CSCE 4610: Computer Architecture

HW #5 C.5, C.6, C7
Due March 21 (Tuesday after Spring break)

EXAM 2: Tuesday April 4, 2017
Help Session: Friday March 31 at 1pm, B192

Problems for HW 5 and possibly on next exam may require you to redesign pipeline to implement different instructions.

I also noticed some confusion about what hardware does and compiler/programs do.

Hardware only deals with binary values – there are no strings or Alphabets

Even if the data being processed is text (character strings)

the data will be presented using ASCII or Unicode binary values

Remember both instructions and data are in memory

Consider for example code and the binary executable

On the left is symbolic or assembly code. It will be converted into binary (below)

Note the label Loop does not exist in binary version. The address of that instruction will be used in branch instructions.

<table>
<thead>
<tr>
<th>Loop:</th>
<th>Opcode</th>
<th>Rd</th>
<th>Rs1</th>
<th>Rs2</th>
<th>funct</th>
<th>Displacement</th>
<th>Data memory address</th>
<th>Contents (actual data)</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD</td>
<td>100011</td>
<td>0001</td>
<td>0010</td>
<td>0000</td>
<td>0000</td>
<td></td>
<td>xxxx40000</td>
<td>000………….0001</td>
</tr>
<tr>
<td>DADDI</td>
<td>001000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td></td>
<td>xxxx40004</td>
<td>0000000000000000</td>
</tr>
<tr>
<td>SD</td>
<td>101010</td>
<td>0001</td>
<td>0010</td>
<td>0000</td>
<td>0000</td>
<td></td>
<td>xxxx40004</td>
<td>0000000000000000</td>
</tr>
<tr>
<td>DADDI</td>
<td>100100</td>
<td>0010</td>
<td>0001</td>
<td>0000</td>
<td>0000</td>
<td></td>
<td>xxxx40000</td>
<td>0000000000000000</td>
</tr>
<tr>
<td>DSUB</td>
<td>100100</td>
<td>0010</td>
<td>0001</td>
<td>0000</td>
<td>0000</td>
<td></td>
<td>xxxx40000</td>
<td>0000000000000000</td>
</tr>
<tr>
<td>BNEZ</td>
<td>100100</td>
<td>0010</td>
<td>0011</td>
<td>0010</td>
<td>0000</td>
<td></td>
<td>xxxx40010</td>
<td>0000000000000000</td>
</tr>
<tr>
<td></td>
<td>000101</td>
<td>0010</td>
<td>0011</td>
<td>0011</td>
<td>1111111111111110</td>
<td></td>
<td>xxxx40014</td>
<td>0000000000000000</td>
</tr>
</tbody>
</table>
CSCE 4610/4610: Computer Architecture

Reasons for not achieving a speed up that is equal to the number of pipeline stages

3 type of “hazards” that require pipeline stalls

1. **Structural Hazards** due to resource dependencies
2. **Data hazards** due to data dependencies
3. **Control hazards** due to branch instructions

**Data Hazards:**
Consider the following example:

- `sub R2, R1, R3`
- `and R12, R2, R5`
- `or R13, R6, R2`
- `add R14, R2, R2`
- `sw R15, 100(R2)`

When is the result from `sub` written to `R2`?

- Write-Back stage.
- But *and* instruction already read its register operands when it reached ID, therefore getting old value from `R2`

---

**Data (or Result) forwarding**

1. Need to detect the hazard among instructions
2. Need to decide on when (and where) to forward

Detection of data hazards:

We use the information in pipestage latches

For example:

1. `ADD R1, R2, R3`
2. `SUB R4, R1, R5`

When instruction 2 is in decode, instruction 1 is in Execute

\[ \text{IF} \text{(IR}(rs1)) = \text{ID}/\text{EX}(\text{IR}(rd)) \]

indicates that the destination of the instruction in EX is the first Source of the instruction in ID.
Likewise IF/ID(IR(rs2)) = ID/EX(IR(rd)) indicates that the destination of the instruction in EX is the second source of the instruction in ID.

To stall an instruction in the pipeline:
1. Send no-op down the pipeline.
2. Prevent "next" instruction overwriting stalled instruction. This can be easily achieved using write-enable inputs associated with latches.

We need to similarly establish conditions to detect hazards between Load and subsequent Arithmetic operations, or between arithmetic operations and subsequent store, between arithmetic operations and branch/compare instructions.
## Pipelining: Basic and Intermediate Concepts

This table shows all conditions to test and forward data (original pipeline):

<table>
<thead>
<tr>
<th>Pipeline register containing source instruction</th>
<th>Opcode of source instruction</th>
<th>Pipeline register containing destination instruction</th>
<th>Opcode of destination instruction</th>
<th>Destination of the forwarded result</th>
<th>Comparison if equal then forward</th>
</tr>
</thead>
<tbody>
<tr>
<td>EX/MEM</td>
<td>Register-register ALU</td>
<td>ID/EX</td>
<td>Register-register ALU, ALU immediate, load, store, branch</td>
<td>Top ALU input</td>
<td>EX/MEM.IR[r] (=) ID/EX.IR[rs]</td>
</tr>
<tr>
<td>EX/MEM</td>
<td>Register-register ALU</td>
<td>ID/EX</td>
<td>Register-register ALU, ALU immediate, load, store, branch</td>
<td>Bottom ALU input</td>
<td>EX/MEM.IR[r] (=) ID/EX.IR[rs]</td>
</tr>
<tr>
<td>MEM/WB</td>
<td>Register-register ALU</td>
<td>ID/EX</td>
<td>Register-register ALU, ALU immediate, load, store, branch</td>
<td>Top ALU input</td>
<td>MEM/WB.IR[rt] (=) ID/EX.IR[rs]</td>
</tr>
<tr>
<td>MEM/WB</td>
<td>Register-register ALU</td>
<td>ID/EX</td>
<td>Register-register ALU, ALU immediate, load, store, branch</td>
<td>Bottom ALU input</td>
<td>MEM/WB.IR[rt] (=) ID/EX.IR[rs]</td>
</tr>
<tr>
<td>MEM/WB</td>
<td>ALU immediate</td>
<td>ID/EX</td>
<td>Register-register ALU, ALU immediate, load, store, branch</td>
<td>Top ALU input</td>
<td>EX/MEM.IR[rt] (=) ID/EX.IR[rs]</td>
</tr>
<tr>
<td>MEM/WB</td>
<td>ALU immediate</td>
<td>ID/EX</td>
<td>Register-register ALU, ALU immediate, load, store, branch</td>
<td>Bottom ALU input</td>
<td>EX/MEM.IR[rt] (=) ID/EX.IR[rs]</td>
</tr>
<tr>
<td>MEM/WB</td>
<td>ALU immediate</td>
<td>ID/EX</td>
<td>Register-register ALU, ALU immediate, load, store, branch</td>
<td>Top ALU input</td>
<td>MEM/WB.IR[rt] (=) ID/EX.IR[rs]</td>
</tr>
<tr>
<td>MEM/WB</td>
<td>ALU immediate</td>
<td>ID/EX</td>
<td>Register-register ALU, ALU immediate, load, store, branch</td>
<td>Bottom ALU input</td>
<td>MEM/WB.IR[rt] (=) ID/EX.IR[rs]</td>
</tr>
<tr>
<td>MEM/WB</td>
<td>Load</td>
<td>ID/EX</td>
<td>Register-register ALU, ALU immediate, load, store, branch</td>
<td>Top ALU input</td>
<td>MEM/WB.IR[rt] (=) ID/EX.IR[rs]</td>
</tr>
<tr>
<td>MEM/WB</td>
<td>Load</td>
<td>ID/EX</td>
<td>Register-register ALU, ALU immediate, load, store, branch</td>
<td>Bottom ALU input</td>
<td>MEM/WB.IR[rt] (=) ID/EX.IR[rs]</td>
</tr>
</tbody>
</table>

Diagram of computer architecture showing pipeline stages (EX, ID, MEM, WB) with register files, ALU, and branch prediction logic.
Note it is not always possible to forward data

1. LW R4, 0 (R1)
2. ADD R1, R4, R3

In this case we need to Stall ADD one cycle until LW completes MEMORY

An example (Problem C.1 from Textbook)

<table>
<thead>
<tr>
<th>Loop:</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD</td>
<td>F</td>
<td>D</td>
<td>E</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADDI</td>
<td>F</td>
<td>S</td>
<td>S</td>
<td>D</td>
<td>E</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SD</td>
<td>F</td>
<td>S</td>
<td>S</td>
<td>D</td>
<td>E</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADDI</td>
<td>F</td>
<td>D</td>
<td>E</td>
<td>M</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

First – no data forwarding
(Assume registers can be written and read in the same cycle, during writeback)
Need 16 cycles to complete one iteration of the loop
(LD-F in cycles 1 and 17)

If we do not use data forwarding

We start the LD from next iteration in cycle 17
So, on average we complete one iteration in 16 cycles
CSCE 4610/4610: Computer Architecture

The loop is repeated 99 times
(R3 is 396 larger than R2; and R2 is incremented by 4 each time)

So we need 16*99 cycles to complete the loop.

Now let us consider data forwarding between instructions.

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD</td>
<td>F</td>
<td>D</td>
<td>E</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DADDI</td>
<td>F</td>
<td>S</td>
<td>D</td>
<td>E</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SD</td>
<td>F</td>
<td>D</td>
<td>E</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DADDI</td>
<td>F</td>
<td>D</td>
<td>E</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DSUB</td>
<td>F</td>
<td>D</td>
<td>E</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BNEZ</td>
<td>F</td>
<td>S</td>
<td>S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OUTSIDE</td>
<td>F</td>
<td>S</td>
<td>S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LD</td>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Now we need only 9 cycles to complete one iteration

CSCE 4610/4610: Computer Architecture

Control Hazards
Consider a conditional branch instruction
We do not know that it is a branch until ID
we already fetched the next instruction
We do not know the decision until EX
We do not know the address of the branch destination until EX

If not taken
branch instruction (i) IF ID EX MEM WB
i+1 IF S ID EX MEM WB
i+2 IF ID EX MEM WB
Loss of one cycle

If taken
branch instruction (i) IF ID EX MEM WB
i+1 IF S - - - -
target instruction IF ID EX MEM WB
Loss of two cycles
a). When do we know that the instruction is a branch (or jump)
b). When do we know if the branch will be taken or not

We know about branch instructions in ID stage
Original pipeline – decision known in EX state

What can we do to improve the situation?

If the condition is simple -- zero testing
a value set by a previous compare instruction
We use two instructions: Compare followed by Zero test
We can move the testing to ID (decision will be known in ID)
CSCE 4610/4610: Computer Architecture

If not taken
branch instruction (i) IF ID EX MEM WB
i+1 IF ID EX MEM WB
i+2 IF ID EX MEM WB

Loss of zero cycles

What if branch is taken?
We need to compute the address of the target -- need to add displacement to PC
Normally done in EX
But we can add an extra adder to ID stage to compute this address

If taken
branch instruction (i) IF ID EX MEM WB
i+1 IF S

target instruction
IF ID EX MEM WB

Loss of one cycle

CSCE 4610 March 9, 2017

15

CSCE 4610/4610: Computer Architecture

Now Consider the application of delayed branch

If not taken
branch instruction (i) IF ID EX MEM WB
i+1 (delayed br slot) IF ID EX MEM WB
i+2 IF ID EX MEM WB

If taken
branch instruction (i) IF ID EX MEM WB
i+1 (delayed br slot) IF ID EX MEM WB
branch target IF ID EX MEM WB

No Loss!
CSCE 4610/4610: Computer Architecture

Another example.

Same example as before with data forwarding we have seen with 9 cycles per iteration

Loop:

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>LD</td>
<td>F</td>
<td>D</td>
<td>E</td>
<td>M</td>
<td>W</td>
</tr>
<tr>
<td>SD</td>
<td>F</td>
<td>D</td>
<td>E</td>
<td>M</td>
<td>W</td>
</tr>
<tr>
<td>DADDI</td>
<td>F</td>
<td>D</td>
<td>E</td>
<td>M</td>
<td>W</td>
</tr>
<tr>
<td>DADDI</td>
<td>F</td>
<td>D</td>
<td>E</td>
<td>M</td>
<td>W</td>
</tr>
<tr>
<td>DSUB</td>
<td>F</td>
<td>D</td>
<td>E</td>
<td>M</td>
<td>W</td>
</tr>
<tr>
<td>BNEZ</td>
<td>F</td>
<td>D</td>
<td>E</td>
<td>M</td>
<td>W</td>
</tr>
<tr>
<td>OUTSIDE</td>
<td>F</td>
<td>S</td>
<td>S</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LD</td>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Here we are stalling on detecting a branch

CSCE 4610 March 9, 2017

CSCE 4610/4610: Computer Architecture

Let us consider using the Delayed branch.

Can we consider re-ordering the code to use delay slot?

Loop:

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>LD</td>
<td>F</td>
<td>D</td>
<td>E</td>
<td>M</td>
<td>W</td>
</tr>
<tr>
<td>DADDI</td>
<td>F</td>
<td>D</td>
<td>E</td>
<td>M</td>
<td>W</td>
</tr>
<tr>
<td>DADDI</td>
<td>F</td>
<td>D</td>
<td>E</td>
<td>M</td>
<td>W</td>
</tr>
<tr>
<td>DADDI</td>
<td>F</td>
<td>D</td>
<td>E</td>
<td>M</td>
<td>W</td>
</tr>
<tr>
<td>DSUB</td>
<td>F</td>
<td>D</td>
<td>E</td>
<td>M</td>
<td>W</td>
</tr>
<tr>
<td>BNEZ</td>
<td>F</td>
<td>D</td>
<td>E</td>
<td>M</td>
<td>W</td>
</tr>
<tr>
<td>SD</td>
<td>F</td>
<td>D</td>
<td>E</td>
<td>M</td>
<td>W</td>
</tr>
<tr>
<td>S</td>
<td>S</td>
<td>S</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LD</td>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Now we need only 8 cycles to complete one iteration.
Suppose branch frequencies are given as follows:
- Conditional branches = 15%
- Taken branches = 60%
- Unconditional branches = 1%

We have a 4 stage pipeline where the branch is resolved at the end of the second cycle for unconditional branches and at the end of the 3rd cycle for conditional branches.

Consider unconditional branches like jumps or calls:

<table>
<thead>
<tr>
<th>Branch</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Branch+1</td>
<td>IF</td>
<td>S</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>From target</td>
<td>IF</td>
<td>ID</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

We still have to wait until EX to find branch address. So we lost 2 cycles. The contribution of such instructions = 0.01 * 2 = 0.02 cycles.

Consider conditional branches. Decision known at the end of third cycle.

- Not taken case:
<table>
<thead>
<tr>
<th>Branch</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>Branch+1</td>
<td>IF</td>
<td>S</td>
<td>ID</td>
<td>EX</td>
</tr>
<tr>
<td>Branch+2</td>
<td>IF</td>
<td>ID</td>
<td>WB</td>
<td></td>
</tr>
</tbody>
</table>

Again we lost 1 cycle. The contribution is 0.15 * 0.4 * 1 = 0.06 cycles.

- Taken case:
<table>
<thead>
<tr>
<th>Branch</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>Branch+1</td>
<td>IF</td>
<td>S</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Target</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td></td>
</tr>
</tbody>
</table>

Now we have lost 2 cycles. The contribution is 0.15 * 0.6 * 2 = 0.18 cycles.

The total loss = 0.02 + 0.06 + 0.18 = 0.26 cycles.
CSCE 4610/4610: Computer Architecture

Branch prediction
Static prediction – always predict not taken

If not taken
branch instruction (i) IF ID EX MEM WB
i+1 IF ID EX MEM WB
i+2 IF ID EX MEM WB

No lost cycles

If taken
branch instruction (i) IF ID EX MEM WB
i+1 IF ID EX MEM WB
i+2 branch target IF ID EX MEM WB

2 lost cycles

Dynamic Branch prediction
Can we change the prediction dynamically?

CSCE 4610 March 9, 2017

CSCE 4610/4610: Computer Architecture

Dynamic Branch prediction
Can we change the prediction dynamically?

We will come back to 2-bit predictors and other techniques later.

Another issue to deal with is the case when the EX stage takes different amounts of time for different instruction types.

Consider the following example of an architecture

Fig C.35, page C-54
CSCE 4610/4610: Computer Architecture

Two important terms to remember

**Delay** – how soon another instruction of the same kind can be issued

**Latency** – how soon can the results of an instruction can be made available
(for data forwarding)

FP divide is not pipelined and needs 24 cycles to complete
but using forwarding, we can forward data from the last
stage back to itself
So latency = 23 and inter-instruction initiation delay = 24

FP multiply --- pipelined and has 7 stages
data can be forwarded from M7 to M1
latency = 6; delay = 1

FP add -- pipelined with 4 stages and data from A4 can be forwarded to A1
latency = 3; delay = 1