent updated value.(i.e Register file reads R1 when the updated value is in Memory stage)

Important : In SUB operation R1 is used in rs1 field of the instruction.

Sample Assembly code 4 :

```
ADD R1, R2, R3
SUB R4,R2,R1
```

Here R1 is reused in the rs2 field of the instruction. This difference is important in generating select signals for muxes. Same as before.

These are the hazards we identified using this module. Even Though there are two types of hazards here. We need to generate mux select signals for 4 scenarios given that hazardous register usage is in rs1 field or rs2 field.

Hazard handling

We generated mux select signals to direct forwarded values from data memory stage and write back stage.

Note: even though we detect hazards using register addresses in alu stage and memory stage, we have to forward the value from data memory stage and write back stage respectively. Because the detection happens 1 clock cycle before the actual hazard in the pipeline.

As i mentioned before there are 4 scenarios we need to address in hazard handling,

Scenario 1

Hazard occur when use the calculated value from the ALU in the next instruction. The hazardous register is used in the rs1 field in the next instruction.

Solution : Forwarded value from the data memory stage should be given as output from mux7. So we gave relevant select signal to mux7.

Scenario 2

Hazard occur when use the calculated value from the ALU in the next instruction. The hazardous register is used in the rs2 field in the next instruction.

Solution : Forwarded value from the data memory stage should be given as output from mux6. So we gave relevant select signal to mux6.

Scenario 3

Hazard occur when use the calculated value from the ALU in one after next instruction. The hazardous register is used in the rs1 field in the next instruction.

Solution : Forwarded value from the write back stage should be given as output from mux7. So we gave relevant select signal to mux7.

Scenario 4

Hazard occur when use the calculated value from the ALU in one after next instruction. The hazardous register is used in the rs2 field in the next instruction.

Solution : Forwarded value from the write back stage should be given as output from mux6. So we gave relevant select signal to mux6.

Special Notes:

- **if hazards occur in rs1 and rs2 simultaneously, it will be handled separately. Therefore no issues occur.**

Sample Assembly code 5 :

```
ADD R1,R2,R3
SUB R2,R3,R5
ADD R4,R2,R1
```

Here at the third instruction, R1, R2 both are hazardous register addresses. Both hazards should be handled at the same time.Since our module handle rs1 and rs2 hazards separately no issue occurs.

- **If the module detects hazards from alu stage and memory stage both with the same rs1 or rs2 field, the priority will be given to the hazard with alu stage.**

For example, consider following code block.

Sample Assembly code 6 :

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

here , hazard is detected within instruction 1 and 2(at this time instruction 1 will be in alu stage, instruction 2 will be in memory stage). After hazard detection hazard handling will be done in the next clock cycle by forwarding the respective value from memory stage.
Now, instruction 1 is in the memory stage, instruction 2 is in ALU stage, instruction 3 will be in instruction decode stage. At this time, the hazard unit indicates hazards in both memory

stage instruction and alu stage instruction with the instruction in instruction decode stage.
Note: the R1 register is used in same rs2 field in instruction 2 and instruction 3.
At this time the forwarding should be allowed to the forwarded value from memory stage.
This is done by duplicating the mem stage forwarded value in the multiplexer(mux7 and mux6). Therefore at this type of a scenario the hazard unit will output 2'b11 and this select signal gives the duplicated forwarded value as output from the multiplexer.

Timing

When hazard identification we bitwise XNORed and then did AND operation. This XNORing & ANDing takes # time unit delay each.
Every positive clock edge if there's a hazard, then setting relevant outputs takes #1 combination logic delay

Load-Use data Hazard Handling Module

We introduced a separate module for this too

I/O signals

Inputs:

Clock : clock signal
Reset : reset signal
Load_signal : to identify the instruction is a load instruction
Memstage_Rd : destination register address retrieved from memory access stage
ALustage_Rs1 : source register address retrieved from ALU stage
ALustage_Rs2 : source register address retrieved from ALU stage

Outputs:

Enable_rs1_forward_from_wb : signal to mux select
Enable_rs2_forward_from_wb : signal to mux select
enable_bubble : introduce a bubble to pipeline register 3 and hold the pipeline register 1 and 2.

Functionality

This module have two main functionalities.

1. Hazard detection
2. Hazard handling

Hazard detection

Hazard detection is done by comparing destination register address from the memory access stage and source register addresses from alu stage.
If even one source register address match with destination register address from the memory stage a hazard occur. But there should be another requirement to detect the hazard. The instruction should be a load instruction too. We check this requirement using read signal from the control unit.

Here the module should generate different signals to muxes considering wether the use of the Hazardous register in the rs1 field or rs2 field in the instruction.

Sample Assembly code 4 :

```
LW R1, 0x77(R2)
ADD R4,R2,R1
```

Sample Assembly code 5 :

```
LW R1, 0x77(R2)
ADD R4,R1,R2
```

Above sample code blocks show how this hazard occurs.

Hazard Handling

To handle this we need to forward the read data of data memory from the write back stage, though we detect the hazard in the previous stage. Also it is important to note that, when we forward the read data from write back stage, the instruction that use that data, should be stalled in the ALU stage. Otherwise at the time of forwarding happen, the use instruction may passed to the memory access stage with wrong values. Therefore we use 'enable bubble' signal to insert a bubble to pipeline register 3 and with the same signal we stall pipeline register 2 and 1 by setting busywait signal of respective pipeline register.

At the end it looks like, we rearrange the sample assembly code 4,

```
LW R1, 0x77(R2)
    bubble
ADD R4,R2,R1
```

This bubble is unavoidable because the reading operation takes the whole clock signal time.

The only way to avoid this would be to get the support from the assembler.when generating machine code the assembler can rearrange the assembly code such that this hazard is avoided. This can be done when there is not any dependency between instructions near the load-use instructions.If there is a dependency even if we have that support from the assembler, this hazard cannot be fully avoided. Therefore at that time, we need to insert a bubble to the pipeline.

What will happen if load use happen in following way?

Sample Assembly code 6 :

```
LW R1, 0x77(R2)
SUB R5,R6,R7            //some other instruction
ADD R4,R2,R1
```

This is exactly the same scenario which was described as scenario 3 and scenario 4 in ALU data hazard section previously. Therefore this hazard is handled by ALU data hazard module already.

Timing

This is the same as ALU data hazard module.
When hazard identification we bitwise XNORed and then did AND operation. This XNORing & ANDing takes # time unit delay each.
Every positive clock edge if theres a hazard, then setting relevant outputs takes #1 combination logic delay

Control Hazard Handling

There are 2 occasions of control hazards identified,

1. Branching : Handled by Dynamic Branch Prediction Unit
2. Jumping : Handled by Branch and Jump unit

Dynamic Branch Prediction Unit

In this module we dynamically predict whether to take the branch or not.

I/O signals

Inputs:

ID_pc : PC from Instruction decode stage
ALU_pc : PC from ALU stage
Reset : Reset signal
ID_stage_branch : Instruction decode stage branch signal
ALU_stage_branch : ALU stage branch signal
ALU_stage_branch_result : condition evaluation from ALU stage (condition true or false)

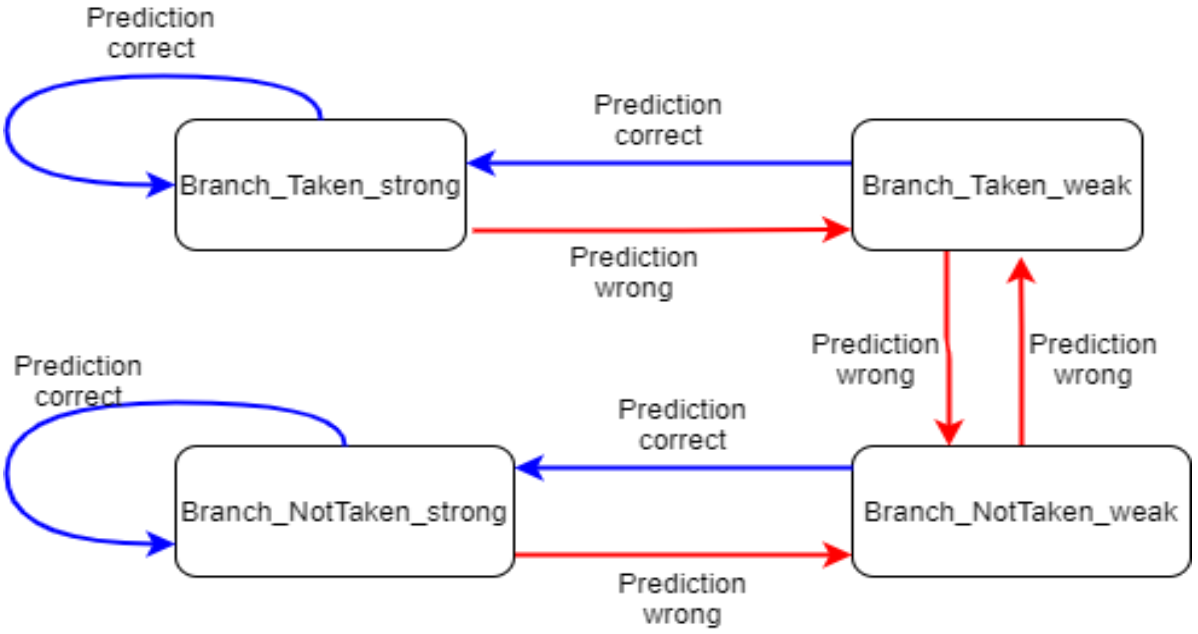
Outputs:

Flush : to flush the Pipeline registers and select bit to mux 10
Early_prediction_is_branch_taken : select bit to mux 9
Signal_to_take_branch : select bit to mux 8

Functionality

This is a fully asynchronous module
In this module we dynamically predict whether
to take the branch or
Not to take branch.

Take branch or not take branch decision is based upon the how wrong is the prediction is. This can be easily understood by the following diagram.



As shown in the diagram, if we are to take_branch_strong or take_branch_weak stage, we predict to take the branch. Then we check wether our prediction is correct, if we are wrong we reassign stages according to above diagram.
To store the stage for a particular PC, we have a buffer named as branch target buffer. In our implementation, our buffer can hold prediction stages for 8 different PC addresses. Therefore, We used the least significant 3 bits of PC as index to do the mapping from PC to prediction stage.

Implementation

For any branch instruction we predict ‘Branch_taken_strong’ as default. First, In the second stage, we check whether the newly fetched instruction is a branch instruction.We do this by checking the ‘ID_branch’ signal from the control unit. If it is a branch signal, our module look in the buffer to get the stage, using the PC value of Branch instruction. This is ‘ID_PC’ signal. Then according to the stage, the multiplexer select signals are set.

The multiplexer is mux8, it has two 32 bit inputs. One from the pc_mux and other from the dedicated adder which generates the branch address. If we predict the branch to take, the modified address will be forwarded from mux8, if not pc_mux output will be forwarded.(please look at the datapath).

After the prediction our module waits to get to know whether the prediction is correct or not. This is evaluated in the ‘branch and jump unit’ in the ALU stage. If the prediction is correct, the ‘Branch and jump’ unit will output a signal. This signal is common to jump and branch. Therefore we need to get the branch signal from the ALU stage to get to know whether this

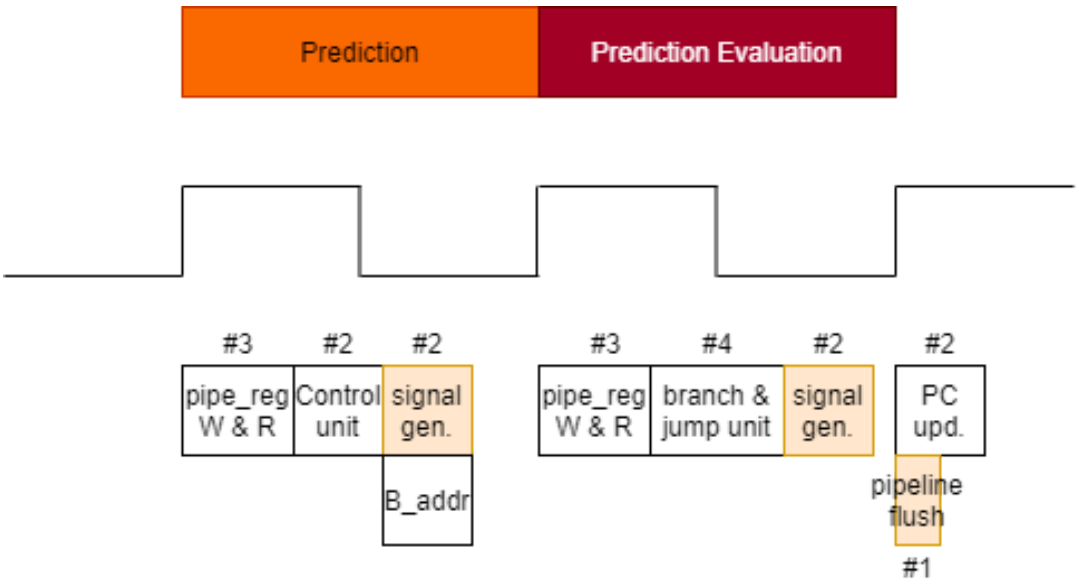
is branch instruction. Using these two signals(branch signal, and evaluation signal) we can decide whether the prediction is correct or wrong.
For example: if branch signal and branch condition evaluation signal are set and we predicted to take branch ---- our prediction is correct
if branch signal is set and branch condition evaluation signal is zero and we predicted to take branch ---- our prediction is incorrect (condition evaluation signal is zero means the condition is false.)
if branch signal is zero and branch condition evaluation signal is set --- this means instruction is not a branch instruction. In particular, it's a jump instruction.

After we evaluate whether our prediction is correct or wrong. The module changes the stage of the particular branch PC according to the diagram before. If our prediction is correct, stage changing is enough. But if our prediction is wrong, we have to flush the pipeline register 1 and 2 and give the correct value to the PC to fetch the correct instruction. We do this using ‘flush’ signal. This signal is not only flush the pipeline register but also acts as a select signal to mux10.

Also, if we predict wrong, the correction would be to take the branch or not to take branch. This depends on what we predict. If we predict to take the branch and our prediction is wrong the correction would be giving PC+4 or otherwise if we predict to not take branch and our prediction is wrong the correction would be giving the modified address by the adder. To choose between these two (PC+4 value or modified PC value) we introduced another mux(mux9) and we generated a select signal for it from the prediction module. This PC+4 value and modified PC value should be taken from the ALU stage. In particular Modified PC value is taken from the ‘Branch and jump unit’. After selecting which value should be inserted to PC.The flush signal flushes out the instructions fetched because of wrong prediction and give the correct address to PC through mux10.

In the second stage right after control signal generated stating that the instruction is ‘Branch’ the unit generates outputs relevant to prediction. This generation takes #2 time delay. Parallel to this the dedicated adder in stage two calculates branch address with #2 time unit delay.
At the next clock edge this prediction evaluates. After branch and jump unit generate its output the branch prediction unit generate necessary outputs given that the prediction is correct or wrong. Specially if prediction is wrong. Flushing and relevant mux select signals should be generated. This includes #2 time delay. At the next positive clock edge, the pipeline flush and correct PC update occurs. This involves #1 for flushing and #2 for PC updating respectively.

Timing

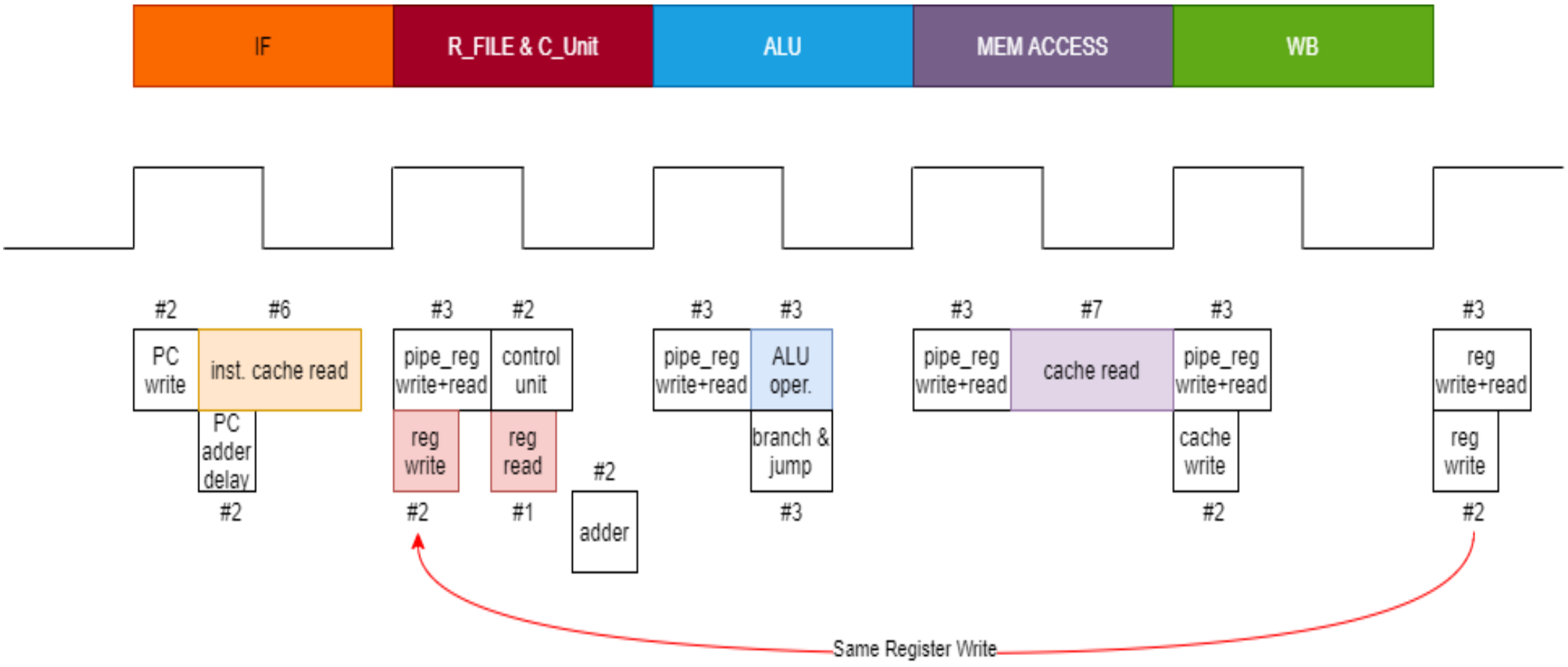


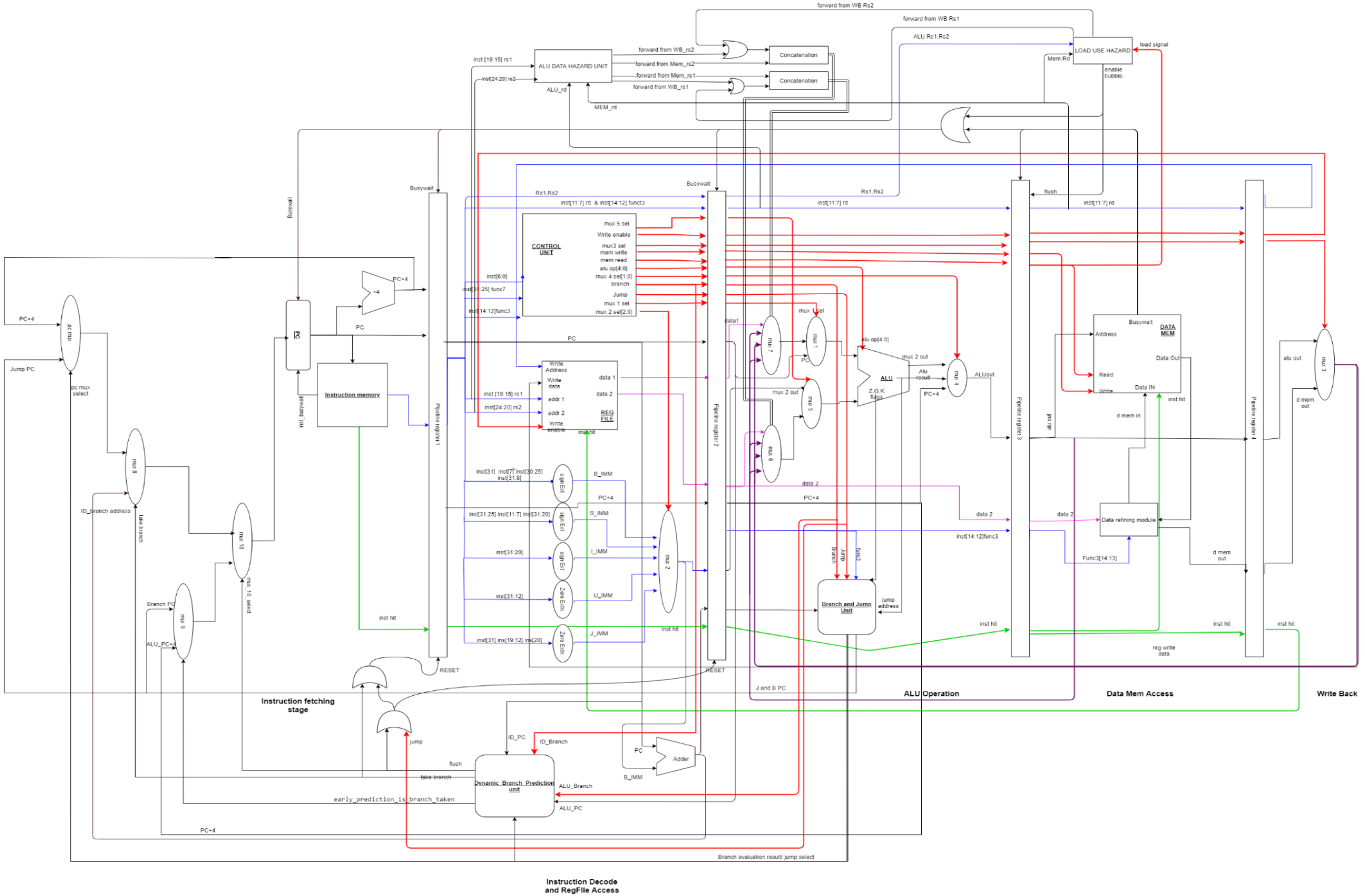
Jump Hazard Handling

Jump hazards are handled by our Branch and Jump unit. If the instruction is a jump, it forward the jump address calculated by ALU to pc_mux and set the jump indicator signal(this signal is same as branch condition evaluation signal) which flushes out pipeline register 1 and 2 and act as a select signal for pc_mux. Pc_mux will forward the Jump address to mux8 and it will go thorough mux 10 and get updated to PC.

Full Timing Diagram

The dominant module in each stage is highlighted.




forward from WB Rs2
forward from WB Rs1
ALU.Rs1,Rs2
load signal
LOAD USE HAZARD
Mem.Rd
enable
bubble
inst [19:15] rs1
inst[24:20] rs2
ALU DATA HAZARD UNIT
forward from WB_rs2
forward from Mem_rs2
forward from Mem_rs1
forward from WB_rs1
Concatenation
Concatenation
ALU_rd
MEM_rd
Busywait
Rs1,Rs2
inst[11:7] rd & inst[14:12] funct3
CONTROL UNIT
mux 5 sel
Write enable
mux3 sel
mem write
mem read
alu op[4:0]
mux 4 sel[1:0]
branch
Jump
mux 1 sel
mux 2 sel[2:0]
inst[6:0]
inst[31:25] func7
inst[14:12]func3
PC+4
+4
PC
pc mux
Jump PC
pc mux select
Instruction memory
Pipeline register 1
Write Address
Write data
addr 1
addr 2
Write enable
data 1
data 2
REG FILE
inst hit
inst [19:15] rs1
inst[24:20] rs2
mux 8
ID_branch address
take branch
mux 10
mux 10 select
mux 9
Branch PC
ALU_PC+4
inst hit
RESET
inst[31] inst[7] inst[30:25] inst[31:8]
sign Ext
B_IMM
inst[31:25] inst[11:7] inst[31:20]
sign Ext
S_IMM
inst[31:20]
sign Ext
I_IMM
inst[31:12]
Zero Extn
U_IMM
inst[31] ins[19:12] ins[20]
Zero Extn
J_IMM
mux 2
inst hit
Pipeline register 2
data1
mux 7
mux 1
PC
mux 2 out
mux 5
mux 6
alu op[4:0]
ALU
Z,G,K flags
Alu result
mux 4
ALUout
PC+4
data 2
inst[14:12]func3
Branch
Jump
func3
Branch and Jump Unit
jump address
inst hit
J and B PC
Pipeline register 3
flush
inst[11:7] rd
DATA MEM
Busywait
Address
Data Out
Read
Write
Data IN
inst hit
d mem in
alu out
data 2
Data refining module
Func3[14:13]
d mem out
Pipeline register 4
alu out
mux 3
d mem out
inst hit
reg write data
ALU Operation
Data Mem Access
Write Back
Instruction fetching stage
jump
flush
take branch
ID_PC
ID_Branch
Dynamic Branch Prediction unit
early_prediction_is_branch_taken
ALU_Branch
ALU_PC
PC
Adder
B_IMM
PC+4
Branch evaluation result/ jump select
Instruction Decode and RegFile Access

DATAPATH V5

In case of not clear enough to observe above DATAPATH. We uploaded it in to Google drive. Can access through below link

DATAPATH V5.png