University of Arizona

CSc 340

Foundations of Computer Systems

Christian Collberg

February 27, 2001

Delay Slots

Pipeline

- Modern CPUs improve performance by executing more than one instruction at once. This is obviously very complicated, and the details are covered in a computer architecture course. It does affect how the CPU is programmed, which is why we have to worry about it.
- A common technique for executing multiple instructions simultaneously is called pipelining. Basically, a new instruction is started every clock cycle, but each instruction takes several cycles to finish. During each cycle an individual instruction uses a different CPU component, allowing multiple instructions to execute simultaneously.

Slide 12–1

Pipelining. . .

- Doing laundry is an example of pipelining. There are three steps — washing, drying, and folding. One load of clothes can be in each stage of the laundry pipeline.
- A canonical CPU has five stages in its pipeline:

  Fetch: fetch the instruction from memory, increment PC

  Decode: decode, access register operands, change PC if branch taken

  Execute: perform the ALU operation (address calculation)

  Memory: access memory

  Write-back: write result to register file

Slide 12–2

Pipelining ⌜`lw $1,100($2)`⌝

1. **Fetch**
   - Get the instruction from memory.
   - PC=PC+1.
2. **Decode**
   (a) It's a `lw` instruction,
   (b) It stores into register $1,
   (c) Look up the value of register $2.
3. **Execute**
   - Compute 100 + $2.
4. **Memory**
   - Load value at location 100 + $2.
5. **Write-back**
   - Store the loaded value into register $1.

Slide 12–3

Pipelining ⌜`add $3,$4,$5`⌝

1. **Fetch**
   - Get the instruction from memory.
   - `PC=PC+1`.
2. **Decode**
   (a) It's an `add-register` instruction,
   (b) It stores into register `$3`,
   (c) Look up the value of registers `$4` and `$5`.
3. **Execute**
   - Compute `$4 + $5`.
4. **Memory**
5. **Write-back**
   - Store the loaded value into register `$3`.

Slide 12–4

Pipeline Example I

Without pipelining

Clock Cycle	1	2	3	4	5	6	7	
`lw $1,100($0)`	Fetch	Decode	Execute	Memory	Write			
`lw $2,200($0)`						Fetch		…
`lw $3,300($0)`		(This instruction doesn't start until cycle 11)						…

With pipelining

Clock Cycle	1	2	3	4	5	6	7	
`lw $1,100($0)`	Fetch	Decode	Execute	Memory	Write			
`lw $2,200($0)`		Fetch	Decode	Execute	Memory	Write		
`lw $3,300($0)`			Fetch	Decode	Execute	Memory		…

Slide 12–5

Pipeline Example II

Clock Cycle	1	2	3	4	5	6	7
`lw $1,100($2)`	Fetch Fetch instr-uction	Decode Fetch value of reg `$2`	Execute Compute 100+ Cont(`$2`)	Memory Load Mem[100+ Cont(`$2`)]	Write Store value in reg `$1`		
`add $3,$4,$5`		Fetch Fetch instr-uction	Decode Fetch value of `$4` and `$5`	Execute Compute `$4 + $5`	Memory	Write Store result in reg `$3`	
`sub $6,$7,$8`			Fetch Fetch instr-uction	Decode Fetch value of `$7` and `$8`	Execute Compute `$7 - $8`	Memory	…

Slide 12–6

Five-Stage Pipeline

- A pipeline with five stages allows five instructions to execute at once:

i	F	D	E	M	W					
$i+1$		F	D	E	M	W				
$i+2$			F	D	E	M	W			
$i+3$				F	D	E	M	W		
$i+4$					F	D	E	M	W	
$i+5$						F	D	E	M	W

Slide 12–7

Slide 12–8

Hazards

- The MIPS instruction set was designed for pipelining:
  1. Instructions are all 4 bytes, so that they can be fetched in the first stage and decoded in the second.
  2. The source registers fields are in the same location in the instructions, so that the register file can be read while the instruction is decoded.
  3. Addresses only appear in loads and stores, so that the address can be computed in stage 3 and used in stage 4.
- A hazard is a situation that prevents the next instruction from executing in its designated clock cycle. Hazards cause pipeline bubbles and stall the pipeline. This reduces performance.

Slide 12–9

Branch Delay Slots

- Any instruction that changes the PC causes a control hazard. The CPU determines that an instruction might change the PC during the decode stage. Note that the instruction at address PC+4 is fetched on the same cycle.
- One possibility is to cancel and refetch the next instruction, but this causes the CPU to stall (a pipeline bubble) until the PC has been changed.

i	F	D	E	M	W				
$i+1$		F	F	D	E	M	W		
$i+2$				F	D	E	M	W	
$i+5$					F	D	E	M	W

Slide 12–10

Branch Delay Slots...

- Note that instruction $i+1$ is fetched twice; once before instruction i changes the PC, and again after instruction i has (possibly) changed the PC. If the PC was changed (the branch was taken) a different instruction is fetched the second time.
- The MIPS CPU has special hardware to determine if the branch is taken and update the PC in the decode stage. It also supports only simple conditional branches. Otherwise the branch could not be performed until after the execute stage, stalling the pipeline even longer.

Slide 12–11

Branch Delay Slots...

- Many RISC CPUs simply execute the next instruction rather than stall the pipeline, leading to a branch delay slot. The instruction following one that changes the PC is always executed, even if the PC is changed.
- The trick is to put an instruction into the branch delay slot that should be executed whether or not the branch is taken.
- The easiest to use is the nop (”no-operation”) instruction that doesn’t do anything. It’s easy, but doesn’t improve performance.

Slide 12–12

Branch Delay Slots. . .

- Your program will run faster if you put an instruction that logically belongs before the branch into the slot as long as the branch doesn't depend on it. E.g.

```
add     $t0, $t1, $t2
b       foo
nop
```

becomes

```
b       foo
add     $t0, $t1, $t2
```

Slide 12–13

Branch Delay Slots. . .

- Make sure you don't fill the slot with a pseudo-instruction that expands into several real instructions.
- The MIPS assembler automatically fills in branch delay slots for you by looking for an instruction that can be moved into the slot. It works about 50% of the time, and you don't have to worry about delay slots when programming the MIPS.
- You can use the ⌜`.set noreorder`⌝ directive to tell the MIPS assembler not to reorder instructions to fill the delay slot (SPIM doesn't accept this directive). Use ⌜`.set reorder`⌝ to turn reordering back on.

Slide 12–14

Example – Version I

- Consider a subroutine called `length` that returns the number of non- zero characters in a null-terminated ASCII string.
- `$t0` counts all characters, including the zero character at the end of the string. `$v0` (the return value) is therefore `$t0 - 1`.
- The body of the loop contains 5 instructions, so it takes 5 instruction to process each character.

Slide 12–15

Example – Version I

```
        .set    noreorder       # don't reorder insts
length: subu    $sp, $sp, 24
        move    $t0, $zero      # count = 0
loop:                           # repeat
        lbu     $t1, 0($a0)     # get char
        addu    $t0, $t0, 1     # count = count+1
        addu    $a0, $a0, 1     # next char addr
        bnez    $t1, loop       # until char = 0
        nop                     # delay slot
        subu    $v0, $t0, 1     # non-zero = count-1
        addu    $sp, $sp, 24
        jr      $ra
```

Slide 12–16

Example – Version II

- Placing the increment instruction into the branch delay slot reduces the body of the loop to 4 instructions.
- Now there are only 4 instructions in the loop, but we have to remember that the `$a0` is incremented when the branch is taken, even though the instruction follows the branch.
- Other uses for the branch delay slot:
  1. Putting a parameter into `$a0-$a3`.
  2. Putting the return value into the return register `$v0`.

Slide 12–17

Example – Version II

```
        .set    noreorder       # don't reorder insts
length: subu    $sp, $sp, 24
        move    $t0, $zero      # count = 0
loop:                           # repeat
        lbu     $t1, 0($a0)     # get char
        addu    $t0, $t0, 1     # count = count+1
        bnez    $t1, loop       # until char = 0
        addu    $a0, $a0, 1     # next char addr
        subu    $v0, $t0, 1     # non-zero = count-1
        addu    $sp, $sp, 24
        jr      $ra
```

Slide 12–18

Load Delay Slots

- Suppose an instruction loads a value into a register, and the next instruction uses that value. The pipeline looks like this:

i	F	D	E	M	W	
$i+1$		F	D	E	M	W

- Instruction i reads the value from memory during its memory stage, but instruction $i+1$ is in its execute stage on the same cycle, and needs the value.

Slide 12–19

Load Delay Slots

- This is a type of data hazard called a load delay. Most CPUs will stall the pipeline to prevent instruction $i+1$ from using the wrong value:

i	F	D	E	**M**	W		
$i+1$		F	D	**stall**	E	M	W

- The MIPS doesn't stall the pipeline – you can't use the result of a load in the next instruction. This is called a load delay slot. Fortunately, the assembler automatically reorders instructions to fill the load delay slot.

Slide 12–20

Load Delay Slots – Reordering

```
.set      noreorder
lw        $t1, 0($t0)
add       $t1, $t1, 1     # data hazard!!
```

- This won't work because the add instruction that uses \$t1 is in the delay slot of the load. The assembler should catch this and complain. You have to change it to:

Slide 12–21

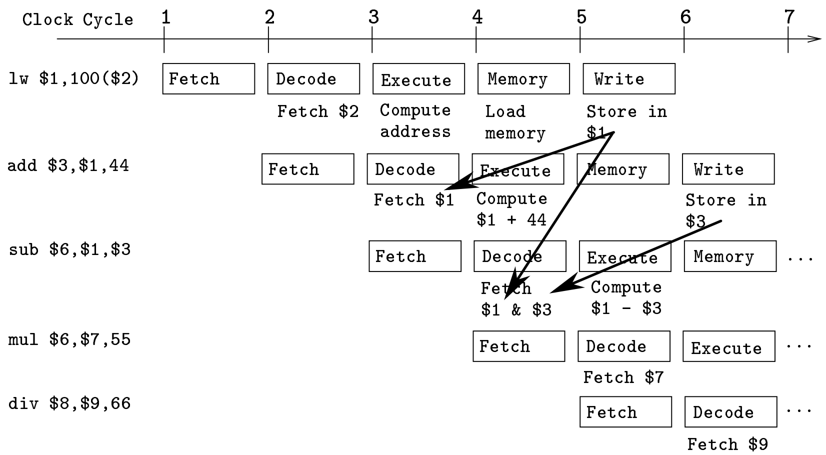
Load Delay Slots – Reordering...

```
.set      noreorder
lw        $t1, 0($t0)
nop
add       $t1, $t1, 1
```

- Even if your CPU stalls to avoid the hazard, stalling reduces performance. To improve performance you should avoid the hazard when you write the program. In general, separate a load from the first instruction that uses the value by as many instructions as possible.

Slide 12–22

Pipeline Data Hazards I



Slide 12–23

Pipeline Data Hazards II

- The previous example shows one kind of problem we have with pipelines. Data hazards occur when the result of one instruction isn't ready in time for when that result is needed.
- Some processors will detect such situations and **stall** the pipeline until the value is ready. This is called a **harware/pipeline interlock**.
- Interlocks are expensive (extra control logic on the chip which takes up valuable chip real estate) so some processors do away with them. Instead they rely on compilers and assemblers to insert NOPs when needed.

Slide 12–24

Pipeline Data Hazards III

- Example: On a MIPS 2000, the value loaded by a `lw` instruction isn't available to the immediately following instruction. The processor doesn't have hardware interlocks. If you give the following code to the assembler

```
lw     100($4), $5
add    $5, $5, 56
```

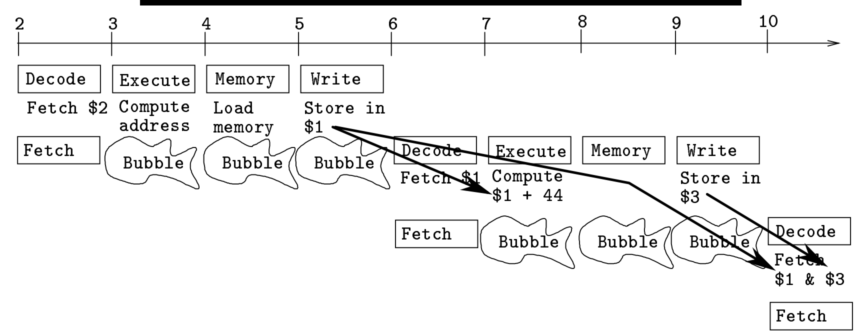
it will insert the necessary NOPs:

```
lw     100($4), $5
nop
add    $5, $5, 56
```

- Even if the hardware does have interlocks, the compiler (or assembler) should pay attention to the order in which instructions are scheduled. A well scheduled program may be upt o 50% faster than an naïvely scheduled one.

Slide 12–25

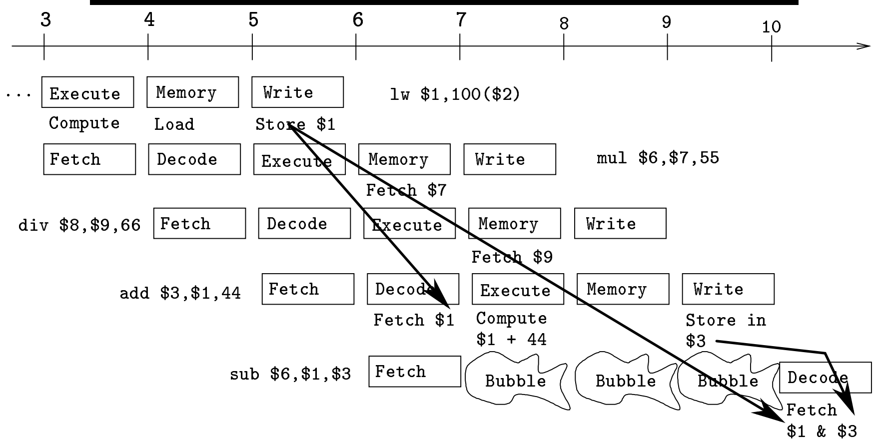
Pipeline Data Hazards – Stalls



- Some processors take care of data hazards themselves – the pipeline is **stalled** for a number of cycles until the hazard is resolved. This is like inserting one or more **bubbles** (NOPs) in the pipeline.

Slide 12–26

Pipeline Data Hazards – Reordering



- The compiler/assembler will sometimes reorder instructions to avoid stalls.

Slide 12–27

Readings and References

- Maccabe, section 4.5.7