• CSCE 5610 Substitute instructor
• Charles F. Shelor
• Charles.shelor@gmail.com

CSCE 5610: Computer Architecture

Midterm: Tuesday Oct. 16, 2018
in F 280
Open Books and Open Notes
Sample questions posted

Review
Pipeline Hazards

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

Consider another example of data dependencies from textbook page C-13

add    x1, x2, x3
sub    x4, x1, x5
and    x6, x1, x7
or     x8, x1, x9
xor    x10, x1, x11
**CSCE 5610: Computer Architecture**

```
add x1, x2, x3
sub x4, x1, x5
and x6, x1, x7
```

If we stall `sub`, all the other instructions are also delayed sub 2 cycles and x1 will be available to them without further delays.

When is the result of add instruction available?

- At the end of Execute cycle
- Can we use the output of ALU as input to the next instruction?

We can forward data from add to sub:

- `sub` will not read register x1 in Decode stage
- the ALU output from add is fed back as input to `sub` instruction

sub instruction does not read x1 from registers

It waits for the result of add instruction and uses the value

Result (or data) forwarding
CSCE 5610: Computer Architecture

It may not always be possible to forward data and eliminate all stalls

Consider

1. \texttt{ld} x1, 0(x2)
2. \texttt{sub} x4, x1, x5
3. \texttt{and} x6, x1, x7
4. \texttt{or} x8, x1, x9

By the time instruction 1 completes execution (in MEM), instruction 2 is already in EX. Therefore, we can’t forward data for x1 in this case; a stall is required.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{ld} x1, 0(x2)</td>
<td>IF</td>
<td>ID</td>
<td>Stall</td>
<td>EX</td>
<td>MEM</td>
</tr>
<tr>
<td>\texttt{sub} x4, x1, x5</td>
<td>IF</td>
<td>ID</td>
<td>Stall</td>
<td>ID</td>
<td>EX</td>
</tr>
<tr>
<td>\texttt{and} x6, x1, x7</td>
<td>IF</td>
<td>Stall</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
</tr>
<tr>
<td>\texttt{or} x8, x1, x9</td>
<td>Stall</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
</tr>
</tbody>
</table>

An example

<table>
<thead>
<tr>
<th>Instruction</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>\texttt{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>\texttt{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>\texttt{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>\texttt{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>
<tr>
<td>\texttt{add}</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>\texttt{sub}</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>\texttt{bnez}</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>\texttt{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>
<td></td>
</tr>
<tr>
<td>\texttt{ld}</td>
<td>F</td>
<td></td>
<td></td>
<td></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)
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></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>D</td>
<td>E</td>
<td>M</td>
<td>W</td>
<td></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>
<td></td>
</tr>
<tr>
<td>addi</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>sub</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>bnez</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>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>
<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>
<td></td>
</tr>
</tbody>
</table>

Now we need only 9 cycles to complete one iteration.

Control Hazards

We detect a branch in Decode (ID) stage
We know if branch is taken or not in EX stage
We also know the address of branch target in EX stage

Choice 1: Stall instructions when branch is decoded

**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
CSCE 5610: Computer Architecture

Choice 2: 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 IF ID EX MEM WB
branch target IF ID EX MEM WB

  2 lost cycles

CSCE 5610 Oct. 11, 2018

CSCE 5610: Computer Architecture

Choice 3: 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 S IF ID EX MEM WB

  1 cycle lost

CSCE 5610 Oct. 11, 2018
Consider our example code for a loop and used delayed branch

**Loop:**

1. `ld x1, 0(x2)`
2. `addi x1, x1, #1`
3. `sd x1, 0(x2)`
4. `addi x2, x2, #4`
5. `sub x4, x3, x2`
6. `bnez x4, Loop`

```
Loop:
ld x1, 0(x2)
addi x1, x1, #1
addi x2, x2, #4
sub x4, x3, x2
bnez x4, Loop
```

Now we need only 8 cycles to complete one iteration.

Can the remaining stall cycles be eliminated? Switch addi instrs; Move br impl to ID stage

**Choice 4: dynamic branch prediction**

We have seen the impact of predict taken vs not-taken
such predictions are static -- can’t change during program execution
Now we will see if we can do the prediction dynamically.

Consider a simple idea. With each branch instruction, we use one bit of the instruction as a prediction bit.
Compiler can initially set the bit for a prediction (either taken or not taken)
Hardware can change this on a mis-prediction

Consider an example

**Loop:**

1. `...`
2. `BNEQ R1, R2, done`

<table>
<thead>
<tr>
<th>opcode</th>
<th>Rs1</th>
<th>Rs2</th>
<th>Target displacement</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Prediction bit
Instead of associating the prediction bit with the instruction, let us consider a separate prediction memory.

How do we address this memory? And how large should the memory be (how many entries)?

At least one entry for each branch instruction in program

Use the instruction address of the branch instruction to address the prediction memory.

Actually we may use only a part of the PC -- otherwise, you need $2^{32}$ entries in the prediction memory.

We look up the prediction memory during IF stage.

We use the prediction information only if the instruction is actually a branch.

Using 1-bit predictor can be bad.

Consider a nested loop

```
for (…) {
    for (..) {…}
}
```

For the branch test of the inner loop, we will mis-predict two times in a row.

```
for (…) {
    …
}
```

Instead of one bit we will use two bits to make sure that we do not change prediction unless you mis-predict twice in a row.

- 00: predict taken
- 01: predict taken (but 1-misprediction)
- 11: predict not taken
- 10: predict not taken (but 1-misprediction)

This material is from Appendix C (starting from page C-23)
Consider the following state diagram to see how a 2-bit predictor works.

What type of hardware (or circuit is needed)? – page C-24

Sequential circuit -- 2 flip flops.

- the input is the decision of the branch (taken or not taken – correct prediction or wrong prediction)
- and the current values of the flip flops.

As with one bit predictors, we keep prediction bits separate from instruction memory.

For example, a 4096 entry 2-bit array. We use the least significant 12 bits of PC as an index.
This graph shows that branch prediction does better for floating point programs. Why?

FP applications are scientific applications involve loops – you can more accurately predict loops as they are likely to be taken.
For current branch we have two 2-bit predictors based on previous branch decision –

one 2-bit predictor if previous branch is taken and another if the previous branch is not taken

We can use 1-bit flip flop to indicate if previous branch is taken or not

Use this 1 bit to select one of two 2-bit predictions for current branch

Let us look at a comparison of correlation branch prediction with 2 bit predictor

The correlation predictor performs better particularly for Integer benchmarks.

(2,2) mean we have 2 bits for previous branches or remember 2 branches

and 2-bit predictors

(or four 1024 entry (2bit) predictors)
More complex predictions -- Global share
Keep the history of last k branches (say 0 means not taken, 1 mean taken)
use this bit vector as an index into pattern history table (PHT).

Use the k-bit history along with PC bits to index Branch History Table
- Gselect – concatenate
- Gshare – XOR (better)

More complex predictions –
For some branches local BHT performed better
For other using global history performed better
So switch between these using tournament style selection
Tournament Predictor in Alpha 21264

- 4K 2-bit counters to choose from among a global predictor and a local predictor
- Global predictor also has 4K entries and is indexed by the history of the last 12 branches; each entry in the global predictor is a standard 2-bit predictor
  - 12-bit pattern: ith bit is 0 \( \Rightarrow \) ith prior branch not taken;
  - ith bit is 1 \( \Rightarrow \) ith prior branch taken;

Local predictor consists of a 2-level predictor:
- Top level a local history table consisting of 1024 10-bit entries; each 10-bit entry corresponds to the most recent 10 branch outcomes for the entry. 10-bit history allows patterns 10 branches to be discovered and predicted
- Next level Selected entry from the local history table is used to index a table of 1K entries consisting a 3-bit saturating counters, which provide the local prediction

Total size: \( 4K \times 2 + 4K \times 2 + 1K \times 10 + 1K \times 3 = 29K \) bits! (~180K transistors)

PROJECT IDEA. Read about various branch prediction techniques. Collect benchmark data and write a report.
Branch prediction is only half the story. We still need to compute the address of target of the branch:
- PC+4 if not taken
- PC + displacement if branch is taken

Along with prediction (say 2-bit predictor), save predicted next instruction address:
- Branch Target Buffer (pre-computed addresses of all branch destinations)
- ADDRESSED USING PC – Any problems?

We are only using the least significant 12 bits of PC. It is possible for two different branch instructions to have the same 12 LSB bits but they may have two different destinations.

We need to make sure that the entry in the BTB clearly indicates to which branch the data belongs.
CSCE 5610: Computer Architecture

Moola multicore cache simulator

- Early to mid 20th century American slang for money (cash, cache)
- Developed by Charles Shelor, similarities to Dinero IV
- Highly configurable trace based cache simulation for multicore
- Single or multiple input trace files, 1 to 32 processor cores
- Includes simulated timing
- Models delays due to cache and memory access conflicts
- Used as a cache filter to indicate memory accesses (AMD and UNT)
- C source code runs on most Unix derivatives (MacOS, RedHat, Ubuntu, Centos)

- Contact me if you want source code to implement 5610 project: implement cache coherency, study multicore memory access congestion, etc. Charles.shelor@gmail.com

Moola Multicore Cache Simulator Results