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. What registers need to be added to achieve pipelining? Add the pipeline registers for the control signals and connect them to the datapath.

Purple objects and lines in the diagram

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

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

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

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

Q3. Remember in the last discussion we calculated the clock time and frequency of a single cycle CPU. Given that minimum clock time is 925ps and freqeuency is 1.08 Ghz of the single cycle CPU, 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 If we want to design a unit that determines forwarding decisions, what inputs does this unit need? Provide the inputs to this forwarding unit, i.e. what information is necessary to make forwarding decisions.

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

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.

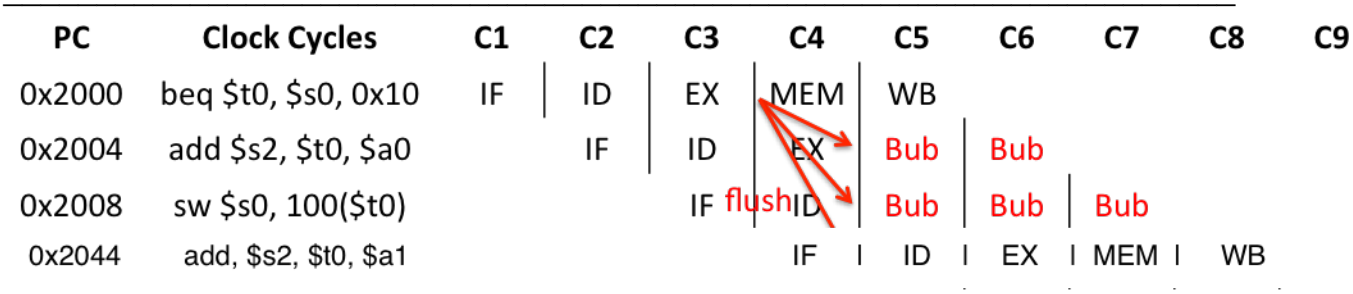
Q3. Similar to the forwarding logic unit in the previous step, provide the inputs to the hazard detection logic unit. How do you check whether there will be a data hazard that necessitates a nop instruction. (Assuming memToReg signal is 1 only for lw instruction.).

First, this hazard only present with lw instruction and we assume that `MemToReg` is 1 only if the instruction is `lw`. (No don't care terms for `MemToReg`), we can use MemToReg signal to determine if nop is necessary. Then, as long as `rs == rsE and rt == rtE, we will need nop`. Here rsE and rtE are rs and rt of the previous instruction. (e.g. checking if $t0 presents in two consecutive lines.)

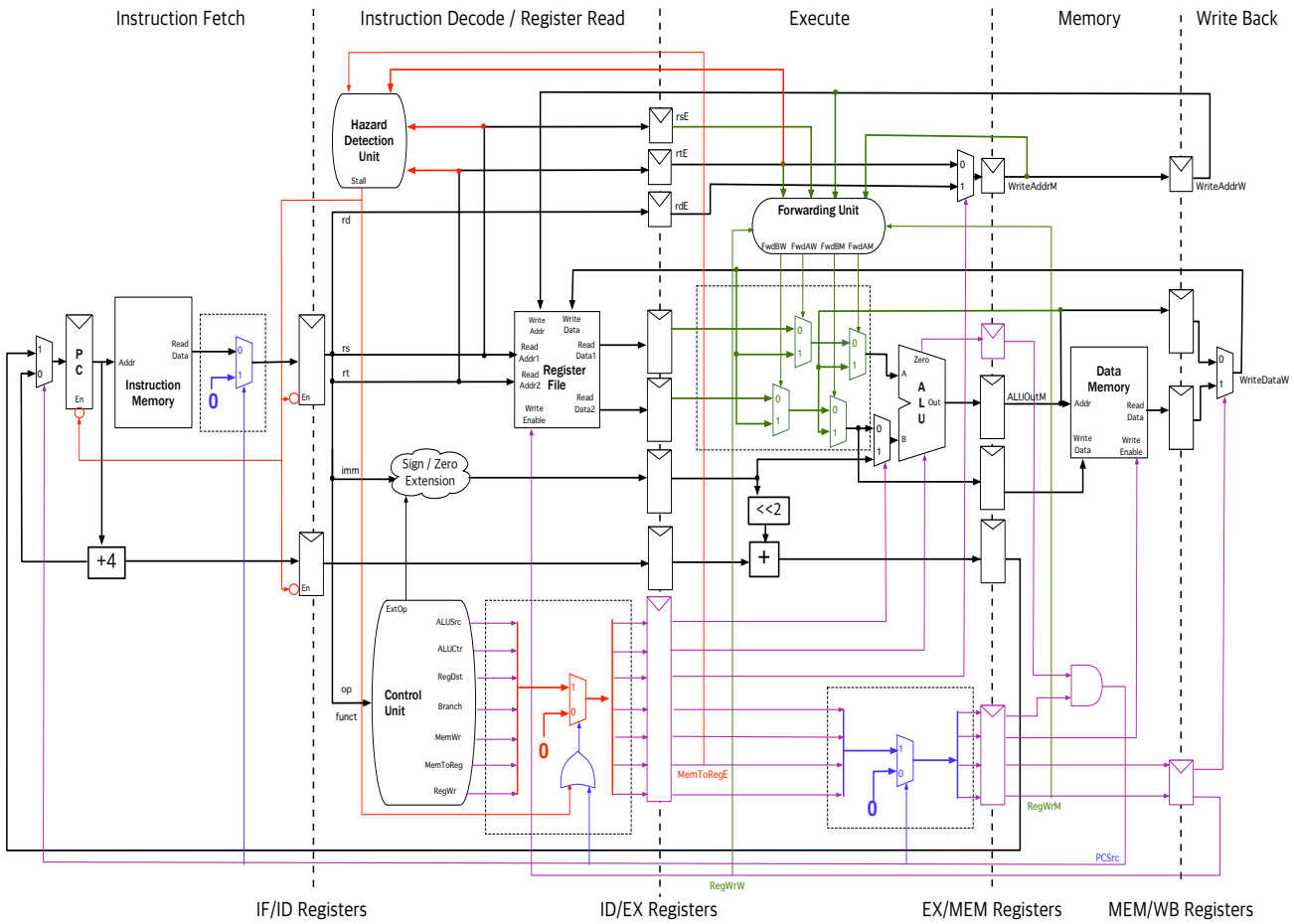
Control Hazard and Prediction

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.



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


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