Reminder: Homeworks

- Homework 3
  - Due Feb 27
  - Out
  - 3 questions
    - LC-3b microcode
    - Adding REP MOVS to LC-3b
    - Pipelining
Homework 2 Grade Distribution

Number of Students

Grade

<table>
<thead>
<tr>
<th>Grade</th>
<th>90</th>
<th>100</th>
<th>110</th>
<th>120</th>
<th>130</th>
<th>140</th>
<th>150</th>
<th>160</th>
<th>165</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>15</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>

Average: 149.9245
Median: 152
Max: 164
Min: 90

Max Possible Points: 165
Total number of students: 53
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
Lab 2 Grade Distribution

- Average: 140
- Median: 144
- Max: 150
- Min: 81
- Max Possible Score: 150
- Total number of students: 48
Prize for Lab 1

- Fastest functional simulator
  - Kee Young Lee
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 Coming Up

- March 7, Wednesday
Lab Note: Beware Implied Latches!

- Is this combinational or stateful?

```verilog
input [1:0] i;
reg a;
always @(*) begin
    case (i)
        2'b00: a = 0;
        2'b01: a = 1;
        2'b10: a = 0;
    endcase
end
```

- If `i = 2'b11` in a cycle, `a` will keep its previous value (**stateful!**)
- In general, if a reg is not assigned on every path, it **implies a latch**
- We did not specify this for lab 2, but in **Lab 3**, you may not imply latches (Check XST output for warnings about this)
Lab Note: Beware Implied Latches!

- How would you eliminate the implied latch?

```verilog
input [1:0] i;
reg a;
always @(*) begin
    case (i)
        2'b00: a = 0;
        2'b01: a = 1;
        2'b10: a = 0;
        2'b11: a = 1; // add this case
    endcase
end
```
Poll: Discussion Sections

- Some discussion sections are not well-attended (especially Tuesday 10:30am)

- Would you prefer a different time?

- Are discussion sections useful?
Readings for Next Lecture

- 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 22, 2012, Wednesday
- Hamerschlag Hall D-210

- Stochastic Computing: Embracing Errors in Architecture and Design of Hardware and Software
- Prof. Rakesh Kumar, University of Illinois

- Hardware is allowed to produce errors that are exposed to the highest layers of software
- Hardware and software are optimized to maximize power savings afforded by relaxed correctness
Review of Last Lecture

- Data dependence handling in pipelined machines
- Control dependence handling in pipelined machines
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
Stall Fetch Until Next PC is Available: Good Idea?

This is the case with non-control-flow instructions!
Doing Better than Stalling Fetch …

- Rather than waiting for true-dependence on PC to resolve, just guess nextPC = PC+4 to keep fetching every cycle
  - Is this a good guess?
  - What do you lose if you guessed incorrectly?

- ~20% of the instruction mix is control flow
  - ~50 % of “forward” control flow (i.e., if-then-else) is taken
  - ~90% of “backward” control flow (i.e., loop back) is taken
    - Over all, typically ~70% taken and ~30% not taken
      - [Lee and Smith, 1984]

- Expect “nextPC = PC+4” ~86% of the time, but what about the remaining 14%?
Control Dependence Handling
How to Handle Control Dependences

- Critical to keep the pipeline full with correct sequence of dynamic instructions. Potential solutions:

  - If the instruction is a control-flow instruction:
    - Stall the pipeline until we know the next fetch address
    - Guess the next fetch address. How?
  - Employ delayed branching (branch delay slot)
  - Do something else (fine-grained multithreading)
  - Eliminate control-flow instructions (predicated execution)
  - Fetch from both possible paths (if you know the addresses of both possible paths) (multipath execution)
Delayed Branching (I)

- Change the semantics of a branch instruction
  - Branch after N instructions
  - Branch after N cycles
- Idea: Delay the execution of a branch. N instructions (delay slots) that come after the branch are always executed regardless of branch direction.

- Problem: How do you find instructions to fill the delay slots?
  - Branch must be independent of delay slot instructions

- Unconditional branch: Easier to find instructions to fill the delay slot
- Conditional branch: Condition computation should not depend on instructions in delay slots → difficult to fill the delay slot
Delayed Branching (II)

Normal code:  

Timeline:  

Delayed branch code:  

Timeline:

Normal code:  

Timeline:  

Delayed branch code:  

Timeline:

X:  

G  

6 cycles  

X:  

G  

5 cycles
Fancy Delayed Branching (III)

- Delayed branch with squashing
  - In SPARC
  - If the branch falls through (not taken), the delay slot instruction is not executed
  - Why could this help?

Normal code:              Delayed branch code:              Delayed branch w/ squashing:

\[
\begin{array}{c}
X: \\
\begin{array}{c}
A \\
B \\
C \\
BC X \\
D \\
E \\
\end{array}
\end{array}
\]

\[
\begin{array}{c}
X: \\
\begin{array}{c}
A \\
B \\
C \\
BC X \\
NOP \\
D \\
E \\
\end{array}
\end{array}
\]

\[
\begin{array}{c}
X: \\
\begin{array}{c}
A \\
B \\
C \\
BC X \\
A \\
D \\
E \\
\end{array}
\end{array}
\]
Delayed Branching (IV)

- Advantages:
  + Keeps the pipeline full with useful instructions in a simple way assuming
    1. Number of delay slots == number of instructions to keep the pipeline full before the branch resolves
    2. All delay slots can be filled with useful instructions

- Disadvantages:
  -- Not easy to fill the delay slots (even with a 2-stage pipeline)
    1. Number of delay slots increases with pipeline depth, superscalar execution width
    2. Number of delay slots should be variable with variable latency operations. Why?
  -- Ties ISA semantics to hardware implementation
    -- SPARC, MIPS, HP-PA: 1 delay slot
    -- What if pipeline implementation changes with the next design?
An Aside: Filling the Delay Slot

reordering data independent (RAW, WAW, WAR) instructions does not change program

within same basic block

a new instruction added to not-taken path??

a new instruction added to taken??

Safe?

[Based on original figure from P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]
Fine-Grained Multithreading

- **Idea:** Hardware has multiple thread contexts. Each cycle, fetch engine fetches from a different thread.
  - By the time the fetched branch/instruction resolves, there is no need to fetch another instruction from the same thread.
  - Branch/instruction resolution latency overlapped with execution of other threads’ instructions.

+ No logic needed for handling control and data dependences within a thread.
-- Single thread performance suffers.
-- Extra logic for keeping thread contexts.
-- Does not overlap latency if not enough threads to cover the whole pipeline.
Branch Prediction: Guess the Next Instruction to Fetch

Branch prediction helps in predicting the next instruction to fetch, reducing the number of cycles needed for execution. The diagram shows the pipeline stages: I-$, DEC, RF, D-$, and WB. The flow chart illustrates the process:

- **I-$**: Instruction Stage
- **DEC**: Decode Stage
- **RF**: Register File Stage
- **D-$**: Data Stage
- **WB**: Write Back Stage

The diagram also highlights the branch prediction mechanism, where the PC (Program Counter) is used to predict the next instruction. The branch prediction outcomes are visualized with different cycles: 12 cycles and 8 cycles.
Misprediction Penalty

LD R0, MEM[R2]
LD R2, MEM[R2]
BR
ZERO
0x0001
LD R1, MEM[R0]
ADD R2, R2, #1
ADD R3, R2, #1
MUL R1, R2, R3
LD R2, MEM[R2]
LD R0, MEM[R2]
Branch Prediction

- Processors are pipelined to increase concurrency
- How do we keep the pipeline full in the presence of branches?
  - Guess the next instruction when a branch is fetched
  - Requires guessing the direction and target of a branch

![Pipeline Diagram]

Branch condition, TARGET

What to fetch next?
- Fetch from the correct target

Processors are pipelined to increase concurrency
How do we keep the pipeline full in the presence of branches?
- Guess the next instruction when a branch is fetched
- Requires guessing the direction and target of a branch

Pipeline

Fetch  Decode  Rename  Schedule  RegisterRead  Execute

B3  [  ]  [  ]  [  ]  [ ]  F  E  D  B1  A

Target Misprediction Detected! Flush the pipeline
Branch Prediction: Always PC+4

When a branch resolves
- branch target (Inst$_k$) is fetched
- all instructions fetched since inst$_h$ (so called “wrong-path” instructions) must be flushed
Pipeline Flush on a Misprediction

Inst_\text{h} is a branch
Performance Analysis

- correct guess ⇒ no penalty ~86% of the time
- incorrect guess ⇒ 2 bubbles
- Assume
  - no data hazards
  - 20% control flow instructions
  - 70% of control flow instructions are taken
  - CPI = \[ 1 + (0.20 \times 0.7) \times 2 \] = 1.28
  - probability of penalty for a wrong guess
  - Can we reduce either of the two penalty terms?
Reducing Branch Misprediction Penalty

- Resolve branch condition and target address early. Downside?

CPI = \[ 1 + (0.2 \times 0.7) \times 1 \] = 1.14

[Based on original figure from P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]
Branch Prediction (Enhanced)

- **Idea:** Predict the next fetch address (to be used in the next cycle)

- Requires three things to be predicted:
  - Whether the fetched instruction is a branch
  - (Conditional) branch direction
  - Branch target address (if taken)

- Target addresses remain the same for conditional direct branches across dynamic instances
  - **Idea:** Store the target address from previous instance and access it with the PC
  - Called Branch Target Buffer (BTB) or Branch Target Address Cache
Fetch Stage with BTB and Direction Prediction

Direction predictor (2-bit counters)

Cache of Target Addresses (BTB: Branch Target Buffer)

Always taken CPI = [ 1 + (0.20*0.3) * 2 ] = 1.12 (70% of branches taken)
More Sophisticated Branch Direction Prediction

- Direction predictor (2-bit counters)
- Cache of Target Addresses (BTB: Branch Target Buffer)
- Global branch history
- Program Counter
- Address of the current branch
- Which direction earlier branches went
- Next Fetch Address

- XOR
- taken?
- hit?
- PC + inst size

Cache of Target Addresses (BTB: Branch Target Buffer)
Simple Branch Direction Prediction Schemes

- Compile time (static)
  - Always not taken
  - Always taken
  - BTFN (Backward taken, forward not taken)
  - Profile based (likely direction)

- Run time (dynamic)
  - Last time prediction (single-bit)
Static Branch Prediction (I)

- Always not-taken
  - Simple to implement: no need for BTB, no direction prediction
  - Low accuracy: ~40%
  - Compiler can layout code such that the likely path is the “not-taken” path

- Always taken
  - No direction prediction
  - Better accuracy: ~60%
    - Backward branches (i.e. loop branches) are usually taken
    - Backward branch: target address lower than branch PC

- Backward taken, forward not taken (BTFN)
  - Predict backward (loop) branches as taken, others not-taken
Static Branch Prediction (II)

- Profile-based
  - Idea: Compiler determines likely direction for each branch using profile run. Encodes that direction as a hint bit in the branch instruction format.

+ Per branch prediction (more accurate than schemes in previous slide)
  -- Requires hint bits in the branch instruction format
  -- Accuracy depends on dynamic branch behavior:
     TTTTTTTTTTNNNNNNNNNNN → 50% accuracy
     TNTNTNTNTNTNTNTNTNTNTNTNTNTN → 50% accuracy
  -- Accuracy depends on the representativeness of profile input set
Dynamic Branch Prediction

- **Idea**: Predict branches based on dynamic information (collected at run-time)

- **Advantages**
  - + No need for profiling: input set representativeness problem goes away
  - + Prediction based on history of the execution of branches
    - + It can adapt to dynamic changes in branch behavior

- **Disadvantages**
  - -- More complex (requires additional hardware)
Last Time Predictor

- Last time predictor
  - Single bit per branch (stored in BTB)
  - Indicates which direction branch went last time it executed
    TTTTTTTTTTTNNNNNNNNNNN → 90% accuracy

- Always mispredicts the last iteration and the first iteration of a loop branch
  - Accuracy for a loop with N iterations = (N-2)/N

+ Loop branches for loops with large number of iterations

-- Loop branches for loops with small number of iterations
  TNTNTNTNTNTNTNTNTNTNTNTNTTN → 0% accuracy

Last-time predictor CPI = [ 1 + (0.20*0.15) * 2 ] = 1.06 (Assuming 85% accuracy)
The 1-bit BHT (Branch History Table) entry is updated with the correct outcome after each execution of a branch.
State Machine for Last-Time Prediction

Predict not taken

Actually not taken

Actually taken

Predict taken

Actually taken

Actually not taken

Predict not taken

Actually not taken

Actually taken
Improving the Last Time Predictor

- Problem: A last-time predictor changes its prediction from T→NT or NT→T too quickly
  - even though the branch may be mostly taken or mostly not taken

- Solution Idea: Add hysteresis to the predictor so that prediction does not change on a single different outcome
  - Use two bits to track the history of predictions for a branch instead of a single bit
  - Can have 2 states for T or NT instead of 1 state for each

Two-Bit Counter Based Prediction

- Each branch associated with a two-bit counter
- One more bit provides hysteresis
- A strong prediction does not change with one single different outcome

- Accuracy for a loop with N iterations = (N-1)/N
  TNTNTNTNTNTNTNTNTNTNTNTN → 50% accuracy
  (assuming init to weakly taken)

+ Better prediction accuracy
  2BC predictor CPI = [ 1 + (0.20 * 0.10) * 2 ] = 1.04 (90% accuracy)

-- More hardware cost (but counter can be part of a BTB entry)
State Machine for 2-bit Saturating Counter

- Counter using saturating arithmetic
  - There is a symbol for maximum and minimum values
Hysteresis Using a 2-bit Counter

Change prediction after 2 consecutive mistakes
The Branch Problem

- Control flow instructions (branches) are frequent
  - 15-25% of all instructions

- Problem: Next fetch address after a control-flow instruction is not determined after N cycles in a pipelined processor
  - N cycles: (minimum) branch resolution latency
  - Stalling on a branch wastes instruction processing bandwidth (i.e. reduces IPC)
    - N x IW instruction slots are wasted

- How do we keep the pipeline full after a branch?
- Problem: Need to determine the next fetch address when the branch is fetched (to avoid a pipeline bubble)
Importance of The Branch Problem

- 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
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
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
Global Branch Correlation (II)

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
Global Branch Correlation (III)

- 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@B3) and B2 is not taken (i.e. bb=0@B3) then B3 is certainly taken
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)
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

How Does the Global Predictor Work?

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
We did not cover the following slides in lecture. These are for your preparation for the next lecture.
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

- **Global branch history**
- **Program Counter**
- **Address of the current branch**
- **Direction predictor (2-bit counters)**
- **Cache of Target Addresses (BTB: Branch Target Buffer)**

Which direction earlier branches went

PC + inst size

taken?

hit?

target address

Next Fetch Address
Two-Level Gshare Predictor

Which direction earlier branches went

Global branch history

Program Counter

Address of the current branch

Direction predictor (2-bit counters)

taken?

PC + inst size

hit?

Next Fetch Address

Cache of Target Addresses (BTB: Branch Target Buffer)
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

- 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)

PC + inst size

taken?

hit?

target address

Next Fetch Address
Hybrid Branch Predictors

- **Idea:** Use more than one type of predictors (i.e., 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

Alpha 21264 Tournament Predictor

- 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)

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

```
if (cond) {
    p1 = (cond)
    branch p1, TARGET
} else {
    p1 = (!p1)
    mov b, 1
    jmp JOIN
}
```

```
TARGET:
    mov b, 0
    add x, b, 1
```

```
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 = (ConditionCode == 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!!
Predicated Execution (III)

- **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
    - Vectorization with control flow

- **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
    - Complex control flow graphs, function calls, and loop branches
Predicated Execution in Intel Itanium

- 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

---

![Conditional Execution Table](image)
Conditional Execution in ARM ISA

<table>
<thead>
<tr>
<th>Cond</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>EQ - Z set (equal)</td>
</tr>
<tr>
<td>0001</td>
<td>NE - Z clear (not equal)</td>
</tr>
<tr>
<td>0010</td>
<td>HS / CS - C set (unsigned higher or same)</td>
</tr>
<tr>
<td>0011</td>
<td>LO / CC - C clear (unsigned lower)</td>
</tr>
<tr>
<td>0100</td>
<td>MI - N set (negative)</td>
</tr>
<tr>
<td>0101</td>
<td>PL - N clear (positive or zero)</td>
</tr>
<tr>
<td>0110</td>
<td>VS - V set (overflow)</td>
</tr>
<tr>
<td>0111</td>
<td>VC - V clear (no overflow)</td>
</tr>
<tr>
<td>1000</td>
<td>HI - C set and Z clear (unsigned higher)</td>
</tr>
<tr>
<td>1001</td>
<td>LS - C clear or Z (set unsigned lower or same)</td>
</tr>
<tr>
<td>1010</td>
<td>GE - N set and V set, or N clear and V clear (≥ or =)</td>
</tr>
<tr>
<td>1011</td>
<td>LT - N set and V clear, or N clear and V set (&gt; or &lt;)</td>
</tr>
<tr>
<td>1100</td>
<td>GT - Z clear, and either N set and V set, or N clear and V set (&gt; or &lt;)</td>
</tr>
<tr>
<td>1101</td>
<td>LE - Z set, or N set and V clear, or N clear and V set (≤ or =)</td>
</tr>
<tr>
<td>1110</td>
<td>AL - always</td>
</tr>
<tr>
<td>1111</td>
<td>NV - reserved</td>
</tr>
</tbody>
</table>
Conditional Execution in ARM ISA

* To execute an instruction conditionally, simply postfix it with the appropriate condition:
  - For example an add instruction takes the form:
    - ADD r0, r1, r2 ; r0 = r1 + r2 (ADDLAL)
  - To execute this only if the zero flag is set:
    - ADDEQ r0, r1, r2 ; If zero flag set then...
      ; ... 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:
    - ADDS r0, r1, r2 ; r0 = r1 + r2
      ; ... 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 register alias table, PC, GHR
  - -- Wasted work (and reduced performance) if paths merge
Dual-Path Execution versus Predication

Hard to predict

A
C
B
D
E
F

Dual-path

path 1

C
D
E
F

path 2

B
D
E
F

Predicated Execution

path 1

C
D
E
F

path 2

B
D
E
F

CFM

CFM
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

```
A
```

[br.cond TARGET]

```
TARG   A+1
```

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

```
R1 = MEM[R2]
```

branch R1

Conditional (Direct) Branch

Indirect Jump

84
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

![Diagram of branch prediction]

- PC + inst size
- BTB target
- Return Address Stack target
- Indirect Branch Predictor target
- Resolved target from Backend

Next Fetch Address

???
“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\(^{st}\) instruction is the predicted taken branch
  \( \rightarrow \) nullify 2\(^{nd}\) instruction fetched
Multiple Instruction Fetch

- 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 runs
   - Can be executed in parallel
     → Simple hardware

2) Superscalar
   - Hardware detects dependencies
     between instructions that are fetched in the same cycle