

Portland State University  
ECE 587/687

# The Microarchitecture of Superscalar Processors

© Copyright by Alaa Alameldeen and Haitham Akkary 2009

# Program Representation

- An application is written as a program, typically using a high level language
- Program is compiled into static machine code (binary)
- Sequencing model implicit in the program
- The sequence of executed instructions forms a dynamic instruction stream
- The address of the next dynamic instruction:
  - ◆ Incremented program counter
  - ◆ Target of a taken branch

# Sequential Execution Model

- Inherent in instruction sets and program binaries
- Led to the concept of precise architecture state
  - ◆ Interrupt and restart
  - ◆ Exceptions
  - ◆ Branch mispredictions
- Out of order issue deviates from sequential execution
  - ◆ But we still need to maintain binary compatibility and retain appearance of sequential execution

# Dependences and Parallel Execution

- To execute more instructions in parallel, control dependences need to be addressed:
  - ◆ Program Counter (PC)
  - ◆ Branches
- To overcome PC dependence, one can view the program as a collection of basic blocks, separated by branches
- There is a limited number of parallel instructions on average within basic blocks

# Dependences and Parallel Execution (cont.)

- Instructions have to be serialized according to true data dependences
  - ◆ A true dependence appears as a read after write (RAW) sequence
- Ideally, we should eliminate output dependences and anti-dependences
  - ◆ An output dependence appears as write after write (WAW) sequence
  - ◆ An anti-dependence appears as write after read (WAR) sequence

# Elements of Superscalar Processing

- **Fetch:** Strategies for fetching multiple instructions every cycle, supported by
  - ◆ Predicting branch outcomes
  - ◆ Fetching beyond conditional branch instructions, well before branches are executed
- **Decode:** Methods for determining true register dependencies and eliminating artificial dependencies
  - ◆ Register renaming
  - ◆ Mechanisms to communicate register values during execution
- **Issue/Dispatch:** Methods for issuing multiple instructions in parallel
  - ◆ Based upon availability of inputs, not upon program order

# Elements of Superscalar Processing (cont.)

- **Execution:** Parallel execution resources
  - ◆ Multiple pipelined functional units
  - ◆ Memory hierarchies capable of simultaneously serving multiple memory requests
- **Memory:** Methods for communicating data through memory via load and store instructions, potentially issued out of order
  - ◆ Memory interfaces have to allow for the dynamic and often unpredictable behavior of memory hierarchies
- **Commit:** Methods for committing architecture state in order
  - ◆ Maintain an outward appearance of sequential execution

# Typical Superscalar Microarchitecture

- Fig. 3 (Paper): Parallel execution model
- Fig. 4 (Paper): Microarchitecture or hardware organization of a typical superscalar processor

# Instruction Fetch

- Read instructions from the instruction cache and write them to a queue (instr. buffer in Fig 4)
  - ◆ The number of instructions fetched per cycle should at least match the peak decode rate (why?)
  - ◆ The fetcher must be told the address of the next block of instructions to fetch
- An instruction cache is usually organized as lines of several instructions
  - ◆ A cache line starts on a fixed boundary (regardless of the instruction needed from the line)
  - ◆ Question: What are the pros and cons of having separate I- and D- caches?

# Instruction Fetch (Cont.)

- Calculating the next address to fetch
  - ◆ Non-branch instructions:
    - the PC is incremented by the number of instructions in a cache block
  - ◆ Branch instructions: the fetch unit has to
    - Recognize a branch
    - Determine its outcome (taken or not taken)
    - Compute branch target address
    - Fetch the next block using
      - Next sequential address or
      - Branch target address

# Instruction Fetch (Cont.)

- Branch prediction is used to avoid having to wait for the branch execution to complete
  - ◆ Target comes from Branch target buffer (BTB)
  - ◆ Outcome comes from
    - Static prediction based on branch type or profile (or even compiler hints)
    - Dynamic prediction based on result of previous branches
- If branch is mispredicted, we must be able to undo the work and fetch the correct instruction
  - ◆ This incurs significant misprediction penalty
- Branch prediction discussed in more detail later in the course

# Instruction Fetch (Cont.)

- Transferring control to target address on a taken branch could cause pipeline bubbles
  - ◆ Stockpile instructions in instruction queue
  - ◆ Or keep next address in cache block
  - ◆ Or use delayed branches?
- The instruction queue helps
  - ◆ Smooth fetch irregularities caused by cache misses
  - ◆ Sustain fetch bandwidth in cycles when fewer than the maximum #instructions can be fetched

# Instruction Fetch (Cont.)

- Superscalar machines pay a penalty for instruction misalignment
  - ◆ Branches and targets don't always fall on cache line boundaries
  - ◆ Fetched instructions that are not executed waste fetch bandwidth
  - ◆ Sometimes called instruction cache fragmentation due to branches

# ICache Fragmentation

<b>X</b>							
<b>X+32</b>	<b>BR X+188</b>	<b>Discard</b>					
<b>X+64</b>							
<b>X+96</b>							
<b>X+128</b>							
<b>X+160</b>	<b>Discard</b>						<b>X+188</b>
<b>X+192</b>							

# Instruction Fetch (Cont.)

- Cache fragmentation caused by branches places a severe limit on very wide superscalars
  - ◆ Easy to fetch sequential runs of instructions
  - ◆ However, the average sequential run length is  $\sim 6$  for general integer programs
  - ◆ The distribution is very broad, with a few long runs raising the average
  - ◆ How many decode cycles are needed for 6 fetched instructions?
    - 3 decode cycles for a two instruction decoder
    - Only 1.5 decode cycles for a four instruction decoder

## Instruction Fetch (Cont.)

- Given enough fetch bandwidth, the fetcher can realign or merge instructions from multiple lines to make more efficient use of the decoder
  - ◆ For a branch target in the middle of a cache line, the fetcher combines the cache line with the one following it
  - ◆ Decoder "lines" are not aligned with cache lines
  - ◆ Harder to find the program counter associated with one instruction
- Trace cache discussed in more detail later in the course

# Instruction Decode

- Instructions are removed from the instruction queue
- Execution tuples are set for each decoded instruction containing
  - ◆ Opcode: Operation to be executed
  - ◆ Sources: Identities of storage elements where the inputs reside
  - ◆ Destination: Identity of the storage element where result must be placed

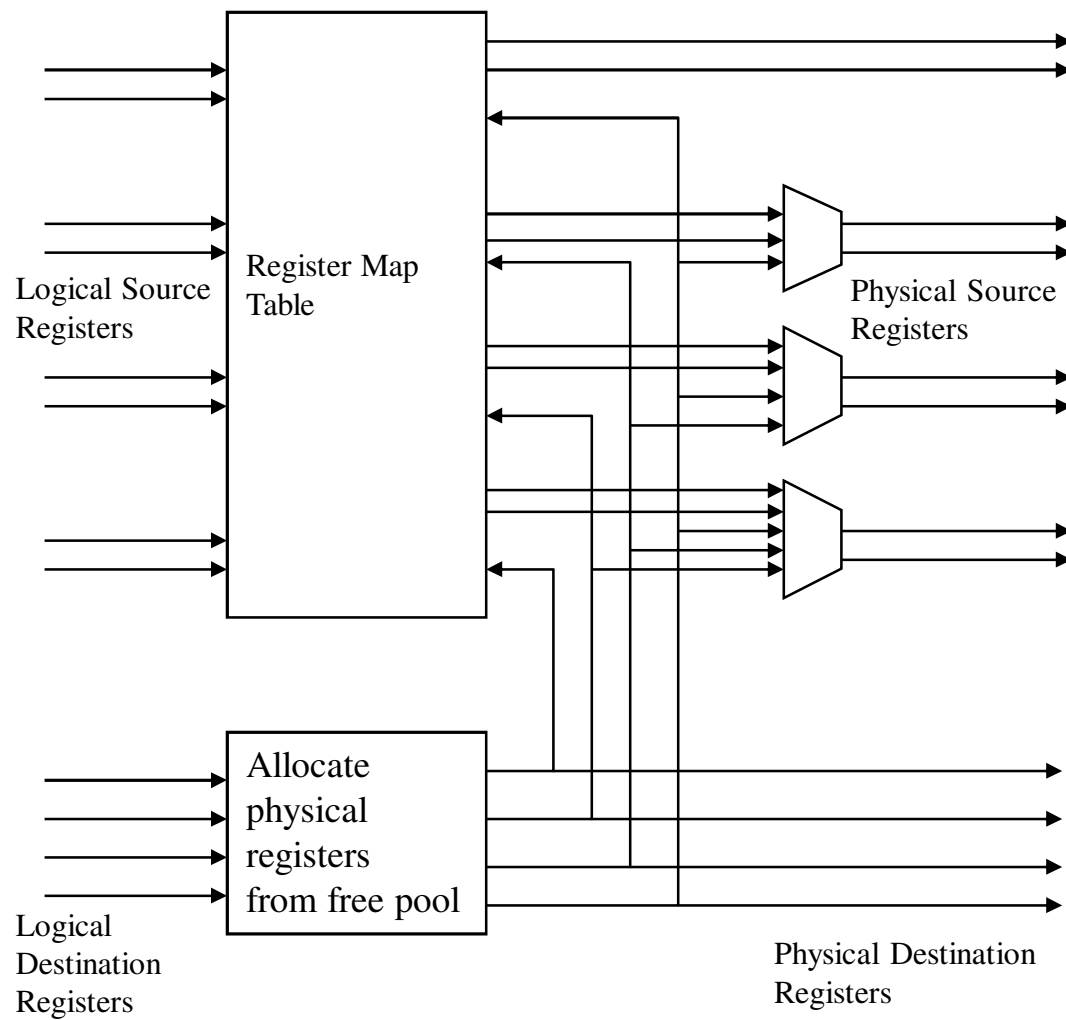
# Instruction Decode (Cont.)

- In the static program, the input and output identifiers represent
  - ◆ Storage locations in the “logical” register file OR
  - ◆ Storage locations in memory
- To overcome WAR and WAW hazards, register renaming maps the register “logical” identifiers into “physical” storage locations
- Allocation logic assigns each instruction physical storage for the result as well as entries in all required instruction buffers

# Instruction Rename

- The decoder looks at one or more instructions and releases them to scheduling stations after renaming
- Register values created by an instruction are assigned physical locations, and recorded in a map table
  - ◆ Map table has as many entries as there are logical registers
- Source register mappings are read from the map table and attached to the instruction
- Renaming happens sequentially
  - ◆ Map table bypass is sometimes necessary
- Subsequent stages in the pipeline use mappings attached to an instruction tuple to read and write the physical locations of register values

# Rename Map Table



# Renaming Methods

- There are two methods commonly used:
  - ◆ Renaming with a physical register file larger than the logical register file
  - ◆ Renaming using a Reorder Buffer (ROB) and a physical register file equal in size to the number of logical registers

# Renaming with a Physical RF

- Paper Fig. 5
- A free list of unused physical registers is kept
- New register results are assigned physical registers from the free list
- Reclaiming of physical registers into the free list:
  - ◆ Usage count is 0 and logical register has been renamed to another physical register
  - ◆ Subsequent instruction writing to the same logical register is committed
- Register map table is checkpointed at conditional branches (why?)

## Freeing Physical Registers at Retirement

$I1 \rightarrow R5 \rightarrow P3$

.

.

$I4 \rightarrow R5 \rightarrow P5$  (free P3 when retired)

.

.

$I7 \leftarrow R5 \leftarrow P5$

# Renaming with a Reorder Buffer

- Physical registers are allocated sequentially in the Reorder Buffer
- Physical registers are freed and their values are copied to the register file at retirement
- Mapping table maps logical registers to entries in the Reorder Buffer or the Register File
- Paper Fig 6 and 7
- Branch handling options:
  - ◆ Map table checkpoints
  - ◆ Resume renaming from the correct path after mispredicted branch has retired

# Instruction Issue

- After instructions are fetched, decoded and renamed, they are placed in instruction buffers where they wait until issue
- An instruction can be issued when its input operands are ready, and there is a functional unit available
- Paper Fig. 8. is an example of parallel execution schedule

## Instruction Issue (Cont.)

- All out-of-order issue methods must handle the same basic steps
  - ◆ Identify all instructions that are ready to issue
  - ◆ Select among ready instructions to issue as many as possible
  - ◆ Issue the selected instructions, e.g., pass operands and other information to the functional units
  - ◆ Reclaim instruction window storage used by the now issued instructions

# Methods of Organizing Instruction Issue Buffers

- Single shared queue
  - ◆ Only for in-order issue
- Multiple queues, one per instruction type
- Multiple reservation stations, one per instruction type
  - ◆ Fig. 10. Shows a typical reservation station
- Single central reservation stations buffer

# Multiple Queues

- Requiring instructions to be issued in order at a functional unit greatly simplifies the identification and selection logic
- Instructions from different queues could be allowed to issue out of order

# Reservation Stations

## ■ Benefits

- ◆ Logic to identify and select ready instructions is simpler since it need only consider a few locations
- ◆ Storage can be optimized for each type of functional unit
  - e.g., stores need not have storage for two source operands

## ■ Drawback

- ◆ Storage is statically allocated to functional units
- ◆ This can result in either wasted storage or a resource bottleneck for some programs

# Central Window

- Benefits
  - ◆ Only one copy of identification and selection logic
  - ◆ Only one copy of storage reclamation logic
  - ◆ Dynamically allocated storage
- Drawbacks
  - ◆ Complex identification and selection logic
  - ◆ Complex storage reclamation logic
  - ◆ Each storage location must be as big as the largest instruction
  - ◆ Functional unit arbitration must be handled

# Memory Ordering

- Stores consist of address and data uops
- Store addresses are buffered in a queue
- Store addresses remain buffered until:
  - ◆ Store data is available
  - ◆ Store instruction is committed in the reorder buffer
- New load addresses are checked with the waiting store addresses. If there is a match:
  - ◆ The load waits OR
  - ◆ Store data is bypassed to the matching load
- Fig. 11. shows typical memory ordering logic
- Memory ordering discussed later in the course
- More details in ECE 588/688

# Commit (Retire)

- Implements appearance of sequential execution
- Recovering a precise state:
  - ◆ Need to maintain both state required for recovery and state being updated
  - ◆ Recovery options:
    - History buffer
    - Future File
- Precise interrupts discussed later in the course

# Reading Assignments

- Wednesday:
  - ◆ Simple-scalar technical report (Read)
  - ◆ Tutorial (Skim)
  - ◆ No reviews are due on Wednesday
- Monday:
  - ◆ T.-Y. Yeh and Y.N. Patt, “Alternative Implementations of Two-Level Adaptive Branch Prediction,” ISCA 1992. (Review)
  - ◆ S. McFarling, “Combining Branch Predictors,” Technical Report TN-36, Digital Western Research Laboratory, June 1993. (Read)