Reminder: Homeworls

- Homework 3
  - Due Feb 27 (today)
  - 3 questions
    - LC-3b microcode
    - Adding REP MOVS to LC-3b
    - Pipelining
Reminder: Lab Assignments

- Lab Assignment 3
  - Due March 2 – Start early.
  - Individual assignment
    - No collaboration; please respect the honor code
  - Extra credit
    - Early check off: 5%
    - Fastest three designs: 5% + prizes
Don’t Forget: Feedback Sheet for Us

- Will be online today
- **Due Wednesday, February 29**

- Please answer the questions thoroughly
- I would like your honest feedback

- Turn in the feedback sheet
  - during lecture on Wednesday, February 29
  - online via email to 447-instructors

- Can turn in anonymously
Midterm I Next Wednesday

- March 7, in class (12:30-2:20pm sharp)
- Closed book, closed notes
- Can bring a single 8.5x11” cheat sheet
Readings for Today

- P&H Chapter 4.9-4.11
  - More advanced pipelining
  - Interrupts and exception handling
  - Out-of-order and superscalar execution concepts
CALCM Seminar Wednesday 4-5pm

- Feb 29, 2012, Wednesday
- Hamerschlag Hall D-210

- Cosmic Rays Don't Strike Twice: Understanding the Nature of DRAM Errors and the Implications for System Design
- Ioan Stefanovici, Toronto
Review: Issues in Pipeline Design

- **Balancing work in pipeline stages**
  - How many stages and what is done in each stage

- **Keeping the pipeline correct, moving, and full in the presence of events that disrupt pipeline flow**
  - Handling dependences
    - Data
    - Control
  - Handling resource contention
  - Handling long-latency (multi-cycle) operations

- **Handling exceptions, interrupts**

- **Advanced: Improving pipeline throughput**
  - Minimizing stalls, minimizing CPI, minimizing cycle time
Review of Last Lecture

- Control dependence handling in pipelined machines
  - Delayed branching
  - Fine-grained multithreading
  - Branch prediction
    - Compile time (static)
      - Always NT, Always T, Backward T Forward NT, Profile based
    - Run time (dynamic)
      - Last time predictor
      - Hysteresis: 2BC predictor
      - Global branch correlation → Two-level global predictor
      - Local branch correlation → Two-level local predictor
Today

- Control dependence handling in pipelined machines
  - Wrap up branch prediction
  - Predicated execution
  - Multipath execution

- Precise exceptions/interrupts

- Out-of-order execution basics (if we get to it)
Assume a 5-wide superscalar pipeline with 20-cycle branch resolution latency

How long does it take to fetch 500 instructions?

- Assume no fetch breaks and 1 out of 5 instructions is a branch
- 100% accuracy
  - 100 cycles (all instructions fetched on the correct path)
  - No wasted work
- 99% accuracy
  - 100 (correct path) + 20 (wrong path) = 120 cycles
  - 20% extra instructions fetched
- 98% accuracy
  - 100 (correct path) + 20 * 2 (wrong path) = 140 cycles
  - 40% extra instructions fetched
- 95% accuracy
  - 100 (correct path) + 20 * 5 (wrong path) = 200 cycles
  - 100% extra instructions fetched
Review: Can We Do Better?

- Last-time and 2BC predictors exploit “last-time” predictability

- Realization 1: A branch’s outcome can be correlated with other branches’ outcomes
  - Global branch correlation

- Realization 2: A branch’s outcome can be correlated with past outcomes of the same branch (other than the outcome of the branch “last-time” it was executed)
  - Local branch correlation
Review: Global Branch Correlation (I)

- Recently executed branch outcomes in the execution path is correlated with the outcome of the next branch

  ```java
  if (cond1)
  ...
  if (cond1 AND cond2)
  ```

- If first branch not taken, second also not taken

  ```java
  branch Y: if (cond1) a = 2;
  ...
  branch X: if (a == 0)
  ```

- If first branch taken, second definitely not taken
branch Y: if (cond1)
...
branch Z: if (cond2)
...
branch X: if (cond1 AND cond2)

- If Y and Z both taken, then X also taken
- If Y or Z not taken, then X also not taken
Review: Global Branch Correlation (III)

- \text{Eqntott, SPEC92}

```c
if (aa==2) ;; B1
aa=0;
if (bb==2) ;; B2
bb=0;
if (aa!=bb) {
    ;; B3
    ....
}
```

If B1 is not taken (i.e. \(aa==0\)) and B2 is not taken (i.e. \(bb=0\)) then B3 is certainly taken
Review: Capturing Global Branch Correlation

- Idea: Associate branch outcomes with “global T/NT history” of all branches
- Make a prediction is based on the outcome of the branch the last time the same global branch history was encountered

- Implementation:
  - Keep track of the “global T/NT history” of all branches in a register → Global History Register (GHR)
  - Use GHR to index into a table of that recorded the outcome that was seen for that GHR value in the recent past → Pattern History Table (table of 2-bit counters)

- Global history/branch predictor
- Uses two levels of history (GHR + history at that GHR)
Review: Two Level Global Branch Prediction

- First level: Global branch history register (N bits)
  - The direction of last N branches
- Second level: Table of saturating counters for each history entry
  - The direction the branch took the last time the same history was seen

Review: How Does the Global Predictor Work?

```plaintext
for (i=0; i<100; i++)
    for (j=0; j<3; j++)
```

After the initial startup time, the conditional branches have the following behavior, assuming GR is shifted to the left:

<table>
<thead>
<tr>
<th>test</th>
<th>value</th>
<th>GR</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>j&lt;3</td>
<td>j=1</td>
<td>1101</td>
<td>taken</td>
</tr>
<tr>
<td>j&lt;3</td>
<td>j=2</td>
<td>1011</td>
<td>taken</td>
</tr>
<tr>
<td>j&lt;3</td>
<td>j=3</td>
<td>0111</td>
<td>not taken</td>
</tr>
<tr>
<td>i&lt;100</td>
<td>j=3</td>
<td>1110</td>
<td>usually taken</td>
</tr>
</tbody>
</table>

Review: Intel Pentium Pro Branch Predictor

- 4-bit global history register
- Multiple pattern history tables (of 2 bit counters)
  - Which pattern history table to use is determined by lower order bits of the branch address
Improving Global Predictor Accuracy

- Idea: Add more context information to the global predictor to take into account which branch is being predicted
  - **Gshare predictor**: GHR hashed with the Branch PC
  - + More context information
  - + Better utilization of PHT
  - -- Increases access latency

One-Level Branch Predictor

Direction predictor (2-bit counters)

Program Counter
Address of the current branch

Cache of Target Addresses (BTB: Branch Target Buffer)

Next Fetch Address

PC + inst size
hit?
taken?
Two-Level Global History Predictor

Which direction earlier branches went

Global branch history

Program Counter

Address of the current branch

Direction predictor (2-bit counters)

taken?

taken?

PC + inst size

hit?

Next Fetch Address

target address

Cache of Target Addresses (BTB: Branch Target Buffer)
Two-Level Gshare Predictor

Direction predictor (2-bit counters)

Which direction earlier branches went

Global branch history

Program Counter

Address of the current branch

XOR

taken?

PC + inst size

hit?

target address

Cache of Target Addresses (BTB: Branch Target Buffer)

Next Fetch Address
Can We Do Better?

- Last-time and 2BC predictors exploit “last-time” predictability

- Realization 1: A branch’s outcome can be correlated with other branches’ outcomes
  - Global branch correlation

- Realization 2: A branch’s outcome can be correlated with past outcomes of the same branch (other than the outcome of the branch “last-time” it was executed)
  - Local branch correlation
Local Branch Correlation

for (i=1; i<=4; i++) {
}

If the loop test is done at the end of the body, the corresponding branch will execute the pattern \((1110)^n\), where 1 and 0 represent taken and not taken respectively, and \(n\) is the number of times the loop is executed. Clearly, if we knew the direction this branch had gone on the previous three executions, then we could always be able to predict the next branch direction.

Capturing Local Branch Correlation

- Idea: **Have a per-branch history register**
  - Associate the predicted outcome of a branch with “T/NT history” of the same branch
- Make a prediction is based on the outcome of the branch the last time the same local branch history was encountered

- Called the local history/branch predictor
- Uses two levels of history (Per-branch history register + history at that history register value)
Two Level Local Branch Prediction

- First level: A set of local history registers (N bits each)
  - Select the history register based on the PC of the branch
- Second level: Table of saturating counters for each history entry
  - The direction the branch took the last time the same history was seen

Two-Level Local History Predictor

Which directions earlier instances of *this branch* went

Direction predictor (2-bit counters)

Program Counter

Address of the current branch

Cache of Target Addresses (BTB: Branch Target Buffer)

Next Fetch Address

PC + inst size

hit?

taken?
Hybrid Branch Predictors

- **Idea:** Use more than one type of predictor (i.e., multiple algorithms) and select the “best” prediction
  - E.g., hybrid of 2-bit counters and global predictor

- **Advantages:**
  + Better accuracy: different predictors are better for different branches
  + Reduced *warmup* time (faster-warmup predictor used until the slower-warmup predictor warms up)

- **Disadvantages:**
  -- Need “meta-predictor” or “selector”
  -- Longer access latency

Minimum branch penalty: 7 cycles
Typical branch penalty: 11+ cycles
48K bits of target addresses stored in I-cache
Predictor tables are reset on a context switch

Branch Prediction Accuracy (Example)

- Bimodal: table of 2bc indexed by branch address

Figure 13: Combined Predictor Performance by Benchmark
Predication (Predicated Execution)

- **Idea:** Compiler converts control dependency into a data dependency → branch is eliminated
  - Each instruction has a predicate bit set based on the predicate computation
  - Only instructions with TRUE predicates are committed (others turned into NOPs)

(normal branch code) (predicated code)

```c
if (cond) {
  b = 0;
}
else {
  b = 1;
}
```

```c
A

T

N

C

B

D

p1 = (cond)
branch p1, TARGET

mov b, 1
jmp JOIN

TARGET:
  mov b, 0

add x, b, 1
```

```c
A

B

C

D

p1 = (cond)
(!p1) mov b, 1
(p1) mov b, 0
add x, b, 1
```
Conditional Move Operations

- Very limited form of predicated execution

- CMOV R1 ← R2
  - $R1 = (\text{ConditionCode} == \text{true}) ? R2 : R1$
  - Employed in most modern ISAs (x86, Alpha)
Predicated execution can be high performance and energy-efficient

Predicated Execution
Fetch Decode Rename Schedule RegisterRead Execute

Branch Prediction
Fetch Decode Rename Schedule RegisterRead Execute

Pipeline flush!!
Advantages:
+ Eliminates mispredictions for hard-to-predict branches
  + No need for branch prediction for some branches
  + Good if misprediction cost > useless work due to predication
+ Enables code optimizations hindered by the control dependency
  + Can move instructions more freely within predicated code

Disadvantages:
-- Causes useless work for branches that are easy to predict
  -- Reduces performance if misprediction cost < useless work
  -- Adaptivity: Static predication is not adaptive to run-time branch behavior. Branch behavior changes based on input set, phase, control-flow path.
-- Additional hardware and ISA support
-- Cannot eliminate all hard to predict branches
  -- Loop branches?
Each instruction can be separately predicated
64 one-bit predicate registers
each instruction carries a 6-bit predicate field
An instruction is effectively a NOP if its predicate is false
Conditional Execution in ARM ISA

- Almost all ARM instructions can include an optional condition code.

- An instruction with a condition code is only executed if the condition code flags in the CPSR meet the specified condition.
## Conditional Execution in ARM ISA

### Instruction Type

- **Data processing / PSR Transfer**
- **Multiply**
- **Long Multiply** (v3M / v4 only)
- **Swap**
- **Load/Store Byte/Word**
- **Load/Store Multiple**
- **Halfword transfer: Immediate offset (v4 only)**
- **Halfword transfer: Register offset (v4 only)**
- **Branch**
- **Branch Exchange** (v4T only)
- **Coprocessor data transfer**
- **Coprocessor data operation**
- **Coprocessor register transfer**
- **Software interrupt**

### Instruction Format

<table>
<thead>
<tr>
<th>Cond</th>
<th>Opcode</th>
<th>S</th>
<th>Rn</th>
<th>Rd</th>
<th>Operand1</th>
<th>Operand2</th>
</tr>
</thead>
<tbody>
<tr>
<td>001</td>
<td>00000000A</td>
<td>S</td>
<td>Rd</td>
<td>Rn</td>
<td>Rs 1001</td>
<td>Rm</td>
</tr>
<tr>
<td>0001</td>
<td>00001UAS</td>
<td>RdHi RdLo</td>
<td>Rs</td>
<td>1001</td>
<td>Rm</td>
<td></td>
</tr>
<tr>
<td>0010</td>
<td>00010BO0</td>
<td>Rn</td>
<td>Rd</td>
<td>000101</td>
<td>Rm</td>
<td></td>
</tr>
<tr>
<td>0111</td>
<td>IPU BWL</td>
<td>Rn</td>
<td>Rd</td>
<td>Offset</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1001</td>
<td>PUSWL</td>
<td>Rn</td>
<td>Register List</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0001</td>
<td>PUIWL</td>
<td>Rn</td>
<td>Rd</td>
<td>Offset1 1SH1 Offset2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0001</td>
<td>PUOWL</td>
<td>Rn</td>
<td>Rd</td>
<td>00001SH1 Rm</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1011</td>
<td>1011L</td>
<td>Offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0001</td>
<td>001011111111111111110001</td>
<td>Rn</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1101</td>
<td>PUNWL</td>
<td>Rn</td>
<td>CRd CPNum Offset</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1110</td>
<td>OP1 CRn</td>
<td>CRd CPNum Op2 0 CRm</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1110</td>
<td>OP1 L CRn</td>
<td>Rd CPNum Op2 1 CRm</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1111</td>
<td>SWI Number</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

38
Conditional Execution in ARM ISA

- 0000 = EQ - Z set (equal)
- 0001 = NE - Z clear (not equal)
- 0010 = HS / CS - C set (unsigned higher or same)
- 0011 = LO / CC - C clear (unsigned lower)
- 0100 = MI - N set (negative)
- 0101 = PL - N clear (positive or zero)
- 0110 = VS - V set (overflow)
- 0111 = VC - V clear (no overflow)
- 1000 = HI - C set and Z clear (unsigned higher)
- 1001 = LS - C clear or Z (set unsigned lower or same)
- 1010 = GE - N set and V set, or N clear and V clear (≥ or =)
- 1011 = LT - N set and V clear, or N clear and V set (>)
- 1100 = GT - Z clear, and either N set and V set, or N clear and V set (>)
- 1101 = LE - Z set, or N set and V clear, or N clear and V set (<, or =)
- 1110 = AL - always
- 1111 = NV - reserved.
* To execute an instruction conditionally, simply postfix it with the appropriate condition:
  
  - For example an add instruction takes the form:
    
    \[
    \text{ADD \ r0,\ r1,\ r2} \quad ; \ r0 = r1 + r2 \quad \text{(ADDAL)}
    \]
  
  - To execute this only if the zero flag is set:
    
    \[
    \text{ADDEQ \ r0,\ r1,\ r2} \quad ; \ \text{If zero flag set then...}
    \]
    
    \[
    \ldots \ r0 = r1 + r2
    \]
    
* By default, data processing operations do not affect the condition flags (apart from the comparisons where this is the only effect). To cause the condition flags to be updated, the S bit of the instruction needs to be set by postfixing the instruction (and any condition code) with an “S”.
  
  - For example to add two numbers and set the condition flags:
    
    \[
    \text{ADDS \ r0,\ r1,\ r2} \quad ; \ r0 = r1 + r2
    \]
    
    \[
    \ldots \ \text{and set flags}
    \]
Conditional Execution in ARM ISA

* Convert the GCD algorithm given in this flowchart into
  1) “Normal” assembler, where only branches can be conditional.
  2) ARM assembler, where all instructions are conditional, thus improving code density.

* The only instructions you need are CMP, B and SUB.
Conditional Execution in ARM ISA

"Normal" Assembler

```
gcd   cmp r0, r1 ;reached the end?
    beq stop
    blt less ;if r0 > r1
    sub r0, r0, r1 ;subtract r1 from r0
    bal gcd
less   sub r1, r1, r0 ;subtract r0 from r1
    bal gcd
stop
```

ARM Conditional Assembler

```
gcd   cmp r0, r1 ;if r0 > r1
subgt r0, r0, r1 ;subtract r1 from r0
sublt r1, r1, r0 ;else subtract r0 from r1
bne  gcd ;reached the end?
```
Multi-Path Execution

- **Idea:** *Execute both paths after a conditional branch*
  - For a hard-to-predict branch: Use dynamic confidence estimation

- **Advantages:**
  + Improves performance if misprediction cost > useless work
  + No ISA change needed

- **Disadvantages:**
  -- What happens when the machine encounters another hard-to-predict branch? Execute both paths again?
    -- Paths followed quickly become exponential
  -- Each followed path requires its own registers, PC, GHR
  -- Wasted work (and reduced performance) if paths merge
Dual-Path Execution versus Predication

Hard to predict

Dual-path

Predicated Execution

path 1

path 2

path 1

path 2
Call and Return Prediction

- **Direct calls are easy to predict**
  - Always taken, single target
  - Call marked in BTB, target predicted by BTB

- **Returns are indirect branches**
  - A function can be called from many points in code
  - A return instruction can have many target addresses
    - Next instruction after each call point for the same function
  - **Observation:** Usually a return matches a call
  - **Idea:** Use a stack to predict return addresses (Return Address Stack)
    - A fetched call: pushes the return (next instruction) address on the stack
    - A fetched return: pops the stack and uses the address as its predicted target
    - Accurate most of the time: 8-entry stack $\Rightarrow > 95\%$ accuracy
Indirect Branch Prediction (I)

- Register-indirect branches have multiple targets

![Diagram of indirect branch prediction]

- Used to implement
  - Switch-case statements
  - Virtual function calls
  - Jump tables (of function pointers)
  - Interface calls
Indirect Branch Prediction (II)

- No direction prediction needed
- Idea 1: Predict the last resolved target as the next fetch address
  + Simple: Use the BTB to store the target address
  -- Inaccurate: 50% accuracy (empirical). Many indirect branches switch between different targets

- Idea 2: Use history based target prediction
  - E.g., Index the BTB with GHR XORed with Indirect Branch PC
  + More accurate
  -- An indirect branch maps to (too) many entries in BTB
    -- Conflict misses with other branches (direct or indirect)
    -- Inefficient use of space if branch has few target addresses
Issues in Branch Prediction (I)

- Need to identify a branch before it is fetched

- How do we do this?
  - BTB hit → indicates that the fetched instruction is a branch
  - BTB entry contains the “type” of the branch

- What if no BTB?
  - Bubble in the pipeline until target address is computed
  - E.g., IBM POWER4
Issues in Branch Prediction (II)

- **Latency:** Prediction is latency critical
  - Need to generate next fetch address for the next cycle
  - Bigger, more complex predictors are more accurate but slower
Complications in Superscalar Processors

- “Superscalar” processors
  - attempt to execute more than 1 instruction-per-cycle
  - must fetch multiple instructions per cycle

- Consider a 2-way superscalar fetch scenario
  (case 1) Both insts are not taken control flow inst
    - nPC = PC + 8
  (case 2) One of the insts is a taken control flow inst
    - nPC = predicted target addr
    - *NOTE* both instructions could be control-flow; prediction based on the first one predicted taken
    - If the 1\textsuperscript{st} instruction is the predicted taken branch
      $\rightarrow$ nullify 2\textsuperscript{nd} instruction fetched
Multiple Instruction Fetch: Concepts

- Fetch 1 inst/cycle
  - Downside:
    - Flynn’s bottleneck
      - If you fetch 1 inst/cycle, you cannot finish >1 inst/cycle

- Fetch 4 inst/cycle

Two major approaches

1) VLIW
   - Compiler decides what insts can be executed in parallel
   - Simple hardware

2) Superscalar
   - Hardware detects dependencies between instructions that are fetched in the same cycle
Pipelining and Precise Exceptions: Preserving Sequential Semantics
Multi-Cycle Execution

- Not all instructions take the same amount of time for “execution”

- Idea: Have multiple different functional units that take different number of cycles
  - Can be pipelined or not pipelined
  - Can let independent instructions to start execution on a different functional unit before a previous long-latency instruction finishes execution
Issues in Pipelining: Multi-Cycle Execute

- Instructions can take different number of cycles in EXECUTE stage
  - Integer ADD versus FP MULtiply

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>FMUL R4 ← R1, R2</td>
<td>F D E E E E E E E E E W</td>
</tr>
<tr>
<td>ADD R3 ← R1, R2</td>
<td>F D E W</td>
</tr>
<tr>
<td></td>
<td>F D E W</td>
</tr>
<tr>
<td></td>
<td>F D E W</td>
</tr>
<tr>
<td></td>
<td>F D E W</td>
</tr>
<tr>
<td>FMUL R2 ← R5, R6</td>
<td>F D E E E E E E E E E E W</td>
</tr>
<tr>
<td>ADD R4 ← R5, R6</td>
<td>F D E W</td>
</tr>
<tr>
<td></td>
<td>F D E W</td>
</tr>
<tr>
<td></td>
<td>F D E W</td>
</tr>
</tbody>
</table>

- What is wrong with this picture?
  - What if FMUL incurs an exception?
  - Sequential semantics of the ISA NOT preserved!
Handling Exceptions in Pipelining

- Exceptions versus interrupts

- Cause
  - Exceptions: internal to the running thread
  - Interrupts: external to the running thread

- When to Handle
  - Exceptions: when detected (and known to be non-speculative)
  - Interrupts: when convenient
    - Except for very high priority ones
      - Power failure
      - Machine check

- Priority: process (exception), depends (interrupt)

- Handling Context: process (exception), system (interrupt)
We did not cover the following slides in lecture. These are for your preparation for the next lecture.
Precise Exceptions/Interrupts

- The architectural state should be consistent when the exception/interrupt is ready to be handled

1. All previous instructions should be completely retired.

2. No later instruction should be retired.

Retire = commit = finish execution and update arch. state
Why Do We Want Precise Exceptions?

- Semantics of the von Neumann model ISA specifies it
- Aids software debugging
- Enables (easy) recovery from exceptions, e.g., page faults
- Enables (easily) restartable processes
- Enables traps into software (e.g., software implemented opcodes)