Pipelined CPU Design

Now, we will optimize a single cycle CPU using pipelining. Pipelining is a powerful logic design method to reduce the clock time and improve the throughput, even though it increases the latency of an individual task and adds additional logic. In a pipelined CPU, multiple instructions are overlapped in execution. This is a good example of parallelism, which is one of the great ideas in computer architecture. To obtain a pipelined CPU, we will take the following steps.

Step1: Pipeline Registers

Pipelining starts from adding pipelining registers by dividing a large combinational logic. We have already chopped a single cycle CPU into five stages, and thus, will add pipeline registers between two stages. We kindly added pipeline registers for the datapath. Note that the write address for the register file should be passed to the write back stage through the pipeline registers so that the instruction writes the result to the correct register. However, we miss the pipeline registers for the control signals.

Q1. Add the pipeline registers for the control signals and connect them to the datapath.

See page 5.

Step2: Performance Analysis

A great advantage of pipelining is the performance improvement with a shorter clock time. We will use the same timing parameters as those in the previous discussion.

Element	Register clk-to-q	Register Setup	MUX	ALU	Mem Read	Mem Write	RegFile Read	RegFile Setup
Parameter	$t_{clk\text{-}to\text{-}q}$	t_{setup}	t_{mux}	t_{ALU}	$t_{MEMread}$	$t_{MEMwrite}$	t_{RFread}	$T_{RFsetup}$
Delay(ps)	30	20	25	200	250	200	150	20

Q1. What was the clock time and frequency of a single cycle CPU?

$t_{clk,single} >= t_{PC,\ clk\text{-}to\text{-}q} + t_{IMEMread} + t_{RFread} + t_{ALU} + t_{DMEMread} + t_{mux} + t_{RFsetup}$

$= 30 + 250 + 150 + 200 + 250 + 25 + 20 = 925$ ps

$f_{clk,single} = 1/t_{clk,pipe} <= 1/\ (925\text{ ps}) = 1.08$ GHz

Q2. What is the clock time and frequency of a pipelined CPU?

$$t_{clkpipe} >= \max \begin{pmatrix} t_{clk\text{-}to\text{-}q} + t_{MEMread} + t_{setup}\ \text{(Fetch)} \\ t_{clk\text{-}to\text{-}q} + t_{RFread} + t_{setup}\ \text{(Decode)} \\ t_{clk\text{-}to\text{-}q} + t_{ALU} + t_{mux} + t_{setup}\ \text{(Execute)} \\ t_{clk\text{-}to\text{-}q} + t_{MEMread} + t_{setup}\ \text{(Memory)} \\ t_{clk\text{-}to\text{-}q} + t_{mux} + t_{RFsetup}\ \text{(Writeback)} \end{pmatrix} = 300\text{ps}$$

$f_{clk,pipe} = 1/t_{clk,pipe} <= 1/\ (300\text{ ps}) = 3.33$ GHz

Q3. What is the speed-up? Why is it less than five?

Speed-up = $t_{clk,pipe}\ /\ t_{clk,single} = f_{clk,pipe}\ /\ f_{clk,single} = 3.08$.

This is because pipeline stages are not balanced evenly and there is overhead from pipeline registers ($t_{clk\text{-}to\text{-}q}$, t_{setup}). Moreover, this does not include the delays from the additional logic for hazard resolution.

Step3: Pipeline Hazard

The performance improvement comes at a cost. Pipelining introduces pipeline hazards we have to overcome.

Structural Hazard

Structural hazards occur when more than one instruction use the same resource at the same time.

- **Register File:** One instruction reads from the register file while another writes to it. We can solve this by having separate read and write ports and writing to the register file at the falling edge of the clock.
- **Memory:** The memory is accessed not only for the instruction but also for the data. Separate caches for instructions and data solve this hazard.

Data Hazard and Forwarding

Data hazards occur due to data dependencies among instructions. Forwarding can solve many data hazards.

Q1. Spot the data dependencies in the code below and figure out how forwarding can resolve data hazards.

Clock Cycles	C0	C1	C2	C3	C4	C5	C6
addi $t0, $s0, -1	IF	ID	EX	MEM	WB		
and $s2, $t0, $a0		IF	ID	EX	MEM	WB	
sw $s0, 100($t0)			IF	ID	EX	MEM	WB

The and and sw instructions need the values of $t0 for their execution stages. Fortunately, these values are ready before their execution stages. Thus, we can forward it from the pipeline registers before writing it to the register file.

Q2. Provide the inputs to the forwarding unit.

`rsE, rtE, WriteAddrM, WriteAddrW, RegWrM, RegWrW`

Q3. Write the condition for each forwarding signal.

`FwdAM` (Forwarding `ALUOutM` to `A`)

`= RegWrM & (rsE != 0) & (rsE == WriteAddrM)`

`FwdBM` (Forwarding `ALUTOutM` to `B`)

`= RegWrM & (rtE != 0) & (rtE == WriteAddrM)`

`FwdAW` (Forwarding `WriteDataW` to `A`)

`= RegWrW & (rsE != 0) & (rsE == WriteAddrW)`

`FwdBW` (Forwarding `WriteDataW` to `B`)

`= RegWrW & (rtE != 0) & (rtE == WriteAddrW)`

Q4. Implement the forwarding logic.

See page 6.

Data Hazard and Stalls

Forwarding cannot solve all data hazards. We need to stall the pipeline in some cases.

Q1. Spot the data dependencies in the code below and figure out why forwarding cannot resolve this hazard.

Clock Cycles	C0	C1	C2	C3	C4	C5
lw $t0, 20($s0)	IF	ID	EX	MEM	WB	
add $t1, $t0, $t0		IF	ID	EX	MEM	WB

The add instruction needs the value of $t0 in the beginning of C3, but it is ready at the end of C3.

Q2. Now we stall the pipeline one cycle and insert nop after the lw instruction. Figure out how this can resolve the hazard.

Clock Cycles	C0	C1	C2	C3	C4	C5	C6
lw $t0, 20($s0)	IF	ID	EX	MEM	WB		
nop		IF	ID	Bub	Bub	Bub	
add $t1, $t0, $t0			IF	ID	EX	MEM	WB

By stalling one cycle, the add instruction can start its execution stage after the $t0 value is ready.

Q3. Provide the inputs to the hazard detection logic.

First, we assume that `MemToReg` is 1 only if the instruction is `lw`. (No don't care terms for `MemToReg`) Then, provide `rs, rt, rtE, MemToRegE` to the logic.

Q4. Write the condition of the stall signal.

```
Stall = MemToRegE & ((rs == rtE) | (rt == rtE))
```

Q5. Implement the stalling logic.

See page 7. Note that we add a nop by zeroing control signals

Control Hazard and Prediction

Control hazards occur due to jumps and branches. We may solve them by stalling the pipeline. However, it is painful since the branch condition is calculated after the execution stage and the pipeline is stalled for three cycles. Instead, we predict branches are not taken and flush the pipeline if they are actually taken.

Q1. Assume that the branch is actually taken for the beq instruction. Figure out how the pipeline flush can resolve the control hazard.

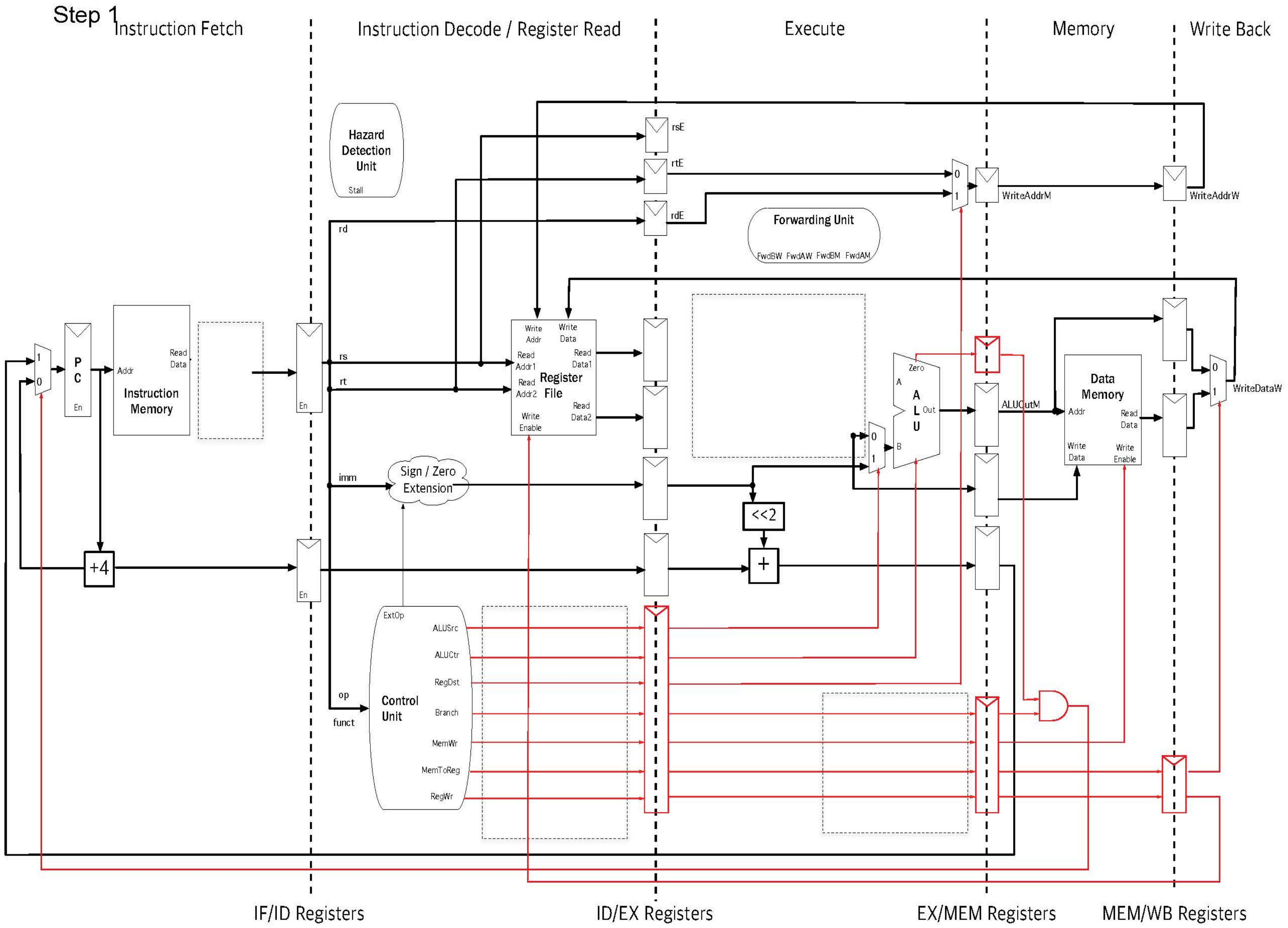
PC	Clock Cycles	C1	C2	C3	C4	C5	C6	C7	C8	C9
0x2000	beq $t0, $s0, 0x10	IF	ID	EX	MEM	WB				
0x2004	add $s2, $t0, $a0		IF	ID	EX	Bub	Bub			
0x2008	sw $s0, 100($t0)			IF flush	ID	Bub	Bub	Bub		
0x2010	xor $s2, $0, $0				IF	Bub	Bub	Bub	Bub	
0x2040	add, $s2, $t0, $a1					IF	ID	EX	MEM	WB

When a taken branch is detected, the incorrect stream of instructions is nullified by inserting bubbles to pipeline registers and fetch the correct instruction.

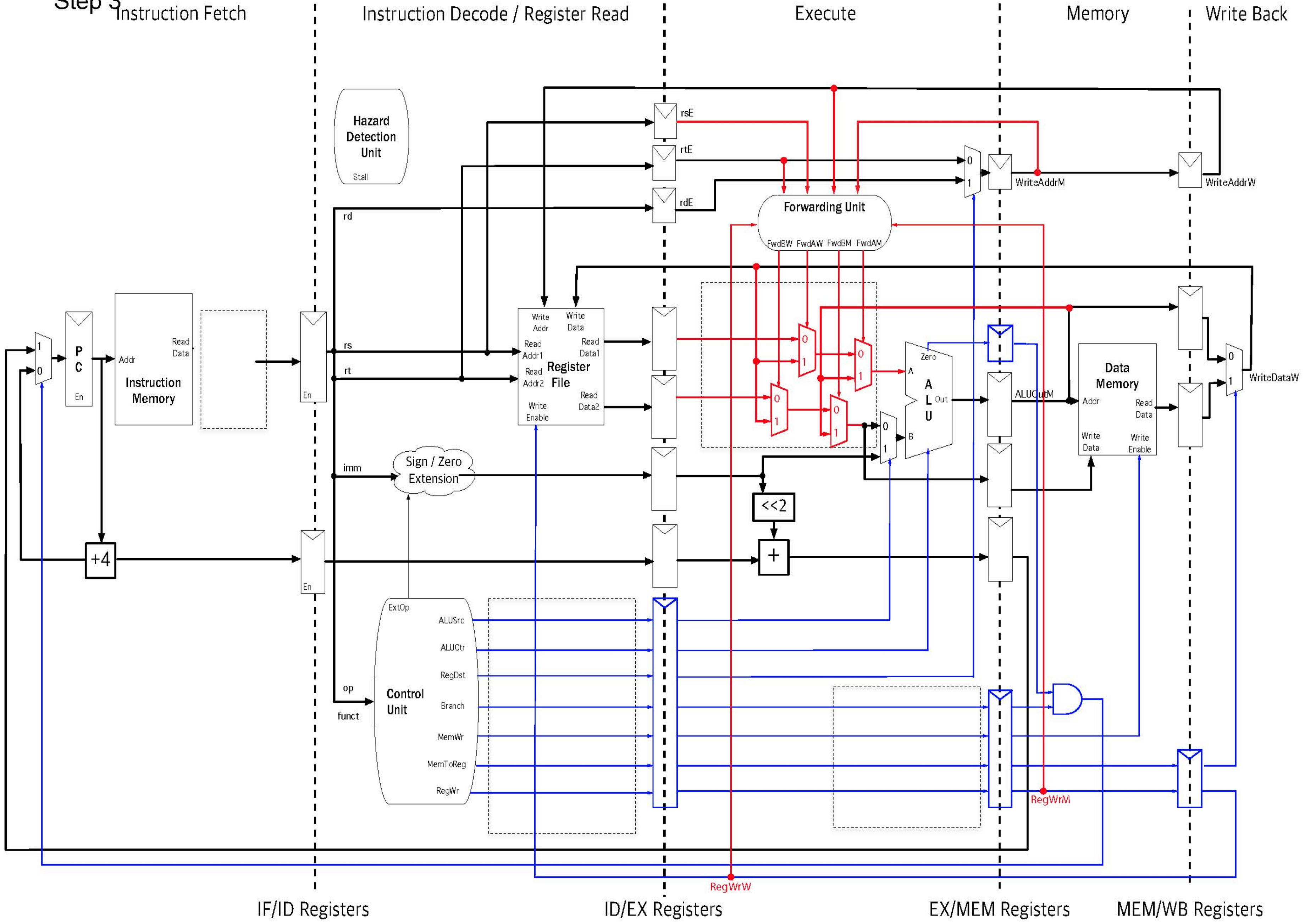
Q2. Implement the pipeline flush.

See page 8. We use the PCSrc signal to flush the pipeline.

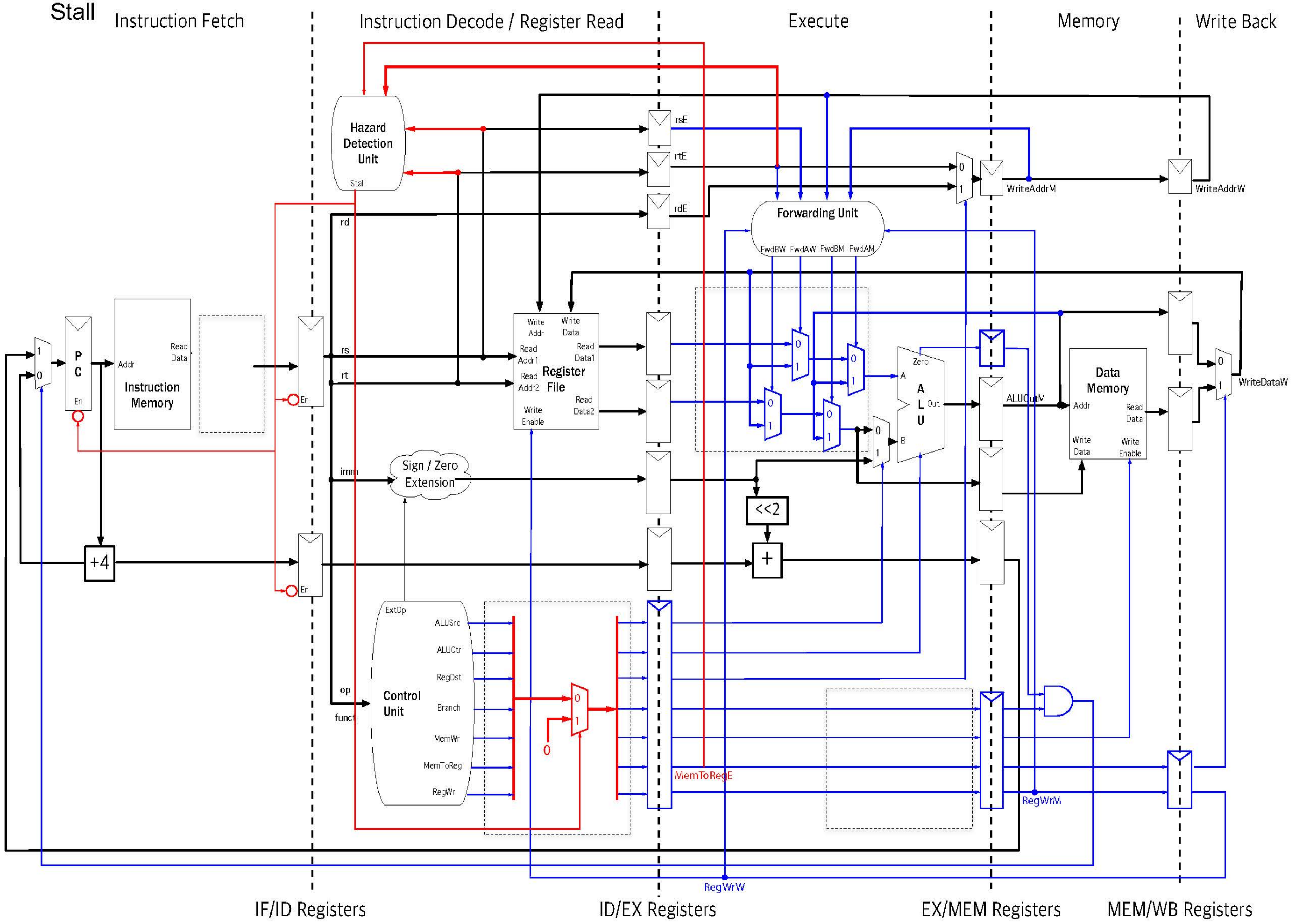
Step 1


Instruction Fetch
Instruction Decode / Register Read
Execute
Memory
Write Back
Hazard Detection Unit
Stall
rsE
rtE
rdE
WriteAddrM
WriteAddrW
Forwarding Unit
FwdBW FwdAW FwdBM FwdAM
rd
rs
rt
imm
op
funct
P C
En
Addr
Read Data
Instruction Memory
Write Addr
Write Data
Read Addr1
Read Addr2
Register File
Read Data1
Read Data2
Write Enable
Sign / Zero Extension
Zero
A
ALU
Out
B
ALUOutM
<<2
+
+4
Data Memory
Addr
Read Data
Write Data
Write Enable
WriteDataW
ExtOp
ALUSrc
ALUCtr
RegDst
Branch
MemWr
MemToReg
RegWr
Control Unit
IF/ID Registers
ID/EX Registers
EX/MEM Registers
MEM/WB Registers

Step 3

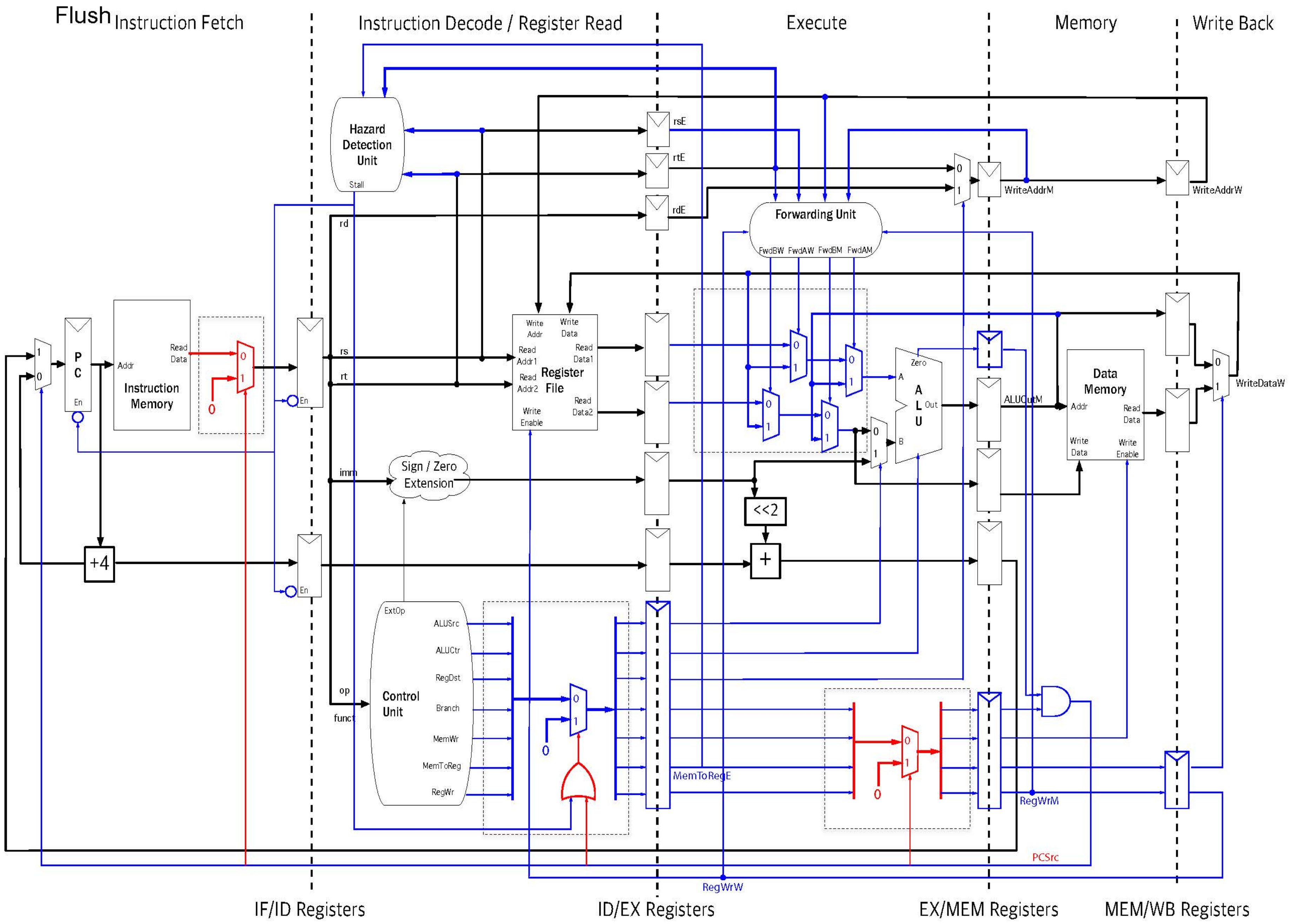

Instruction Fetch
Instruction Decode / Register Read
Execute
Memory
Write Back
Hazard Detection Unit
Stall
rsE
rtE
rdE
WriteAddrM
WriteAddrW
Forwarding Unit
FwdBW FwdAW FwdBM FwdAM
rd
rs
rt
imm
op
funct
P C
En
Addr
Read Data
Instruction Memory
Write Addr
Write Data
Read Addr1
Read Addr2
Register File
Read Data1
Read Data2
Write Enable
Sign / Zero Extension
+4
<<2
+
Zero
A
B
A L U
Out
ALUOutM
Data Memory
Addr
Read Data
Write Data
Write Enable
WriteDataW
ExtOp
ALUSrc
ALUCtr
RegDst
Branch
MemWr
MemToReg
RegWr
Control Unit
RegWrM
RegWrW
IF/ID Registers
ID/EX Registers
EX/MEM Registers
MEM/WB Registers

Stall



Instruction Fetch
Instruction Decode / Register Read
Execute
Memory
Write Back
Hazard Detection Unit
Stall
rsE
rtE
rdE
rd
rs
rt
imm
op
funct
Forwarding Unit
FwdBW
FwdAW
FwdBM
FwdAM
WriteAddrM
WriteAddrW
PC
En
Addr
Read Data
Instruction Memory
Write Addr
Write Data
Read Addr1
Read Addr2
Read Data1
Read Data2
Register File
Write Enable
Sign / Zero Extension
ExtOp
Control Unit
ALUSrc
ALUCtr
RegDst
Branch
MemWr
MemToReg
RegWr
MemToRegE
RegWrM
RegWrW
+4
<<2
+
Zero
A
B
ALU
Out
ALUOutM
Data Memory
Addr
Read Data
Write Data
Write Enable
WriteDataW
0
1
IF/ID Registers
ID/EX Registers
EX/MEM Registers
MEM/WB Registers

Flush


Instruction Fetch
Instruction Decode / Register Read
Execute
Memory
Write Back
Hazard Detection Unit
Stall
rsE
rtE
rdE
rd
rs
rt
imm
op
funct
Forwarding Unit
FwdBW FwdAW FwdBM FwdAM
WriteAddrM
WriteAddrW
PC
En
Addr
Read Data
Instruction Memory
Register File
Write Addr
Write Data
Read Addr1
Read Data1
Read Addr2
Read Data2
Write Enable
Sign / Zero Extension
<<2
+
+4
Zero
A
B
ALU
Out
ALUOutM
Data Memory
Write Data
Write Enable
WriteDataW
ExtOp
ALUSrc
ALUCtr
RegDst
Branch
MemWr
MemToReg
RegWr
Control Unit
MemToRegE
RegWrM
RegWrW
PCSrc
IF/ID Registers
ID/EX Registers
EX/MEM Registers
MEM/WB Registers