Announcements

- Homework 2 due Wednesday (Feb 12)
- Lab 3 available online (due Feb 21)
Readings for Next Few Lectures (I)

- P&H Chapter 4.9-4.11

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


Readings for Next Few Lectures (II)

Readings Specifically for Today


  - More advanced pipelining
  - Interrupt and exception handling
  - Out-of-order and superscalar execution concepts
Readings for Friday and next Monday

- Virtual Memory
- P&H Chapter 5.4
- Hamacher et al., Chapter 8.8
Lab Late Day Policy Adjustment

- Your total late days have increased to 7

- Each late day beyond all exhausted late days costs you 15% of the full credit of the lab
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)
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 = MEM[R2] branch R1
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
More Ideas on Indirect Branches?

- Virtual Program Counter prediction
  - Idea: Use conditional branch prediction structures \textit{iteratively} to make an indirect branch prediction
  - i.e., devirtualize the indirect branch in hardware

- Curious?
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

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

Diagram:
```
???
```

```
Next Fetch Address
```
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
      → nullify 2\textsuperscript{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

| FMUL R4 ← R1, R2 | F D E E E E E E E E E E E W |
| ADD R3 ← R1, R2   | F D E W                      |
|                  | F D E W                      |
|                  | F D E W                      |
|                  | F D E W                      |
| FMUL R2 ← R5, R6 | F D E E E E E E E E E E E E W |
| ADD R4 ← R5, R6   | F D E W                      |
|                  | F D E W                      |

- 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**
  - Worst-case instruction latency determines all instructions’ latency
  - What about memory operations?
  - Each functional unit takes 500 cycles?
Solutions

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

Readings
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

```
Instruction Cache -> Register File -> Func Unit -> Func Unit -> Func Unit -> Reorder Buffer
```
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?
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

- Idea: Reducing reorder buffer entry storage

<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 disadvantages**
  - **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
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 updates
  - 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, copy architectural file to future file, 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 entire pipeline

- **History buffer**
  - Undo all instructions after the branch by rewinding from the tail of the history buffer until the branch & restoring old values one by one into the register file
  - Flush instructions in pipeline younger than the branch

- **Future file**
  - Wait until branch is the oldest instruction in the machine
  - Copy arch. reg. file to future file
  - Flush entire 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 checkpointed 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 instructions in pipeline younger than the branch
  - Deallocate checkpoints younger than the branch
Checkpointing

- Advantages?

- Disadvantages?