CSE471—Computer Design and Organization

Prof. Steven Burns *Autumn 1994*

Lecture Notes: Sixteenth Day

Performance

- Let's compare the performance of the three MIPS subset implementation we've encountered so far and the one we'll be doing next:
  - Single-cycle
  - FSM-based multi-cycle
  - Micro-programmed-based multi-cycle
  - Pipelined

Ground Rules for Comparison

- We'll assume that control logic, muxes, and sign-extenders have insignificant delays compared to the major datapath units. We do this to make the analysis simple.
- We'll assign delay values to these major datapaths. We'll use

Adders, ALUs	10 ns
Memory access	10 ns
Register access	5 ns

- Then we'll determine the smallest legal clock period considering these delays.
- We'll also take the CPI into account.

Long (Critical) Delay Paths

- Remember that the clock period is determined by the longest path through the combinational logic separated by flip-flops.
- At what period can you run the processor for each instruction in the single-cycle implementation?

Inst Type	Imem	Reg R	ALU	Dmem	Reg W	Total
R-format	10	5	10		5	30 ns
Load	10	5	10	10	5	40 ns
Store	10	5	10	10		35 ns
Branch	10	5	10			25 ns
Jump	10					10 ns

- Unless we use a clock that changes period based on instruction type, we can't run this machine faster than 25 MHz (40 ns).

Multi-Cycle Implementation

- Here the CPI changes based on the instruction (from 3 to 5), but the clock period should be smaller. Let's figure out the minimum clock period, remembering that some actions occur in parallel:

Inst Type	Mem	Reg R	ALU	Reg W	Total
State 0	10		10		10 ns
State 1		5	10		10 ns
State 2			10		10 ns
State 3	10				10 ns
State 4				5	5 ns
State 5	10				10 ns
State 6			10		10 ns
State 7				5	5 ns
State 8			10		10 ns
State 9					0 ns

- We can run the FSM or microprogrammed engine at 100MHz (10ns period).

Conclusion

- Taking CPI and clock period into account, the single and multiple clock cycle implementation perform at about 25 native MIPS.
- To do better, we'll introduce pipelining.

Pipelining

- Pipelining will add a lot of complexity, but leads to high performance.
- The maximum speed up from pipelining is the number of pipeline stages.
- This is only achievable if we can break the computation into equal amounts of work.
- We can almost achieve this with our MIPS subset using the simple performance model (10 ns clocks, 5 pipeline stages, 40 ns worth of work to perform. The optimal would be 8 ns.)

Simple MIPS Pipeline

- Five stages
  - IF — instruction fetch
  - ID — instruction decode, register fetch
  - EX — ALU execute, memory address calculation
  - MEM — Data memory load or store
  - WB — Write back result to register
- Carefully crafted pipeline for balanced critical paths.
- Study Figures 6.3 through 6.11 for a walk through of how it works.

Pipeline Control

- We need to specify the control to the datapath in the various pipeline stages:
  - IF — no control signals
  - ID — no control signals
  - EX — *RegDst*, *ALUop*, *ALUSrc*
  - MEM — *Branch*, *MemRead*, *MemWrite*
  - WB — *MemtoReg*, *RegWrite*
- We can use the control signals values for a particular instruction as with the single-cycle implementation

Pipelining the Control

- We must delay the control signals by the appropriate number of clock cycles so that the control corresponds to the pipelined data.
- This is easy to do. Put it through pipelined registers just like the data.
- Example:

```
...
<before 2>
<before 1>
lw    $10, 9($1)       // 0 00 1  0 1 0  1 1
sub   $11, $2, $3      // 1 10 0  0 0 0  1 0
and   $12, $4, $5      // 1 10 0  0 0 0  1 0
or    $13, $6, $7      // 1 10 0  0 0 0  1 0
add   $14, $8, $9      // 1 10 0  0 0 0  1 0
<after 1>
<after 2>
...
```

Data Hazards

- Pipelining is not this simple.
- The pipelined machine we just described has a peculiar behavior if we read from a register that is written to in a previous instruction.
- Example:

```
...
<before 2>
<before 1>
sub    $2, $1, $3
and   $12, $2, $5
or    $13, $6, $2
add   $14, $2, $2
sw    $15, 100($2)
<after 1>
<after 2>
...
```

RAW Hazards

- Three ways to deal with Read-After-Write hazards:
  - Redefine the meaning of the instruction set so that this behavior is what is intended. The compiler is responsible for translating high-level source code to the funny architecture.
  - Modify the control hardware so the machine behaves like the single-cycle implementation would by introducing `nop` instructions where necessary.

- Modify the control and datapath by inserting a direct connection from stages further down the pipe to the input of the ALU thus *bypassing* the register file.

Introducing NOP Instructions

- We need to detect when RAW hazards will occur.
- The best stage to do this is in the ID stage.
- There is a potential RAW hazard if any of the following are true:

```
IF/EX.RegWrite &
(ID/EX.WriteRegisterRt  = IF/ID.ReadRegister1 |
 ID/EX.WriteRegisterRd  = IF/ID.ReadRegister1 |
 ID/EX.WriteRegisterRt  = IF/ID.ReadRegister2 |
 ID/EX.WriteRegisterRd  = IF/ID.ReadRegister2) |
EX/MEM.RegWrite &
(EX/MEM.WriteRegister = IF/ID.ReadRegister1 |
 EX/MEM.WriteRegister = IF/ID.ReadRegister2) |
MEM/WB.RegWrite &
(MEM/WB.WriteRegister = IF/ID.ReadRegister1 |
 MEM/WB.WriteRegister = IF/ID.ReadRegister2)
```

Stalling

- If a RAW hazard is detected, we stall the ID and the IF stages by not updating the PC nor the IF/ID register.
- We insert a `nop` into the pipeline by specifying all zero control signals for the ID/EX register. With these control values, the datapath will not change any state of the machine.