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.

Step 1: 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.

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

对于单周期CPU而言，时钟周期则应选择执行时间最长的lw指令作为结果。

$T1>= t_{clk\text{-}to\text{-}q}+t_{MEMread}+t_{RFread}+ t_{mux}+t_{ALU}+t_{MEMread}+t_{mux}+t_{RFsetup}=30+250+150+25+200+250+25+20=950ps$

$f1=1/T1<=1.05GHz$

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

选择5个阶段（IF、ID、EX、MEM、Wr）中最长的MEM阶段作为流水线CPU的时钟周期。

$T2>= t_{clk\text{-}to\text{-}q}+t_{MEMread}+ t_{mux}+t_{RFsetup}=30+250+25+20=325ps$

$f2=1/T2<=3.07GHz$

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

speed-up=950ps/325ps=2.92

因为每一步的时间并不均衡，所以跟单周期相比并没有提高5倍。

Step 3: 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.

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

and指令的C2周期和sw指令的C3周期需要的$t0，在addi指令的C4周期（WB结束）之后才能用。如果通过转发解决数据冒险的话，则将addi指令的ALU结果转发过去即可。

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.

当满足(rt(0)==regDst(-1)||rs(0)==regDst(-1))&&regWrite(-1)时则转发ALUout(-1)
当满足(rt(0)==regDst(-2)||rs(0)==regDst(-2))&&regWrite(-2)时则转发ALUout(-2)
当满足(rt(0)==regDst(-3)||rs(0)==regDst(-3))&&regWrite(-3)时则转发ALUout(-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

addiu指令需要$t0的值，是在C3周期的开始需要，而lw指令得到的$t0的值是在C3周期结束，故无法解决冒险。

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

加入nop指令后，addiu需要的$t0的值是在C4周期开始，而lw指令在C3周期末已经更新$t0的值，故addiu可以正常执行。

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.

当出现load-use冒险时，需要插入一条nop指令。

转发条件：(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 three 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)