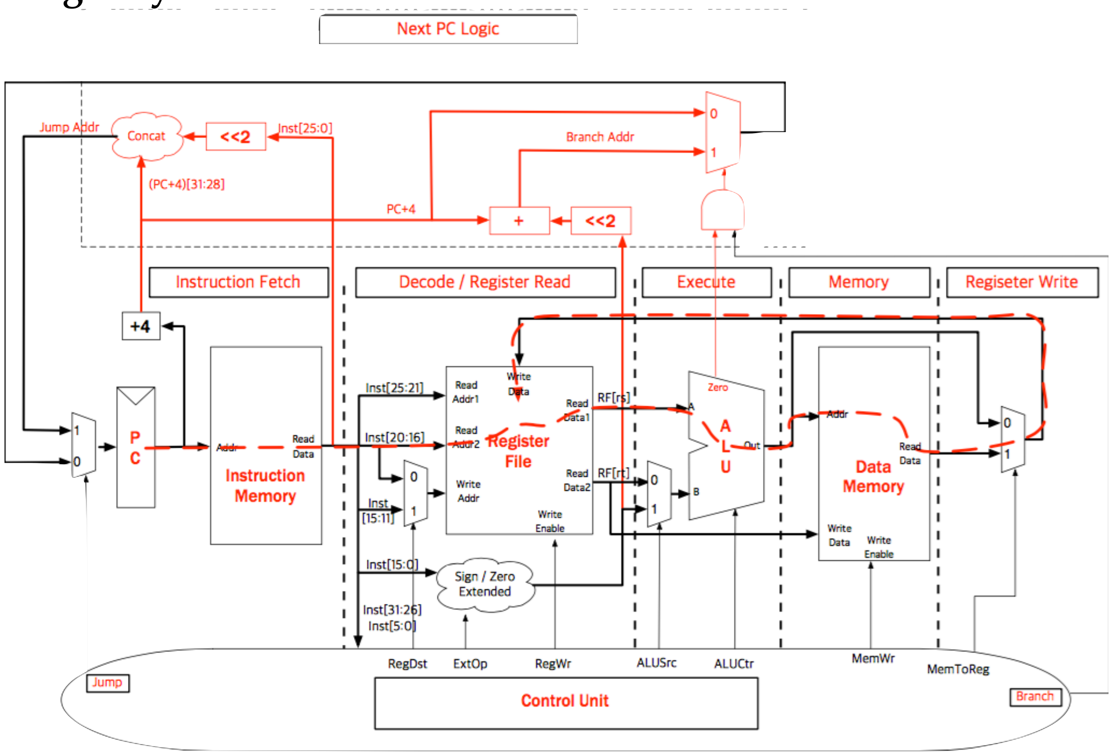
CS61C Spring 2017
Discussion 6 – Control and Pipelining

Single Cycle CPU Review



Single Cycle CPU Control Logic

Fill out the values for the control signals from the previous CPU diagram.

Instrs.	Control Signals								
	Jump	Branch	RegDst	ExtOp	ALUSrc	ALUCtr	MemWr	MemtoReg	RegWr
`add`	0	0	1	X	0	0010	0	0	1
`ori`	0	0	0	0	1	0001	0	0	1
`lw`	0	0	0	1	1	0010	0	1	1
`sw`	0	0	X	1	1	0010	1	X	0
`beq`	0	1	X	1	0	0110	0	X	0
`j`	1	X	X	X	X	XXXX	0	X	0

X: don't care value(either 0 or 1 is ok)

This table shows the ALUCtr values for each operation of the ALU:

Operation	AND	OR	ADD	SUB	SLT	NOR
ALUCtr	0000	0001	0010	0110	0111	1100

1. Why is a single cycle CPU inefficient?

   -Not all instructions exercise the critical path.

   -It is not parallelized. Each component can be active concurrently.

2. How can you improve its performance? What is the purpose of pipelining?

Pipelining: Put pipeline registers between two datapath stages. [illegible] reduce the clock time

CS61C Spring 2017
Discussion 6 – Control and Pipelining

Pipelined CPU Design

Now we optimize a single cycle CPU using pipelining, in which multiple instructions are executed in different stages simultaneously– in ***parallel.*** This can help improve the ***throughput*** (number of instructions completed over time), even though it increases the ***latency*** (total time for one instruction) of an individual task and adds additional logic. To obtain a pipelined CPU, we will take the following steps.

Step 1: Pipeline Registers

Add pipelining registers to divide the combinational logic into smaller blocks. We will put registers between the five stages we have named in a single cycle CPU.

Step 2: 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 is the clock time and frequency of a pipelined CPU?

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

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

Q2. Recall that the maximum frequency of a single-cycle CPU 1/925ps = 1.08 GHz. 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.

Step 3: Pipeline Hazards

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.

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

The REG step for instructions 2 and 3 depend on data in the registers only available after the WB step of instruction 1. We can forward the ALU output of the first instruction to the EX stages of future instructions

Q2. In general, under what conditions will an EX stage need to take in forwarded inputs from previous instructions? Where should those inputs come from in regards to the current cycle? Assume you have the signals ALUout(n), rt(n), rs(n), regWrite(n), and regDst(n), where n is 0 for the signal of the current instruction being executed by the EX stage, -1 for the previous, etc.

Forward ALUout(-1) if (rt(0) == regDst(-1) | | rs(0) == regDst(-1)) && regWrite(-1)
Forward ALUout(-2) if (rt(0) == regDst(-2) | | rs(0) == regDst(-2)) && regWrite(-2)
Forward ALUout(-3) if (rt(0) == regDst(-3) | | rs(0) == regDst(-3)) && regWrite(-3)

Data Hazard and Stall

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.

Instruction	C0	C1	C2	C3	C4	C5
lw $t0, 20($s0)	IF	REG	EX	MEM	WB	
addiu $t1, $t0, $t0		IF	REG	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.

Instruction	C0	C1	C2	C3	C4	C5	C6
lw $t0, 20($s0)	IF	REG	EX	MEM	WB		
nop		IF	REG	EX	MEM	WB	
addiu $t1, $t0, $t0			IF	REG	EX	MEM	WB

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

Q3. Under what conditions do we need to introduce a nop? Under what conditions do we need to forward the output of the MEM stage to the EX stage? Assume you have the signals memToReg(n), rt(n), rs(n), regWrite(n), and regDst(n), where n is 0 for the signal of the current instruction being executed by the EX stage, -1 for the previous, etc.

We forward if (rt(0) == regDst(-2) || rs(0) == regDst(-2)) && memToReg(-2) && regWrite(-2)

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 two cycles. Instead, we add a branch comparator inside the register read stage and introduce the branch delay slot, and redefine MIPS so that the instruction after a branch statement will always be executed.

Q1. Reorder the following sets of instructions to account for the branch delay slot. You may have to insert a nop instruction.

Set 1	Reordered set 1	Set 2	Reordered Set 2
addiu $t0, $t1, 5	addiu $t0, $t1, 5	addiu $t0, $t1, 5	addiu $t0, $t1, 5
ori $t2, $t3, 0xff	beq $t0, $s0, label	ori $t2, $t3, 0xff	ori $t2, $t3, 0xff
beq $t0, $s0, label	ori $t2, $t3, 0xff	beq $t0, $t2, label	beq $t0, $t2, label
lw $t4, 0($t0)	lw $t4, 0($t0)	lw $t4, 0($t0)	nop
			lw $t4, 0($t0)