Homework 3

- Homework 3
  - Due Feb 25
  - REP MOVS in Microprogrammed LC-3b, Pipelining, Delay Slots, Interlocking, Branch Prediction
Lab Assignment 2

- Due this Friday, Feb 15
- Single-cycle MIPS implementation in Verilog
- All labs are individual assignments
- No collaboration; please respect the honor code
- Do not forget the extra credit portion!
Feedback Sheet

- Due this Friday (Feb 15), in class
- Hard copy

- We would like your honest feedback on the course
Readings for Today


  - More advanced pipelining
  - Interrupt and exception handling
  - Out-of-order and superscalar execution concepts
Last Lecture

- Branch prediction
Today’s Agenda

- Wrap up control dependence handling
- State recovery mechanisms, interrupts, exceptions
Control Dependence Handling
Review: 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** (*branch prediction*)
  - 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*)
Review: 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
Review: Local and Global Branch Prediction

- 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: Hybrid Branch Prediction in Alpha 21264

- 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

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 (branch prediction)
  - 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)
Complex predicates are converted into multiple branches

- if ((a == b) && (c < d) && (a > 5000)) { ... }
  3 conditional branches

Problem: This increases the number of control dependencies

Idea: **Combine predicate operations to feed a single branch instruction**

- Predicates stored and operated on using *condition registers*
- A *single branch* checks the value of the combined predicate

+ Fewer branches in code → fewer mipredictions/stalls

-- Possibly unnecessary work

  -- If the first predicate is false, no need to compute other predicates

- Condition registers exist in IBM RS6000 and the POWER architecture
**Predication (Predicated Execution)**

- **Idea:** Compiler converts control dependence into data dependence $\rightarrow$ 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)  
```
if (cond) {
    b = 0;
} else {
    b = 1;
}
```

(predicated code)  
```
if (cond) {
    p1 = (cond)
    branch p1, TARGET
    mov b, 1
    jmp JOIN
} else {
    p1 = (!p1)
    mov b, 0
    add x, b, 1
    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)
Review: CMOV Operation

- Suppose we had a Conditional Move instruction...
  - CMOV condition, R1 ← R2
  - R1 = (condition == true) ? R2 : R1
  - Employed in most modern ISAs (x86, Alpha)

- Code example with branches vs. CMOVs
  if (a == 5) {b = 4;} else {b = 3;}

  CMPEQ condition, a, 5;
  CMOV condition, b ← 4;
  CMOV !condition, b ← 3;
Predicated execution can be high performance and energy-efficient.
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

**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?
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

### Table

<table>
<thead>
<tr>
<th>Cond</th>
<th>Opcode</th>
<th>S</th>
<th>Rn</th>
<th>Rd</th>
<th>Operand2</th>
<th>Rs</th>
<th>1001</th>
<th>Rm</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000</td>
<td>A S</td>
<td></td>
<td>Rd</td>
<td>Rn</td>
<td></td>
<td>Rs</td>
<td>1001</td>
<td>Rm</td>
</tr>
<tr>
<td>00000101</td>
<td>U A S</td>
<td></td>
<td>RdHi</td>
<td>RdLo</td>
<td></td>
<td>Rs</td>
<td>1001</td>
<td>Rm</td>
</tr>
<tr>
<td>00010100</td>
<td>B 00</td>
<td></td>
<td>Rn</td>
<td>Rd</td>
<td>00001001</td>
<td>Rm</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01110101</td>
<td>PUBW L</td>
<td></td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td>Offset</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10010101</td>
<td>PSWL</td>
<td></td>
<td>Rn</td>
<td></td>
<td>Register List</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00010111</td>
<td>PUWL</td>
<td></td>
<td>Rn</td>
<td>Rd</td>
<td>Offset1</td>
<td>1 S H</td>
<td>1 Offset2</td>
<td></td>
</tr>
<tr>
<td>00010111</td>
<td>PUWL</td>
<td></td>
<td>Rn</td>
<td>Rd</td>
<td>00001001</td>
<td>S H</td>
<td>1 Rm</td>
<td></td>
</tr>
<tr>
<td>10111100</td>
<td>0010</td>
<td>1111</td>
<td>1111</td>
<td>1111</td>
<td>10001</td>
<td>Rn</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11010110</td>
<td>PUNWL</td>
<td></td>
<td>Rn</td>
<td>CRd</td>
<td>CPNum</td>
<td>Offset</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11110101</td>
<td>Op1L</td>
<td></td>
<td>CRn</td>
<td>CRd</td>
<td>CPNum</td>
<td>Op2</td>
<td>0</td>
<td>CRm</td>
</tr>
<tr>
<td>11110101</td>
<td>Op1</td>
<td></td>
<td>CRn</td>
<td>Rd</td>
<td>CPNum</td>
<td>Op2</td>
<td>1</td>
<td>CRm</td>
</tr>
<tr>
<td>11111</td>
<td>SWI Number</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Diagram

[Diagram of ARM ISA instruction set depicting various fields and their functionalities.]
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 (>=)
- **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.
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.

The ARM Instruction Set - ARM University Program - V1.0
"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?
```
Idealism

- Wouldn’t it be nice
  - If the branch is eliminated (predicated) when it will actually be mispredicted
  - If the branch were predicted when it will actually be correctly predicted

- Wouldn’t it be nice
  - If predication did not require ISA support
Improving Predicated Execution

- Three major limitations of predication
  1. **Adaptivity**: non-adaptive to branch behavior
  2. **Complex CFG**: inapplicable to loops/complex control flow graphs
  3. **ISA**: Requires large ISA changes

- **Wish Branches** [Kim+, MICRO 2005]
  - Solve 1 and partially 2 (for loops)

- **Dynamic Predicated Execution**
  - **Diverge-Merge Processor** [Kim+, MICRO 2006]
    - Solves 1, 2 (partially), 3
Wish Branches

- The **compiler** generates code (with wish branches) that can be executed **either** as predicated code **or** non-predicated code (normal branch code).
- The **hardware decides** to execute predicated code or normal branch code at run-time based on the confidence of branch prediction.
- **Easy to predict:** normal branch code
- **Hard to predict:** predicated code

Wish Jump/Join

Normal branch code

- p1 = (cond) branch p1, TARGET
- mov b, 1
- jmp JOIN
- TARGET: mov b, 0

Predicated code

- p1 = (cond)
- (!p1) mov b, 1
- (p1) mov b, 0

High Confidence

- wish jump
- nop
- wish join

Low Confidence

- wish jump
- nop
- wish join

Join code

- p1 = (cond)
- wish_jump p1
- TARGET

- (!p1) mov b, 1
- wish_jump (!p1)
- JOIN

- TARGET:
- (p1) mov b, 0

- JOIN:
- wish jump/join code

30
Wish Branches vs. Predicated Execution

- **Advantages compared to predicated execution**
  - *Reduces the overhead* of predication
  - Increases the benefits of predicated code by allowing the compiler to generate more **aggressively-predicated code**
  - Makes predicated code less dependent on machine configuration (e.g. branch predictor)

- **Disadvantages compared to predicated execution**
  - Extra branch instructions use machine resources
  - Extra branch instructions increase the contention for branch predictor table entries
  - Constrains the compiler’s scope for **code optimizations**
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 (branch prediction)
  - 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)
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
### Remember: Branch Types

<table>
<thead>
<tr>
<th>Type</th>
<th>Direction at fetch time</th>
<th>Number of possible next fetch addresses?</th>
<th>When is next fetch address resolved?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conditional</td>
<td>Unknown</td>
<td>2</td>
<td>Execution (register dependent)</td>
</tr>
<tr>
<td>Unconditional</td>
<td>Always taken</td>
<td>1</td>
<td>Decode (PC + offset)</td>
</tr>
<tr>
<td>Call</td>
<td>Always taken</td>
<td>1</td>
<td>Decode (PC + offset)</td>
</tr>
<tr>
<td>Return</td>
<td>Always taken</td>
<td>Many</td>
<td>Execution (register dependent)</td>
</tr>
<tr>
<td>Indirect</td>
<td>Always taken</td>
<td>Many</td>
<td>Execution (register dependent)</td>
</tr>
</tbody>
</table>

Different branch types can be handled differently
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

Conditional (Direct) Branch

Indirect Jump

$$R1 = \text{MEM}[R2]$$
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 $\rightarrow$ 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
    - \( nPC = \text{predicted target addr} \)
    - *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: Concepts

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

- Fetch 4 instr/cycle

Two major approaches:

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

2) Superscalar
   - Hardware detects dependencies between instrs that are fetched in the same cycle
Review of Last Few Lectures

- 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
  - Predicated execution
  - Multipath execution
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

\[
\begin{align*}
\text{FMUL R4} & \leftarrow \text{R1, R2} \\
\text{ADD} & \leftarrow \text{R3} \\
\text{FMUL R2} & \leftarrow \text{R5, R6} \\
\text{ADD} & \leftarrow \text{R4}
\end{align*}
\]

- What is wrong with this picture?
  - What if FMUL incurs an exception?
  - Sequential semantics of the ISA NOT preserved!
Exceptions vs. 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)
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
  - Remember von Neumann vs. dataflow

- 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)
Ensuring Precise Exceptions in Pipelining

- Idea: Make each operation take the same amount of time

```
FMUL R3 ← R1, R2  
ADD   R4 ← R1, R2
```

- Downside
  - What about memory operations?
  - Each functional unit takes 500 cycles?
Solutions

- Reorder buffer
- History buffer
- Future register file
- Checkpointing

Recommended Reading
Solution I: Reorder Buffer (ROB)

- **Idea:** Complete instructions out-of-order, but reorder them before making results visible to architectural state
- When instruction is decoded it reserves an entry in the ROB
- When instruction completes, it writes result into ROB entry
- When instruction oldest in ROB and it has completed without exceptions, its result moved to reg. file or memory
What’s in a ROB Entry?

- Need valid bits to keep track of readiness of the result(s)
Reorder Buffer: Independent Operations

- Results first written to ROB, then to register file at commit time

- What if a later operation needs a value in the reorder buffer?
  - Read reorder buffer in parallel with the register file. How?
We did not cover the following slides in lecture. These are for your preparation for the next lecture.
Reorder Buffer: How to Access?

- A register value can be in the register file, reorder buffer, (or bypass/forwarding paths)
Simplifying Reorder Buffer Access

- **Idea:** Use indirection
- **Access register file first**
  - If register not valid, register file stores the ID of the reorder buffer entry that contains (or will contain) the value of the register
  - Mapping of the register to a ROB entry
- **Access reorder buffer next**

- **What is in a reorder buffer entry?**

<table>
<thead>
<tr>
<th>V</th>
<th>DestRegId</th>
<th>DestRegVal</th>
<th>StoreAddr</th>
<th>StoreData</th>
<th>PC/IP</th>
<th>Control/valid bits</th>
<th>Exc?</th>
</tr>
</thead>
</table>

- **Can it be simplified further?**
Aside: Register Renaming with a Reorder Buffer

- Output and anti dependencies are not true dependencies
  - WHY? The same register refers to values that have nothing to do with each other
  - They exist due to lack of register ID’s (i.e. names) in the ISA

- The register ID is renamed to the reorder buffer entry that will hold the register’s value
  - Register ID → ROB entry ID
  - Architectural register ID → Physical register ID
  - After renaming, ROB entry ID used to refer to the register

- This eliminates anti- and output- dependencies
  - Gives the illusion that there are a large number of registers
In-Order Pipeline with Reorder Buffer

- **Decode (D):** Access regfile/ROB, allocate entry in ROB, check if instruction can execute, if so **dispatch** instruction
- **Execute (E):** Instructions can complete out-of-order
- **Completion (R):** Write result to reorder buffer
- **Retirement/Commit (W):** Check for exceptions; if none, write result to architectural register file or memory; else, flush pipeline and start from exception handler
- **In-order dispatch/execution, out-of-order completion, in-order retirement**
Reorder Buffer Tradeoffs

- **Advantages**
  - Conceptually simple for supporting precise exceptions
  - Can eliminate false dependencies

- **Disadvantages**
  - Reorder buffer needs to be accessed to get the results that are yet to be written to the register file
    - **CAM or indirection** → increased latency and complexity

- **Other solutions aim to eliminate the cons**
  - History buffer
  - Future file
  - Checkpointing
Solution II: History Buffer (HB)

- **Idea:** Update the register file when instruction completes, but UNDO UPDATES when an exception occurs.

- When instruction is decoded, it reserves an HB entry.
- When the instruction completes, it stores the old value of its destination in the HB.
- When instruction is oldest and no exceptions/interrupts, the HB entry discarded.
- When instruction is oldest and an exception needs to be handled, old values in the HB are written back into the architectural state from tail to head.
History Buffer

- Advantage:
  - Register file contains up-to-date values. History buffer access not on critical path

- Disadvantage:
  - Need to read the old value of the destination register
  - Need to unwind the history buffer upon an exception → increased exception/interrupt handling latency

Diagram:

- Instruction Cache → Register File
- Register File → Func Unit
- History Buffer
- Used only on exceptions
Solution III: Future File (FF) + ROB

- **Idea:** Keep two register files (speculative and architectural)
  - Arch reg file: Updated in program order for precise exceptions
    - Use a reorder buffer to ensure in-order updated
  - Future reg file: Updated as soon as an instruction completes (if the instruction is the youngest one to write to a register)

- Future file is used for fast access to latest register values (speculative state)
  - Frontend register file

- Architectural file is used for state recovery on exceptions (architectural state)
  - Backend register file
*Future File*

- **Advantage**
  - No need to read the values from the ROB (no CAM or indirection)

- **Disadvantage**
  - Multiple register files
  - Need to copy arch. reg. file to future file on an exception
In-Order Pipeline with Future File and Reorder Buffer

- **Decode (D):** Access future file, allocate entry in ROB, check if instruction can execute, if so **dispatch** instruction
- **Execute (E):** Instructions can complete out-of-order
- **Completion (R):** Write result to reorder buffer and future file
- **Retirement/Commit (W):** Check for exceptions; if none, write result to architectural register file or memory; else, flush pipeline and start from exception handler
- **In-order dispatch/execution, out-of-order completion, in-order retirement**
When the **oldest instruction ready-to-be-retired is detected to have caused an exception**, the control logic:

- Recovers architectural state (register file, IP, and memory)
- Flushes all younger instructions in the pipeline
- Saves IP and registers (as specified by the ISA)
- Redirects the fetch engine to the exception handling routine
  - Vectored exceptions
Pipelining Issues: Branch Mispredictions

- A branch misprediction resembles an “exception”
  - Except it is not visible to software

- What about branch misprediction recovery?
  - Similar to exception handling except can be initiated before the branch is the oldest instruction
  - All three state recovery methods can be used

- Difference between exceptions and branch mispredictions?
  - Branch mispredictions are much more common → need fast state recovery to minimize performance impact of mispredictions
How Fast Is State Recovery?

- Latency of state recovery affects
  - Exception service latency
  - Interrupt service latency
  - Latency to supply the correct data to instructions fetched after a branch misprediction

- Which ones above need to be fast?

- How do the three state maintenance methods fare in terms of recovery latency?
  - Reorder buffer
  - History buffer
  - Future file
Branch State Recovery Actions and Latency

- Reorder Buffer
  - Wait until branch is the oldest instruction in the machine
  - Flush pipeline afterwards

- History buffer
  - Undo all instructions after the branch by rewinding from the tail of the history buffer until the branch & restoring old values into the register file

- Future file
  - Wait until branch is the oldest instruction in the machine
  - Copy arch. reg. file to future file and flush pipeline
Can We Do Better?

- **Goal:** Restore the frontend state (future file) such that the correct next instruction after the branch can execute right away after the branch misprediction is resolved.

- **Idea:** Checkpoint the frontend register state at the time a branch is fetched and keep the state updated with results of instructions older than the branch.

Checkpointing

- When a branch is decoded
  - Make a copy of the future file and associate it with the branch

- When an instruction produces a register value
  - All future file checkpoints that are younger than the instruction are updated with the value

- When a branch misprediction is detected
  - Restore the checkpointed future file for the mispredicted branch when the branch misprediction is resolved
  - Flush all younger instructions younger than the branch
  - Deallocate all checkpoints younger than the branch
Checkpointing

- Advantages?

- Disadvantages?
Registers versus Memory

- So far, we considered mainly registers as part of state

- What about memory?

- What are the fundamental differences between registers and memory?
  - Register dependences known statically – memory dependences determined dynamically
  - Register state is small – memory state is large
  - Register state is not visible to other threads/processors – memory state is shared between threads/processors (in a shared memory multiprocessor)
Pipelining Issues: Stores

- Handling out-of-order completion of memory operations
  - UNDOing a memory write more difficult than UNDOing a register write. Why?
  - One idea: Keep store address/data in reorder buffer
    - How does a load instruction find its data?
  - Store/write buffer: Similar to reorder buffer, but used only for store instructions
    - Program-order list of un-committed store operations
    - When store is decoded: Allocate a store buffer entry
    - When store address and data become available: Record in store buffer entry
    - When the store is the oldest instruction in the pipeline: Update the memory address (i.e. cache) with store data