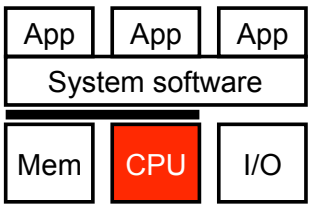
CMPE 110: Computer Architecture

Week 5
Out-of-Order Execution

Jishen Zhao (http://users.soe.ucsc.edu/~jzhao/)

[Adapted in part from Jose Renau, Mary Jane Irwin, Joe Devietti, Onur Mutlu, and others]

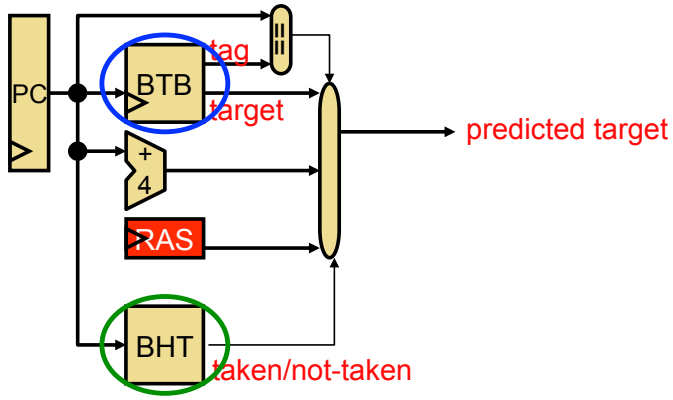
Review: Summary of pipelining



- Single-cycle datapaths vs. pipelined datapath
  - Basic pipelining
- Data hazards
- Structural hazards
- Multi-cycle operations
- Control hazards
- Fine-grained multithreading

Review: inside a branch predictor?

- BTB & branch direction predictor during fetch



- Step #1: is it a branch? BTB
- Step #2: is the branch taken or not taken? BHT
- Step #3: if the branch is taken, where does it go? BTB, RAS

Review: Uniprocessor Concurrency

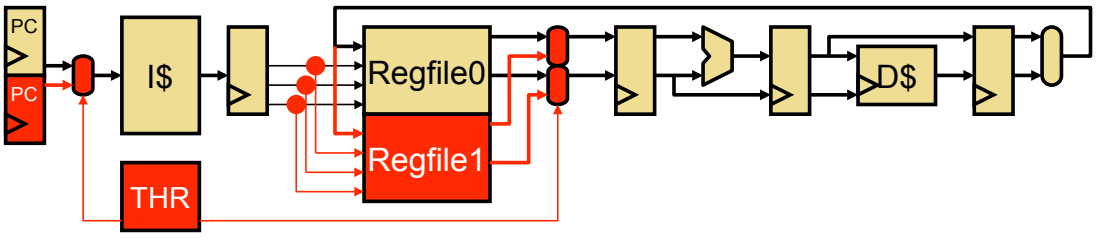
- **Software "thread":** Independent flows of execution
  - "Per-thread" state
    - Context state: PC, registers
    - Stack (per-thread local variables)
  - "Shared" state: globals, heap, etc.
  - Threads generally share the same memory space
    - A process is like a thread, but with its own memory space
  - Java has thread support built in, C/C++ use the pthreads library
- Generally, system software (the O.S.) manages threads
  - "Thread scheduling", "context switching"
  - In single-core system, all threads share one processor
    - Hardware timer interrupt occasionally triggers O.S.
    - Quickly swapping threads gives illusion of concurrent execution
  - Much more in an operating systems course

Review: Uniprocessor Concurrency

- Programmer explicitly creates multiple threads
- A "thread switch" can occur at any time
  - Pre-emptive multithreading by OS
- Common uses:
  - Handling user interaction (GUI programming)
  - Handling I/O latency (send network message, wait for response)
  - **Expressing parallel work via Thread-Level Parallelism (TLP)**

Review: Hardware Multithreading

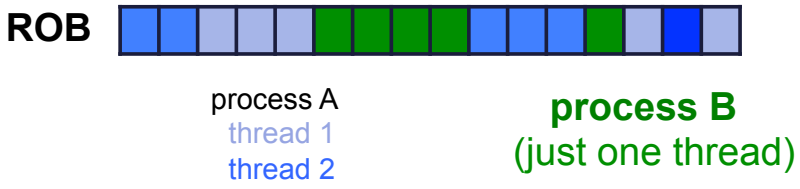
- **Not** the same as software multithreading!
- A **hardware thread** is a sequential stream of insns
  - could be a software thread or a single-threaded process



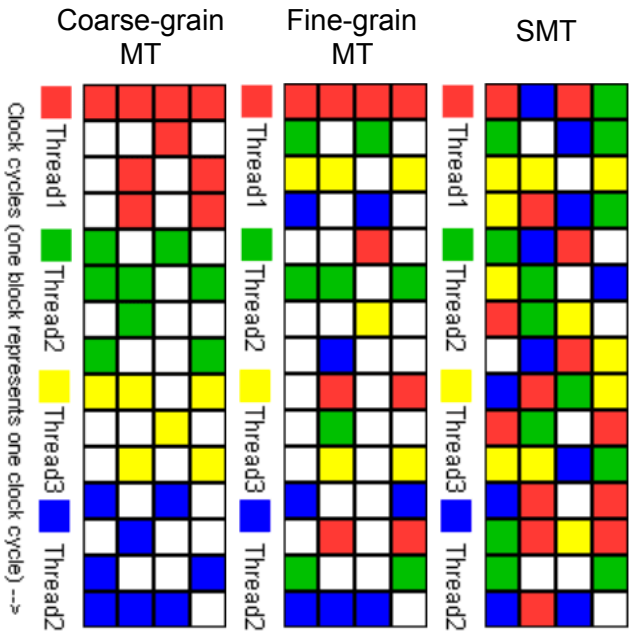
- **Hardware Multithreading (MT)**
  - Multiple hardware threads dynamically share a single pipeline
  - Replicate only per-thread structures: program counter & registers
  - Hardware interleaves instructions

Review: Hardware Multithreading

- Why use hw multithreading?
  - + **Multithreading improves utilization and throughput**
    - Single programs utilize <50% of pipeline (branch, cache miss)
    - allow insns from different hw threads in pipeline at once
  - **Multithreading does not improve single-thread performance**
    - Individual threads run as fast or even slower
  - **Coarse-grain MT**: switch on cache misses Why?
  - **Simultaneous MT**: no explicit switching, fine-grain interleaving
    - Intel's "hyperthreading"

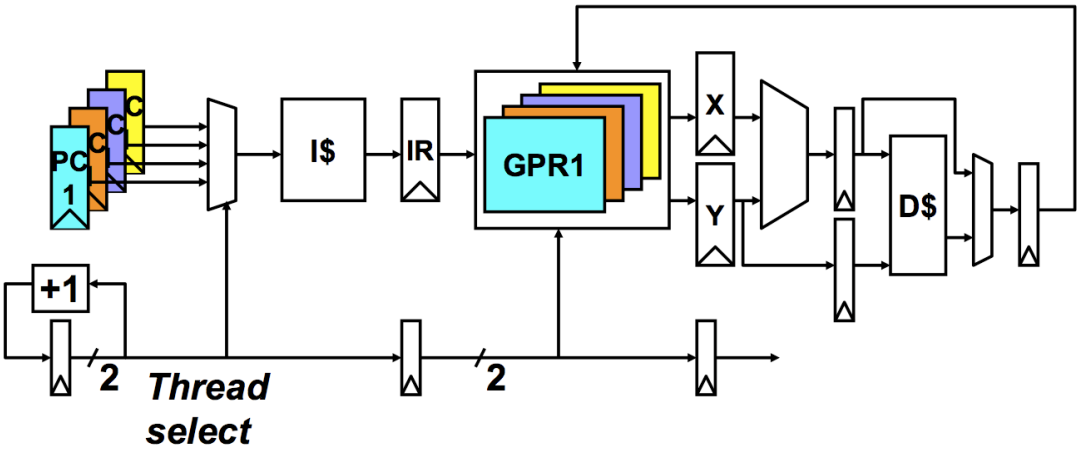


Reivew: Hardware Multithreading



Multithreaded Pipeline Example

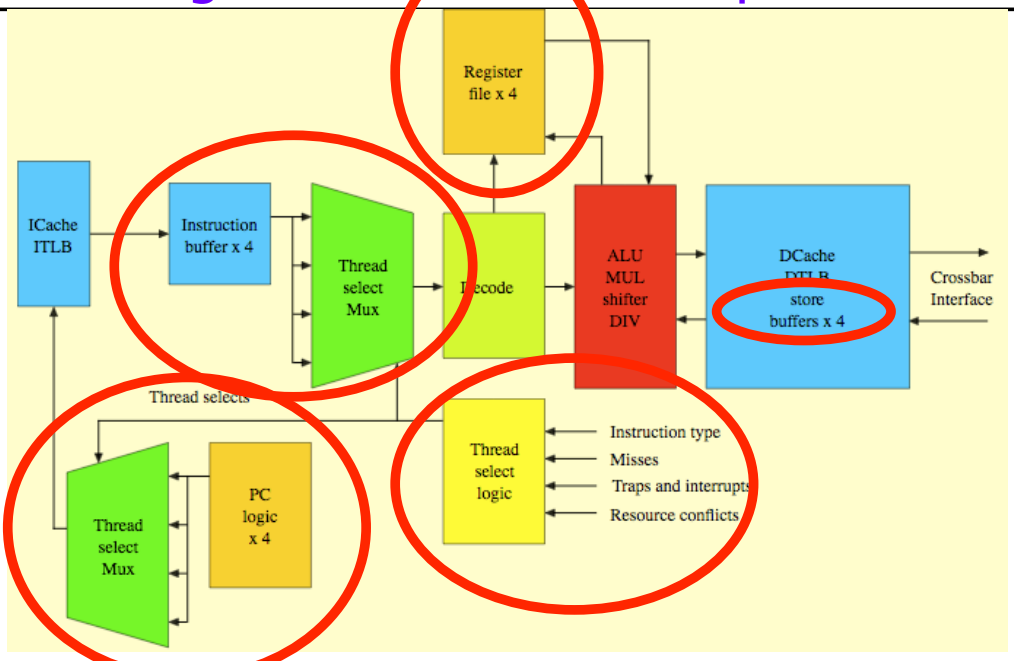
Need to store multiple "thread contexts"



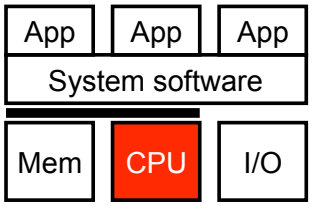
GPR: general-purpose register

Slide credit: Joel Emer

Sun Niagara Multithreaded Pipeline



Kongetira et al., "Niagara: A 32-Way Multithreaded Sparc Processor," IEEE Micro 2005.



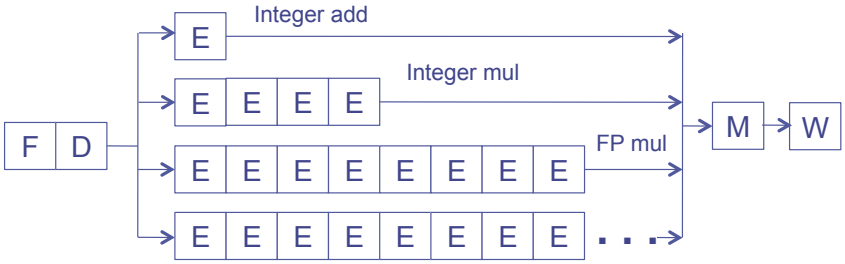
Out-of-order Execution

Preliminary: Dynamic Instruction Scheduling

- Hardware has knowledge of dynamic events on a per-instruction basis (i.e., at a very fine granularity)
  - Branch mispredictions
  - Load/store addresses
  - Cache misses
- Wouldn't it be nice if hardware did the scheduling of instructions?

An In-order Pipeline

F, D, E, M, W



- Problem: A true data dependency stalls dispatch of younger instructions into functional (execution) units
- Dispatch: Act of sending an instruction to an execution functional unit (i.e., in E stage)
- Load and store instructions can take multiple cycles in M stage

Can We Do Better?

- What do the following two pieces of code have in common (with respect to execution in the previous design)?

```
IMUL  R3 ← R1, R2          LD    R3 ← R1 (0) // M stage may take multiple cycles
ADD   R3 ← R3, R1          ADD   R3 ← R3, R1
ADD   R1 ← R6, R7          ADD   R1 ← R6, R7
IMUL  R5 ← R6, R8          IMUL  R5 ← R6, R8
ADD   R7 ← R9, R9          ADD   R7 ← R9, R9
```

- Answer: First ADD can stall the whole pipeline!
  - ADD cannot dispatch because its source registers unavailable
  - Later **independent** instructions cannot get executed
- How are the above code portions different?
  - Answer: Load latency is variable (unknown until runtime)
  - What does this affect? Think compiler vs. microarchitecture

Preventing Dispatch Stalls

- Multiple ways of doing it
- You have already seen at least one:
  - Fine-grained multithreading

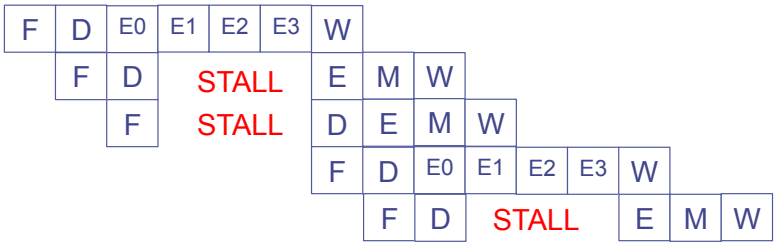
  May see more later on:
  - Value prediction
  - Compile-time instruction scheduling/reordering (i.e., static instruction scheduling)
- What are the disadvantages of fine-grained multithreading?
- Any other way to prevent dispatch stalls?
  - Problem: in-order dispatch (scheduling, or execution)
  - Solution: out-of-order dispatch (scheduling, or execution)

Out-of-order Execution (Dynamic Scheduling)

- Idea: Move the dependent instructions out of the way of independent ones (s.t. independent ones can execute)
  - Resting areas for dependent instructions: "**Reservation stations**"
- Monitor the source "values" of each instruction in the reservation stations
- When all source "values" of an instruction are available, "fire" (i.e. dispatch) the instruction
- Benefit:
  - Latency tolerance: Allows independent instructions to execute and complete in the presence of a long latency operation

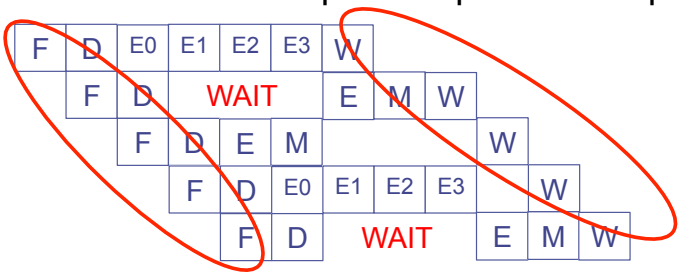
In-order vs. Out-of-order Dispatch

- In order dispatch + precise exceptions:



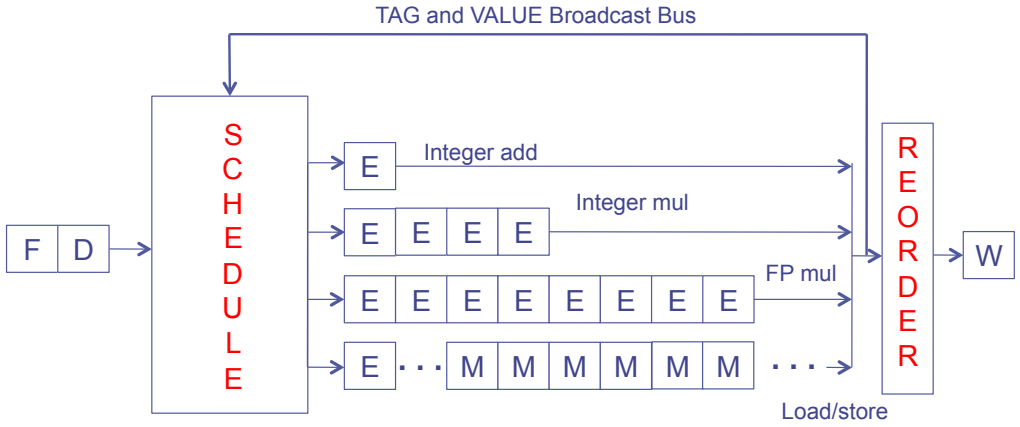
IMUL R3 ← R1, R2
ADD R3 ← R3, R1
ADD R1 ← R6, R7
IMUL R5 ← R6, R8
ADD R7 ← R3, R5

- Out-of-order dispatch + precise exceptions:



15 vs. 12 cycles

Two Humps in a Modern Pipeline



- Hump 1: Reservation stations (scheduling window)
- Hump 2: Reordering (reorder buffer, aka instruction window or active window)

Register Renaming

The processor only has 4 registers

```
add R3<- R1, R2
addi R1<- R4, #100
```

- Some dependencies are not true dependencies
  - The same register refers to values that have nothing to do with each other
  - **They exist because not enough register ID's (i.e. names) in the ISA**

True dependency:

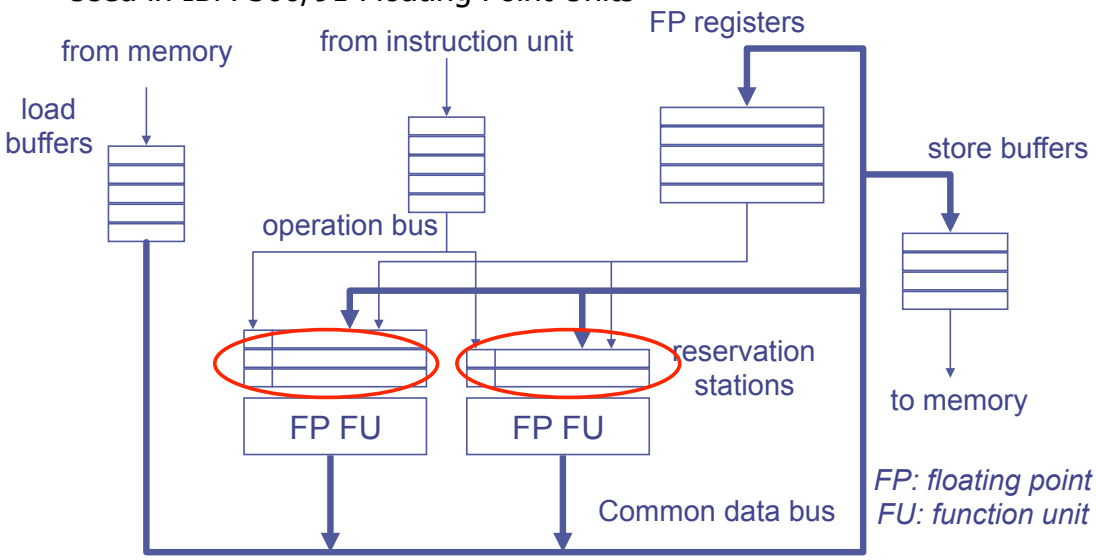
```
addi R1<- R4, #100
add R3<- R1, R2
```

- Solution: The register ID is renamed to the reservation station entry that will hold the register's value
  - Register ID → Reservation Station entry ID
  - After renaming, Reservation Station entry ID is used to refer to the register
- This eliminates the dependency shown in our example
  - Achieves (approximates) the performance of having a large number of registers, even though ISA does not support that many

Register renaming

Out-of-order execution (with register renaming) invented by Robert Tomasulo in 1960's

- Used in IBM 360/91 Floating Point Units



Tomasulo's Algorithm

- OoO with register renaming invented by Robert Tomasulo
  - Used in IBM 360/91 Floating Point Units
  - Tomasulo, "An Efficient Algorithm for Exploiting Multiple Arithmetic Units," IBM Journal of R&D, Jan. 1967.

- What is the major difference today?
  - Precise exceptions: IBM 360/91 did NOT have this
  - Patt, Hwu, Shebanow, "HPS, a new microarchitecture: rationale and introduction," MICRO 1985.
  - Patt et al., "Critical issues regarding HPS, a high performance microarchitecture," MICRO 1985.

- Variants are used in most high-performance processors
  - Initially in Intel Pentium Pro, AMD K5
  - Alpha 21264, MIPS R10000, IBM POWER5, IBM z196, Oracle UltraSPARC T4, ARM Cortex A15