CprE 3810: Computer Organization and Assembly-Level Programming

Project Part 2 Report

Team Members: Jongwoo Kim

Veda Vegiraju

Project Teams Group #: Project Group F_04

Refer to the highlighted language in the project 1 instruction for the context of the following questions.

[1.a] Come up with a global list of the datapath values and control signals that are required during each pipeline stage.

IF stage (Instruction Fetch)

- **Datapath values**
  - PC_IF (current PC)
  - NextPC (output of fetch / branch–jump logic)
  - Inst_IF (instruction from instruction memory)
- **Control signals**
  - PCWrite (PC write enable)
  - PCSrc / branch-jump select (uses branch/jump decision from EX)
  - optional: IF_Stall, IF_Flush (for hazards later)

ID stage (Instruction Decode / Register Fetch)

- **Datapath values**
  - PC_ID
  - Inst_ID
  - Decoded fields: opcode, rd, rs1, rs2, funct3, funct7
  - Rs1Addr, Rs2Addr
  - ReadData1, ReadData2 (regfile outputs)
  - Imm_ID (immediate from imm_gen)

- **Control signals (generated in ID, passed forward)**
    - RegWrite_ID
    - MemRead_ID
    - MemWrite_ID
    - MemtoReg_ID[1:0]
    - ALUSrcA_ID
    - ALUSrcB_ID
    - ALUOp_ID[1:0]
    - Branch_ID
    - Jump_ID[1:0]
    - AUIPCSrc_ID (for auipc)
    - Halt_ID
    - For hazards/forwarding: Rd_ID, Rs1Addr_ID, Rs2Addr_ID

EX stage (Execute / Address Calculation)

- **Datapath values**
    - PC_EX
    - ReadData1_EX, ReadData2_EX
    - Imm_EX
    - Funct3_EX, Funct7_EX
    - Rd_EX
    - Rs1Addr_EX, Rs2Addr_EX
    - ALUInputA, ALUInputB
    - ALUResult
    - Condition flags: Zero, Sign, Cout
- **Control signals**
    - ALUSrcA_EX
    - ALUSrcB_EX
    - ALUOp_EX[1:0]
    - Branch_EX
    - Jump_EX[1:0]

    - MemRead_EX
    - MemWrite_EX
    - RegWrite_EX
    - MemtoReg_EX[1:0]
    - Halt_EX

MEM stage (Memory Access)

- **Datapath values**
    - PC_MEM
    - ALUResult_MEM (address or ALU result)
    - ReadData2_MEM (store value)
    - Imm_MEM
    - Funct3_MEM
    - Rd_MEM
    - DMemAddr (usually ALUResult_MEM)
    - DMemDataIn (store data, same as ReadData2_MEM)
    - DMemOut (raw memory read)
    - LoadData (output of load_extender)
- **Control signals**
    - MemRead_MEM
    - MemWrite_MEM
    - RegWrite_MEM
    - MemtoReg_MEM[1:0]
    - Halt_MEM

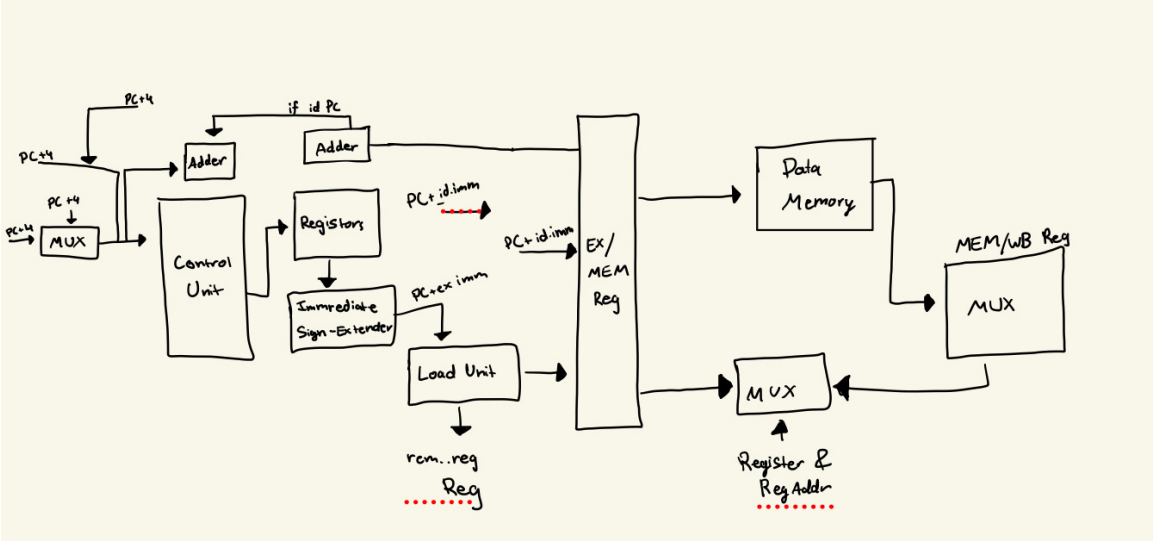
WB stage (Write Back)

- **Datapath values**
    - PC_WB
    - ALUResult_WB
    - LoadData_WB
    - Imm_WB
    - Rd_WB

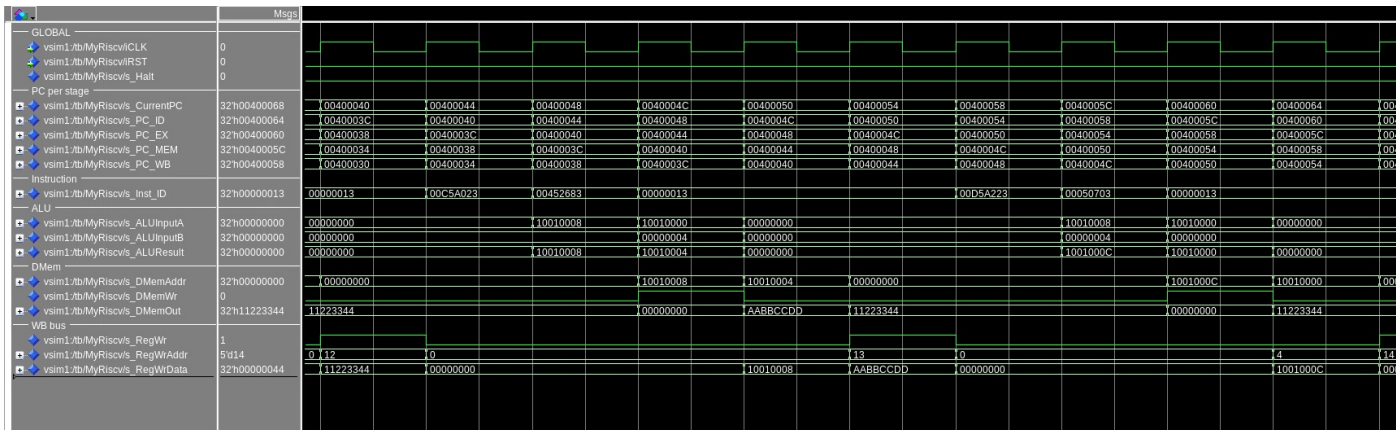
- RegWrData (final WB mux output to regfile)

- **Control signals**
  - RegWrite_WB
  - MemtoReg_WB[1:0]
  - Halt_WB

[1.b.ii] high-level schematic drawing of the interconnection between components.



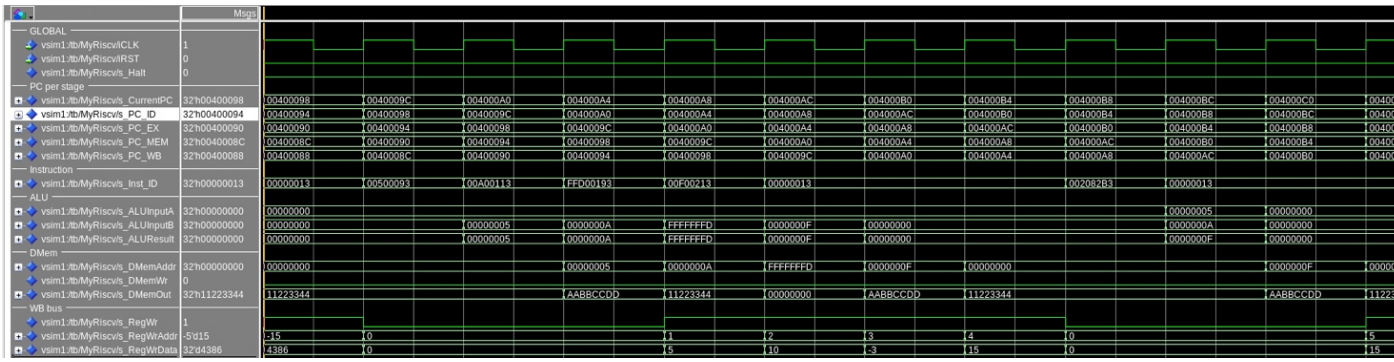
[1.c.i] include an annotated waveform in your writeup and provide a short discussion of result correctness.



1.Hazard-free-5-stage pipeline execution for the load/store memory test

The waveform in Figure 1 shows a portion of proj2_simple_full_test.s executing on my 5-stage RISC-V pipeline. The PCs for IF, ID, EX, MEM and WB are spaced by 4 bytes, so at each cycle five consecutive instructions occupy the five stages, confirming that the pipeline is issuing one instruction per cycle without hardware stalls. Around the highlighted region, the instruction in ID corresponds to the expected lw/sw sequence, and the ALU computes the correct effective addresses (e.g., 0x10010008 and 0x1001000C) from the base registers and offsets.

In the MEM stage, s_DMemAddr, s_DMemWr, and s_DMemOut show that the correct words 0x11223344 and 0xAABBCCDD are read from buf_src and then written to buf_dst. In the WB stage, s_RegWr, s_RegWrAddr, and s_RegWrData confirm that these same values are written back into the intended destination registers (x12 and x13). Because the test program is manually scheduled with NOPs between producer and consumer instructions, no RAW or control hazards appear, and all observed results match the expected behavior of the program.



2.ALU/Logic instruction Block from proj2_test.s

Figure 2 shows the execution of the basic ALU and logical instruction sequence from proj2_test.s. The **PC-per-stage signals** (PC_IF, PC_ID, PC_EX, PC_MEM, PC_WB) advance in a clean diagonal pattern, indicating that each instruction moves one pipeline stage per cycle with no unintended stalls; the only bubbles are the explicit addi x0,x0,0 NOPs inserted in the test program. The **Inst_ID** trace matches the expected instruction stream in this region (e.g., addi setup instructions followed by add, sub, and other ALU operations), confirming that the fetch and decode logic are correctly aligned with the PC.

Within the **ALU group**, ALUInputA and ALUInputB carry the expected operand values from the register file, and **ALUResult** shows the correct outputs for each instruction (for example, the sum and difference for add/sub, and the proper bit-wise results for and, or, and xor). On the **write-back bus**, the signals RegWr, RegWrAddr, and RegWrData demonstrate that these ALU results are written back to the correct destination registers exactly one cycle after the corresponding instruction is in the EX stage. Overall, Figure 2 demonstrates that, for the proj2_test.s ALU block, the 5-stage pipeline executes instructions in order, produces correct results, and performs register write-back with the expected timing and without unintended hazards.

[1.c.ii] Include an annotated waveform in your writeup of two iterations or recursions of these programs executing correctly and provide a short discussion of result correctness. In your waveform and annotation, provide 3 different examples (at least one data-flow and one control-flow) of where you did not have to use the maximum number of NOPs.

Fail to pass the test.

[1.d] report the maximum frequency your software-scheduled pipelined processor can run at and determine what your critical path is (specify each module/entity/component that this path goes through).

From the Quartus timing report, the maximum clock frequency **our** software-scheduled pipelined processor can run at is **59.53 MHz**, which corresponds to a data arrival time of 19.77 ns and a positive slack of 3.20 ns with respect to the 20 ns clock constraint.

The critical path starts in the **ID/EX pipeline register**, at

id_ex_reg:U_ID_EX|s_Funct7_reg[4], then goes through the following logic:

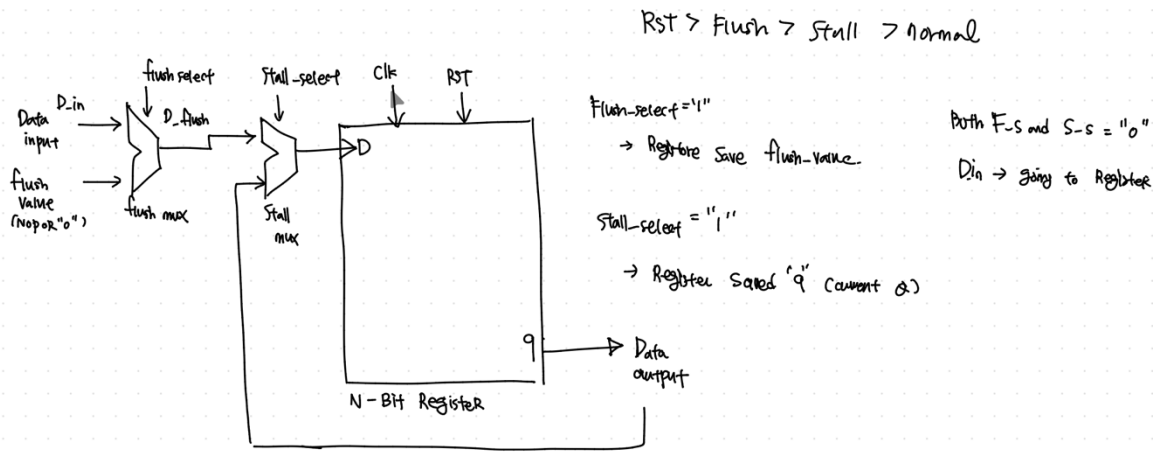
- the **ALU control unit** U_ALUCTRL (Mux7 chain that generates i_ALUCtrl),
- the **ALU adder datapath** U_ALU|U_ADDER (ripple-carry add chain),
- the **ALU result selection and comparison logic** (U_ALU|Mux47 and U_ALU|Equal4),
- the **fetch logic / next-PC generation** block U_FETCH|o_NextPC,

and finally arrives at the **PC register** bit pc_reg:U_PC|s_PC[9].

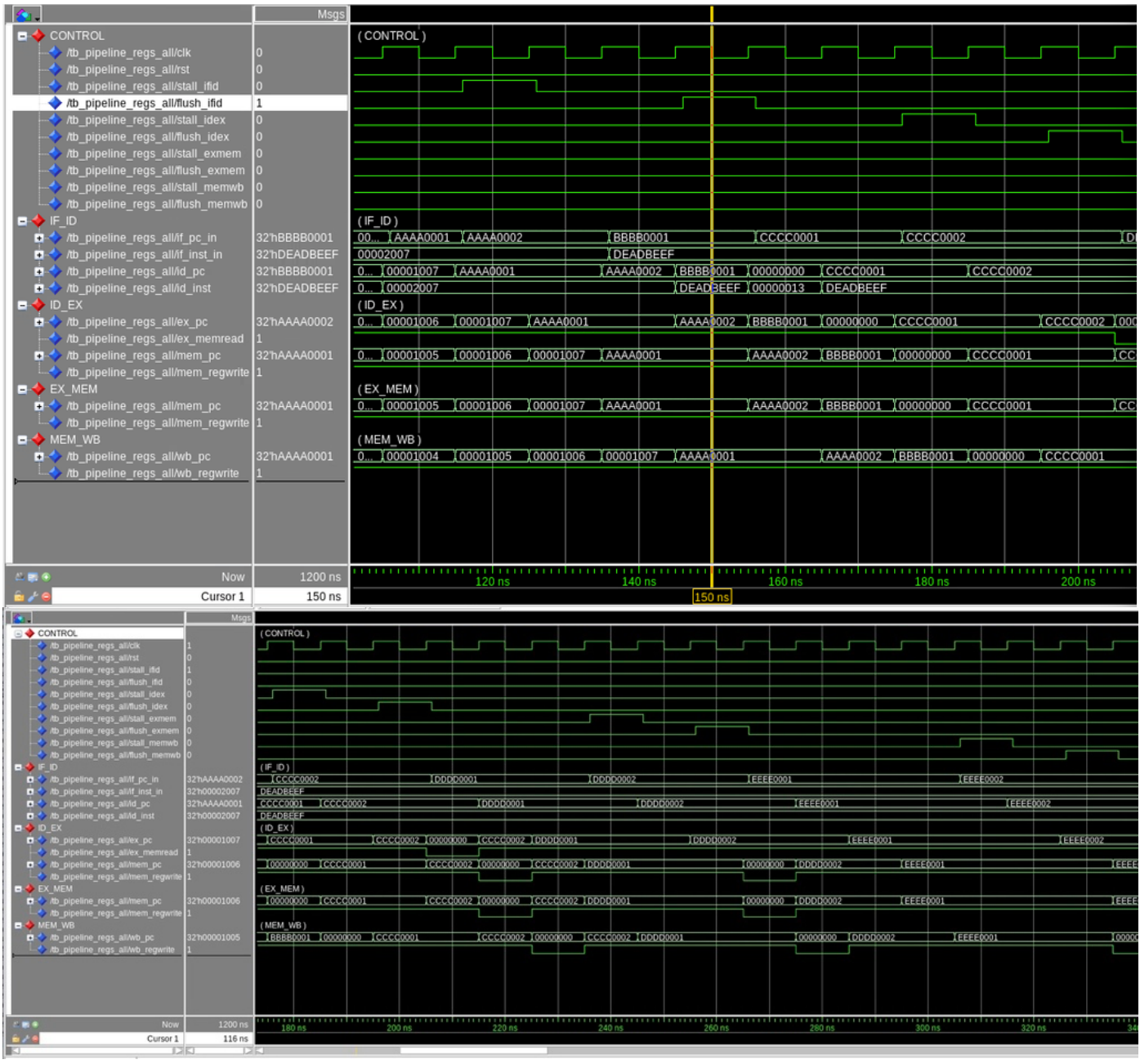
In summary, **our** critical path for the software-scheduled pipeline is an ID→EX→branch/fetch feedback path:

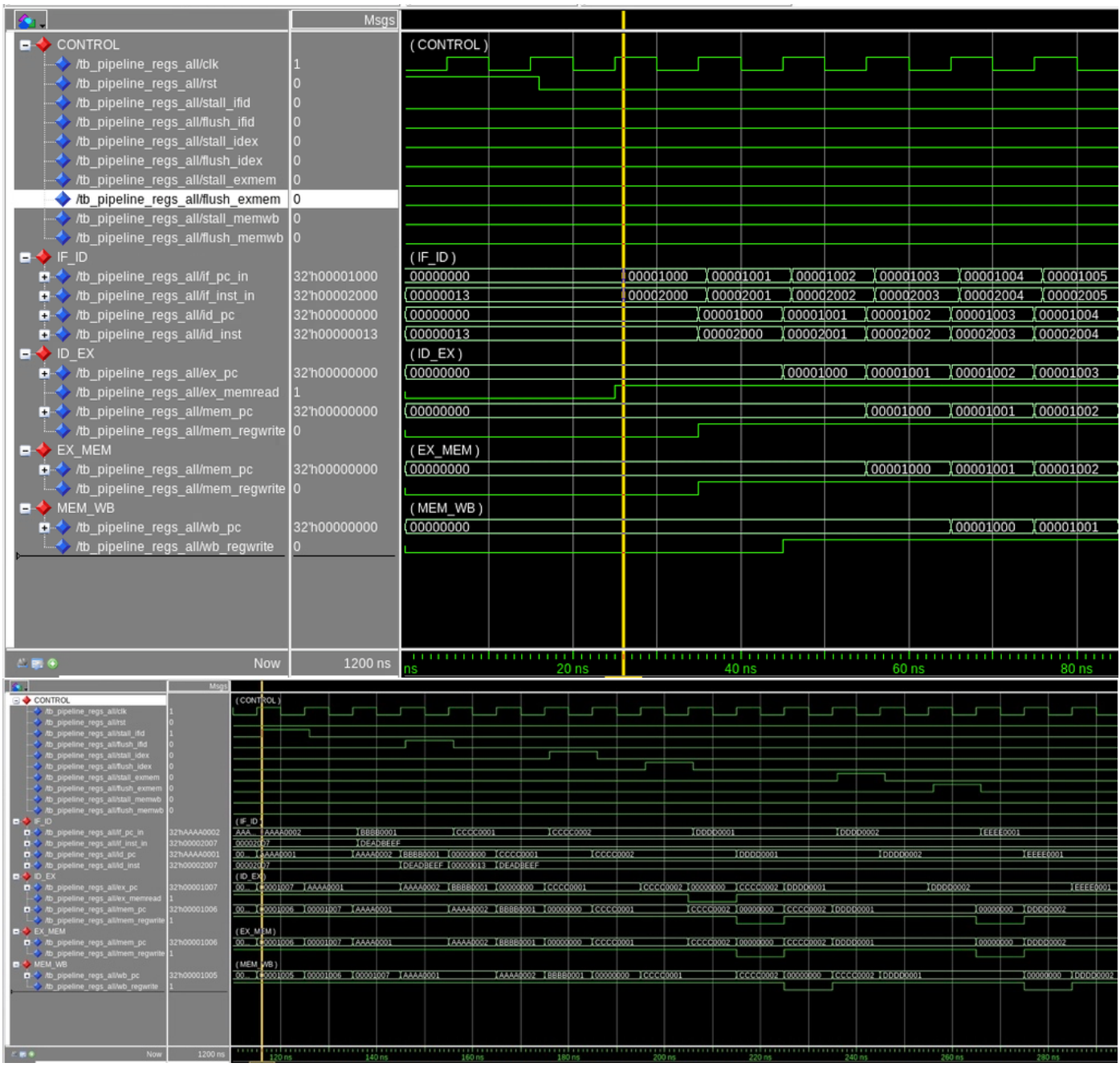
ID/EX.Funct7 → ALU control → ALU adder + result mux/compare → fetch_logic (NextPC) → PC register.

[2.a.ii] Draw a simple schematic showing how you could implement stalling and flushing operations given an ideal N-bit register.



[2.a.iii] Create a testbench that instantiates all four of the registers in a single design. Show that values that are stored in the initial IF/ID register are available as expected four cycles later, and that new values can be inserted into the pipeline every single cycle. Most importantly, this testbench should also test that each pipeline register can be individually stalled or flushed.


Msgs
CONTROL
/tb_pipeline_regs_all/clk
/tb_pipeline_regs_all/rst
/tb_pipeline_regs_all/stall_ifid
/tb_pipeline_regs_all/flush_ifid
/tb_pipeline_regs_all/stall_idex
/tb_pipeline_regs_all/flush_idex
/tb_pipeline_regs_all/stall_exmem
/tb_pipeline_regs_all/flush_exmem
/tb_pipeline_regs_all/stall_memwb
/tb_pipeline_regs_all/flush_memwb
IF_ID
/tb_pipeline_regs_all/if_pc_in
/tb_pipeline_regs_all/if_inst_in
/tb_pipeline_regs_all/id_pc
/tb_pipeline_regs_all/id_inst
ID_EX
/tb_pipeline_regs_all/ex_pc
/tb_pipeline_regs_all/ex_memread
/tb_pipeline_regs_all/mem_pc
/tb_pipeline_regs_all/mem_regwrite
EX_MEM
/tb_pipeline_regs_all/mem_pc
/tb_pipeline_regs_all/mem_regwrite
MEM_WB
/tb_pipeline_regs_all/wb_pc
/tb_pipeline_regs_all/wb_regwrite
32'hBBBB0001
32'hDEADBEEF
32'hBBBB0001
32'hDEADBEEF
32'hAAAA0002
32'hAAAA0001
32'hAAAA0001
(CONTROL)
(IF_ID)
(ID_EX)
(EX_MEM)
(MEM_WB)
Now
1200 ns
Cursor 1
150 ns
120 ns
140 ns
160 ns
180 ns
200 ns
Msgs
32'hAAAA0002
32'h00002007
32'h00001007
32'h00001006
32'h00001005
116 ns
180 ns
200 ns
220 ns
240 ns
260 ns
280 ns
300 ns
320 ns



[2.b.i] list which instructions produce values, and what signals (i.e., bus names) in the pipeline these correspond to.

R-type ALU: `add, sub, and, or, xor, slt, sll, srl, sra`	Yes	Regfile	ALU result	`s_ALUResult` → `s_ALUResult_MEM` → `s_ALUResult_WB` → `s_RegWrData`
I-type ALU: `addi, andi, ori, xori, slti, sltiu, slli, srli, srai`	Yes	Regfile	ALU result	`s_ALUResult` → `s_ALUResult_MEM` → `s_ALUResult_WB` → `s_RegWrData`
Loads: `lw, lb, lh, lbu, lhu`	Yes	Regfile	Load data	`s_DMemOut` → `s_LoadData` → `s_LoadData_WB` → `s_RegWrData`
Store: `sw`	Yes	Data memory	Store data (rs2 value)	`s_ReadData2` → `s_ReadData2_EX` → `s_ReadData2_MEM` → `s_DMemData`
LUI: `lui`	Yes	Regfile	Immediate value	`s_Imm` → `s_Imm_EX` → `s_Imm_MEM` → `s_Imm_WB` → `s_RegWrData`
AUIPC: `auipc`	Yes	Regfile	PC + Imm (design-dependent path)	**Imm path exists:** `s_Imm` → `s_Imm_EX` → `s_Imm_MEM` → `s_Imm_WB` → `s_RegWrData` **and/or ALU path:** `s_ALUResult` → `s_ALUResult_MEM` → `s_ALUResult_WB` → `s_RegWrData`
JAL / JALR: `jal, jalr`	Yes	Regfile	PC+4	`s_PC_WB` → (WB mux generates `s_PC_WB + 4`) → `s_RegWrData`
Branches: `beq, bne, blt, bge, bltu, bgeu`	No	—	—	No produced value (no reg write, no mem write)
HALT/WFI: `wfi` (or HALT)	No	—	—	No produced value

[2.b.ii] List which of these same instructions consume values, and what signals in the pipeline these correspond to.

R-type ALU: `add, sub, and, or, xor, slt, sll, srl, sra`	Yes	`rs1, rs2`	**ID read:** `s_ReadData1, s_ReadData2` → **ID/EX:** `s_ReadData1_EX, s_ReadData2_EX` → **ALU inputs:** `s_ALUInputA, s_ALUInputB`
I-type ALU: `addi, andi, ori, xori, slti, sltiu, slli, srli, srai`	Yes	`rs1`	`s_ReadData1` → `s_ReadData1_EX` → `s_ALUInputA` (B-side uses `s_Imm_EX` via `s_ALUInputB`)
Loads: `lw, lb, lh, lbu, lhu`	Yes	`rs1` (base)	`s_ReadData1` → `s_ReadData1_EX` → `s_ALUInputA` (address calc)
Branches: `beq, bne, blt, bge, bltu, bgeu`	Yes	`rs1, rs2` (compare)	`s_ReadData1/s_ReadData2` → `s_ReadData1_EX/s_ReadData2_EX` → `s_ALUInputA/s_ALUInputB` (compare result shows up as `s_Zero/s_Sign/s_Cout` used by `fetch_logic`)
JALR: `jalr`	Yes	`rs1` (target base)	`s_ReadData1` → `s_ReadData1_EX` (fed to `fetch_logic` as `i_RS1 => s_ReadData1_EX`)
JAL: `jal`	No (reg operands)	—	No `rs1/rs2` consumption
LUI: `lui`	No	—	No `rs1/rs2` consumption
AUIPC: `auipc`	No (reg operands)	**PC** (internal)	Consumes PC via pipeline: `s_PC_ID` → `s_PC_EX` → `s_ALUInputA` when `s_ALUSrcA_EX='1'`
Store: `sw`	**Yes (tricky one)**	`rs1` (address base) **and** `rs2` (store data)	**Address path:** `s_ReadData1` → `s_ReadData1_EX` → `s_ALUInputA` (addr calc). **Data path:** `s_ReadData2` → `s_ReadData2_EX` → `s_ReadData2_MEM` → `s_DMemData` (the actual write data into DMem)
HALT/WFI: `wfi`	No	—	No operand consumption

[2.b.iii] generalized list of potential data dependencies. From this generalized list, select those dependencies that can be forwarded (write down the corresponding pipeline stages that will be forwarding and receiving the data), and those dependencies that will require hazard stalls.

This global list summarizes the datapath values and control signals required in each pipeline stage. It has been updated to include the **additional datapath metadata (e.g., register addresses)** that must be stored to ensure correct operation with **hazard detection and data forwarding**.

IF Stage

Required datapath values

- Current PC
- Fetched instruction

Required control signals

- PC write enable (used externally for stalling IF)

Stored in IF/ID

- PC
- Instruction

ID Stage

Required datapath values

- Instruction fields
    - rs1_addr, rs2_addr, rd_addr
- Register file outputs:
    - ReadData1, ReadData2
- Immediate value
- funct3, funct7

Required control signals (generated in ID)

- RegWrite
- MemRead
- MemWrite
- MemtoReg
- ALUSrcA, ALUSrcB
- ALUOp
- Branch
- Jump
- Halt

Stored in ID/EX (UPDATED for hazard/forwarding)

Datapath

- PC
- ReadData1
- ReadData2
- Imm
- funct3, funct7
- rd
- **NEW: rs1_addr (to EX)**
- **NEW: rs2_addr (to EX)**

Control

- RegWrite
- MemRead
- MemWrite
- MemtoReg
- ALUSrcA, ALUSrcB
- ALUOp
- Branch
- Jump
- Halt

The addition of rs1_addr and rs2_addr in ID/EX enables generalized EX-stage forwarding and accurate load-use hazard detection by comparing against rd in later stages.

EX Stage

Required datapath values

- Operand values:
    - ReadData1_EX, ReadData2_EX
- Immediate
- Destination register:
    - rd_EX
- **Operand addresses for forwarding selection:**
    - **rs1_addr_EX, rs2_addr_EX**
- ALU inputs and ALU result
- Branch comparison flags

Required control signals

- RegWrite_EX
- MemRead_EX
- MemWrite_EX
- MemtoReg_EX
- Branch_EX
- Jump_EX
- Halt_EX

Stored in EX/MEM (optionally updated)

Datapath

- PC
- ALUResult
- ReadData2 (store data)
- Imm
- funct3
- rd
- **Optional NEW: rs2_addr_MEM** (useful for store-data forwarding decisions)

Control

- RegWrite
- MemRead
- MemWrite
- MemtoReg
- Halt

MEM Stage

Required datapath values

- ALUResult_MEM (effective address)
- ReadData2_MEM (store data)
- DMemOut → LoadData
- rd_MEM
- **Optional: rs2_addr_MEM** (if used for store-data forwarding)

Required control signals

- MemRead_MEM
- MemWrite_MEM
- RegWrite_MEM
- MemtoReg_MEM
- Halt_MEM

Stored in MEM/WB

Datapath

- PC
- ALUResult
- LoadData
- Imm
- rd

Control

- RegWrite
- MemtoReg
- Halt

WB Stage

Required datapath values

- Writeback candidates:
    - ALUResult_WB
    - LoadData_WB
    - Imm_WB
    - PC_WB (+4)
- rd_WB
- Final writeback data (WB mux output)

Required control signals

- RegWrite_WB
- MemtoReg_WB
- Halt_WB

[2.b.iv] global list of the datapath values and control signals that are required during each pipeline stage

IF stage

Datapath values

- PC_IF
- Inst_IF

Controls

- PCWrite (from hazard/pipeline control)

Stored in

- **IF/ID**: PC, Inst

ID stage

Datapath values

- PC_ID
- Inst_ID
- ReadData1_ID, ReadData2_ID
- Imm_ID
- Funct3_ID, Funct7_ID
- Rd_ID
- **Rs1Addr_ID**, **Rs2Addr_ID** *(extracted from instruction)*

Controls

- RegWrite_ID
- MemRead_ID
- MemWrite_ID
- MemtoReg_ID
- ALUSrcA_ID, ALUSrcB_ID
- ALUOp_ID
- Branch_ID
- Jump_ID
- Halt_ID

Stored in

- **ID/EX**: everything needed for EX
    - **Rs1Addr_EX (added)**
    - **Rs2Addr_EX (added)**

EX stage

Datapath values

- PC_EX
- ReadData1_EX, ReadData2_EX
- Imm_EX
- Funct3_EX, Funct7_EX
- Rd_EX
- **Rs1Addr_EX**, **Rs2Addr_EX** *(for compare logic)*
- ALUResult_EX

Controls

- RegWrite_EX
- MemRead_EX
- MemWrite_EX
- MemtoReg_EX
- ALUSrcA_EX, ALUSrcB_EX
- ALUOp_EX
- Branch_EX
- Jump_EX
- Halt_EX

Forwarding-related runtime signals

- Compare consumers vs producers:
    - Consumers: **Rs1Addr_EX, Rs2Addr_EX**
    - Producers: Rd_MEM, Rd_WB
    - Qualifiers: RegWrite_MEM, RegWrite_WB

Stored in

- **EX/MEM**:

- **Rs2Addr_MEM (added)**

  This helps cleanly support **store-data forwarding** checks when needed.
- PC_MEM, ALUResult_MEM, ReadData2_MEM, Imm_MEM, Funct3_MEM, Rd_MEM
- RegWrite_MEM, MemRead_MEM, MemWrite_MEM, MemtoReg_MEM, Halt_MEM

MEM stage

Datapath values

- PC_MEM
- ALUResult_MEM
- ReadData2_MEM *(store data)*
- Imm_MEM
- Funct3_MEM
- Rd_MEM
- **Rs2Addr_MEM** *(if you keep your added path)*
- LoadData_MEM (from load extender)

Controls

- MemRead_MEM
- MemWrite_MEM
- RegWrite_MEM
- MemtoReg_MEM
- Halt_MEM

Stored in

- **MEM/WB**:
  - PC_WB, ALUResult_WB, LoadData_WB, Imm_WB, Rd_WB
  - RegWrite_WB, MemtoReg_WB, Halt_WB

WB stage

Datapath values

- ALUResult_WB
- LoadData_WB
- Imm_WB

- PC_WB *(for PC+4 writeback path)*
- Rd_WB

Controls

- RegWrite_WB
- MemtoRe

[2.c.i] list all instructions that may result in a non-sequential PC update and in which pipeline stage that update occurs.
Non-sequential PC instructions: beq, bne, blt, bge, bltu, bgeu, jal, jalr
PC redirect / control decision stage: EX stage

[2.c.ii] For these instructions, list which stages need to be stalled and which stages need to be squashed/flushed relative to the stage each of these instructions is in.
For beq, bne, blt, bge, bltu, bgeu resolved in EX:
Stall: none
Flush on taken: IF and ID stages (instructions younger than the branch)

For jal and jalr resolved in EX:
Stall: none
Flush: IF and ID stages (instructions fetched after the jump)

[2.d] implement the hardware-scheduled pipeline using only structural VHDL. As with the previous processors that you have implemented, start with a high-level schematic drawing of the interconnection between components.

[2.e – i, ii, and iii] In your writeup, show the QuestaSim output for each of the following tests, and provide a discussion of result correctness. It may be helpful to also annotate the waveforms directly.

[2.e.i] Create a spreadsheet to track these cases and justify the coverage of your testing approach. Include this spreadsheet in your report as a table.

```
F1   EX/MEM → EX (rs1)      ForwardA      fwd_exmem_rs1.s       ForwardA = "10"

F2   EX/MEM → EX (rs2)      ForwardB      fwd_exmem_rs2.s       ForwardB = "10"

F3   MEM/WB → EX (rs1)      ForwardA      fwd_memwb_rs1.s       ForwardA = "01"

F4   MEM/WB → EX (rs2)      ForwardB      fwd_memwb_rs2.s       ForwardB = "01"

S1   EX/MEM → Store         StoreFwd_EX   store_fwd_exmem.s     StoreFwd_EX = "10"

S2   MEM/WB → Store         StoreFwd_EX   store_fwd_memwb.s     StoreFwd_EX = "01"

H1   Load-use (lw → add)    HDU + pipe_control  hazard_load_use_add.s
     Expected control: PCWrite = 0, IFID_Write = 0, IDEX_Flush = 1

H2   Load-use (lw → sw)     HDU + pipe_control  hazard_load_use_sw.s
     Expected control: same as H1
     (PCWrite = 0, IFID_Write = 0, IDEX_Flush = 1)
```

[2.e.ii] Create a spreadsheet to track these cases and justify the coverage of your testing approach. Include this spreadsheet in your report as a table.

Control Hazard Test Coverage Table

Case ID	Control Instruction(s)	Scenario	What This Test Validates	Expected Pipeline Response	Key Signals to Observe	Test File	Pass Criteria
CH-S1	BEQ	Not-taken → Taken sequence	Correct branch decision timing and correct squash of wrong-path instructions	**Flush IF/ID and ID/EX only when branch is taken**	`s_Branch_EX`, `s_IFID_Flush`, `s_IDEX_Flush`, `s_PCWrite`, `s_NextInstAddr`	`ch_beq.s`	Taken branch removes fall-through effects; final stored/register value matches expectation
CH-S2	BNE	Not-taken → Taken	Same branch mechanism with inverted condition	Same as above	Same as above	`ch_bne.s`	BNE behavior matches RARS; flush only on taken
CH-S3	BLT (signed)	Positive compare not-taken + negative vs positive taken	Signed compare path correctness (via sign flag)	Flush on taken	`s_ALUSign`, `s_Branch_EX`, flush signals	`ch_blt.s`	Wrong-path instruction after taken BLT is squashed; final value correct
CH-S4	BGE (signed)	Mixed signed cases	Signed ≥ decision correctness	Flush on taken	`s_ALUSign`, `s_Branch_EX`, flush signals	`ch_bge.s`	Correct taken/not-taken behavior across signed values
CH-S5	BLTU (unsigned)	Unsigned less-than cases	Unsigned compare correctness (borrow/carry logic)	Flush on taken	`s_Cout`, `s_Branch_EX`, flush signals	`ch_bltu.s`	Cout-based decision matches expected results
CH-S6	BGEU (unsigned)	Unsigned ≥ cases	Unsigned ≥ path correctness	Flush on taken	`s_Cout`, `s_Branch_EX`, flush signals	`ch_bgeu.s`	Correct unsigned branch behavior
CH-S7	JAL	Unconditional jump with link	Jump priority and correct PC+4 writeback	**Jump causes flush of younger wrong-path instructions**	`s_Jump_EX`, `s_IFID_Flush`, `s_IDEX_Flush`, `s_PC_WB`, `s_RegWrite_WB`	`ch_jal.s`	Fall-through instructions do not commit; rd receives PC+4
CH-S8	JALR	Register-indirect jump with link	Correct target computation (rs1+imm, LSB cleared) + jump flush	Same as JAL	`s_Jump_EX`, `s_ReadData1_EX`, `s_Imm_EX`, `s_NextInstAddr`, flush signals	`ch_jalr.s`	Correct jump target; fall-through squashed; rd receives PC+4
CH-C1	BEQ + BNE	Back-to-back conditional branches	Stable flush behavior across consecutive branch events	Each taken branch triggers correct flush	flush signals across multiple cycles	`ch_combo_beq_bne.s`	No wrong-path state changes across consecutive branches
CH-C2	BLT + BGE	Signed branch combinations	Signed compare interactions in a realistic pipeline sequence	Correct per-branch flush	`s_ALUSign` + flush signals	`ch_combo_blt_bge.s`	Signed branch outcomes remain consistent under interaction
CH-C3	BLTU + BGEU	Unsigned combinations	Carry/borrow-based branch stability under interaction	Correct per-branch flush	`s_Cout` + flush signals	`ch_combo_bltu_bgeu.s`	Unsigned branches behave correctly when combined
CH-C4	JAL + BEQ	Jump + conditional mix	Jump priority over conditional control paths	Jump-driven flush dominates when applicable	`s_Jump_EX`, `s_Branch_EX`, flush signals	`ch_combo_jal_beq.s`	Jump behavior remains correct even with nearby conditional branches
CH-C5	JALR + Branch	Indirect jump + conditional	JALR target correctness + flush correctness under mixed control	Jump-driven flush dominates	`s_ReadData1_EX`, `s_Imm_EX`, `s_Jump_EX`, flush signals	`ch_combo_jalr_branch.s`	No fall-through commits; correct link + target

[2.f] report the maximum frequency your hardware-scheduled pipelined processor can run at and determine what your critical path is (specify each module/entity/component that this path goes through).

From the Quartus timing report, the maximum clock frequency **our** hardware-scheduled pipelined processor can run at is **39.33 MHz** (corresponding to a data-arrival time of 28.40 ns and a negative slack of -5.43 ns against the 20 ns clock constraint).

The critical path starts at the **EX/MEM pipeline register** bit fs_ex_mem_reg:U_EX_MEM|s_ALUResult_reg[6], then passes through the **data memory** blocks DMem|ram, the **load extender** U_LOADEXT (Mux36), the **write-back / operand multiplexers** Mux28 and Mux60 that form s_ALUInputA, then through the **ALU adder** U_ALU|U_ADDER and its result-selection / compare logic (U_ALU|Mux43 and U_ALU|Equal4), into the **branch/CHU unit** U_CHU|o_BranchTaken, then the **fetch logic** U_FETCH|o_NextPC, and finally into the **PC register** pc_reg:U_PC|s_PC[8].

In short, **our** critical path is an EX→MEM→WB→branch→fetch feedback path:

EX/MEM s_ALUResult → DMem → load_extender → ALU input mux → ALU adder & branch compare → CHU branch-taken logic → fetch_logic o_NextPC → pc_reg (PC[8]).