CSE 431
Computer Architecture
Fall 2015

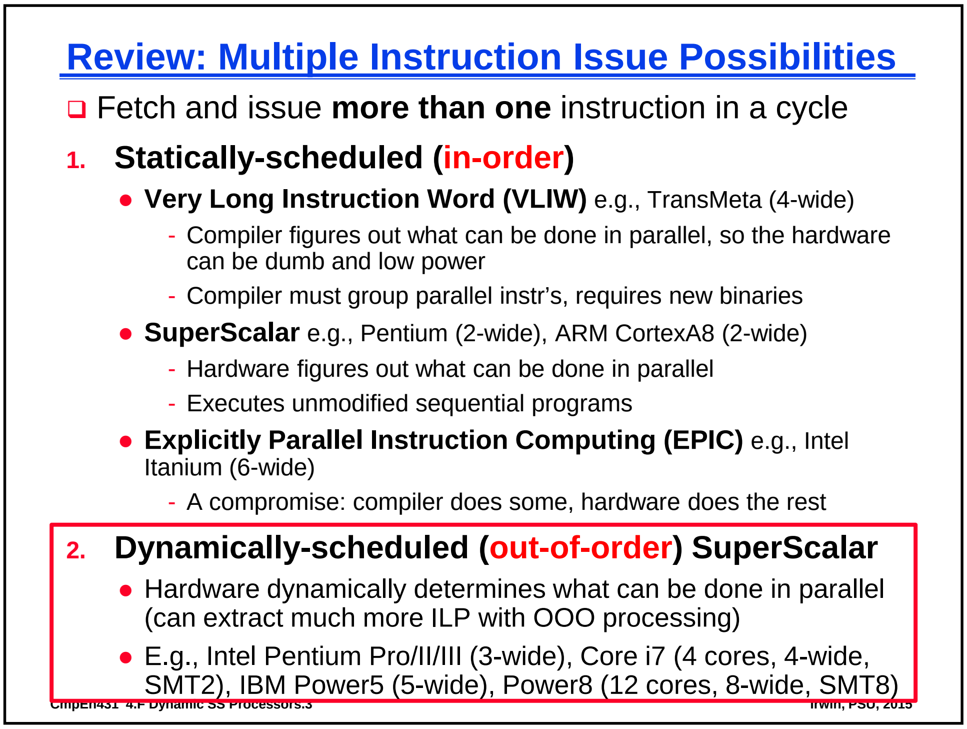
Chapter 4F: The Dynamic SuperScalar (OOO) Processor

Mary Jane Irwin (www.cse.psu.edu/~mji)

[Adapted from *Computer Organization and Design, 5th Edition*,
Patterson & Hennessy, © 2014, MK
With additional thanks/credits to Amir Roth, Milo Martin, CIS/UPenn]

Reminders

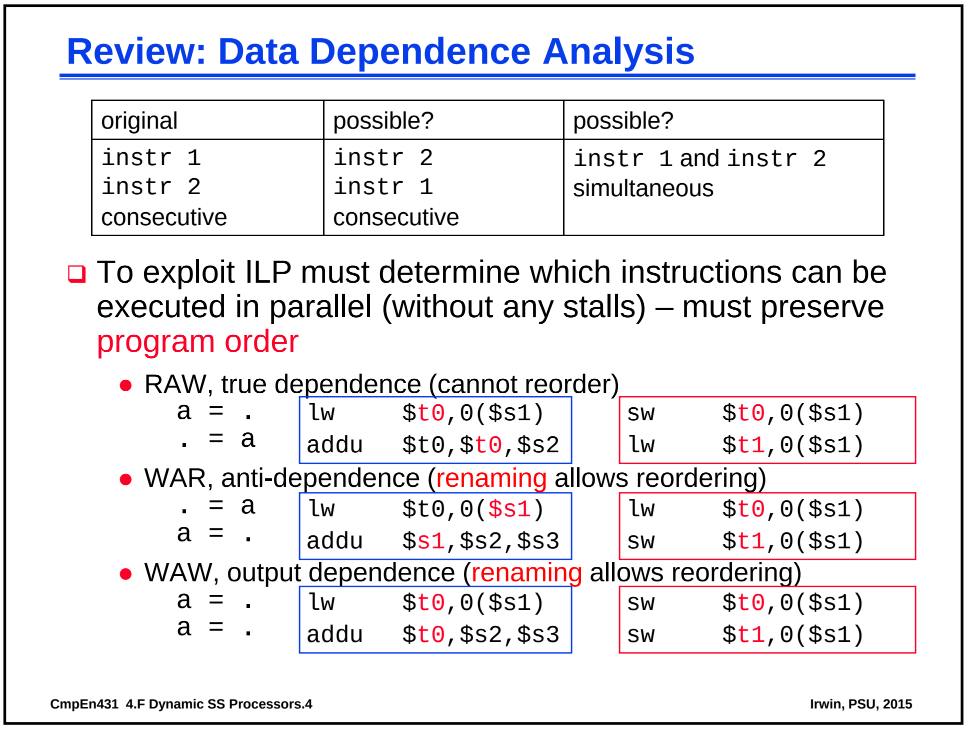
- This week
  - Dynamic superscalar processors – out-of-order and SMT
    - PH 4.10 and PH 6.4
- Next week
  - Multicores and multicore (multiprocessors); cache coherence
    - PH 2.11, 5.10, 6.1, 6.5
- Reminders
  - HW4 is due Oct 26th, HW5 will come out Oct 27th and will be due Nov 12th, HW6 (Project) will come out Nov 5th and will be due Dec 14th
  - Quiz4 will close on Oct 23rd, Quiz5 will close on Nov 2nd, Quiz 6 will close on Nov 16th
  - Second (and **last**) evening exam scheduled
    - Tuesday, November 17, 20:15 to 22:15, Location 22 Deike
    - If you have a conflict I **MUST** KNOW before October 31st



Review: Multiple Instruction Issue Possibilities

- Fetch and issue **more than one** instruction in a cycle

1. **Statically-scheduled (in-order)**
   - **Very Long Instruction Word (VLIW)** e.g., TransMeta (4-wide)
     - Compiler figures out what can be done in parallel, so the hardware can be dumb and low power
     - Compiler must group parallel instr's, requires new binaries
   - **SuperScalar** e.g., Pentium (2-wide), ARM CortexA8 (2-wide)
     - Hardware figures out what can be done in parallel
     - Executes unmodified sequential programs
   - **Explicitly Parallel Instruction Computing (EPIC)** e.g., Intel Itanium (6-wide)
     - A compromise: compiler does some, hardware does the rest
2. **Dynamically-scheduled (out-of-order) SuperScalar**
   - Hardware dynamically determines what can be done in parallel (can extract much more ILP with OOO processing)
   - E.g., Intel Pentium Pro/II/III (3-wide), Core i7 (4 cores, 4-wide, SMT2), IBM Power5 (5-wide), Power8 (12 cores, 8-wide, SMT8)



Review: Data Dependence Analysis

original	possible?	possible?

`instr 1`
`instr 2`
consecutive
	`instr 2`
	`instr 1`
	consecutive
		`instr 1` and `instr 2`
		simultaneous

- To exploit ILP must determine which instructions can be executed in parallel (without any stalls) – must preserve program order
  - RAW, true dependence (cannot reorder)

```
a = .    lw    $t0,0($s1)        sw    $t0,0($s1)
. = a    addu  $t0,$t0,$s2       lw    $t1,0($s1)
```

  - WAR, anti-dependence (renaming allows reordering)

```
. = a    lw    $t0,0($s1)        lw    $t0,0($s1)
a = .    addu  $s1,$s2,$s3       sw    $t1,0($s1)
```

  - WAW, output dependence (renaming allows reordering)

```
a = .    lw    $t0,0($s1)        sw    $t0,0($s1)
a = .    addu  $t0,$s2,$s3       sw    $t1,0($s1)
```

More on Data Dependence

- RAW
  - When more than one applies, RAW dominates:

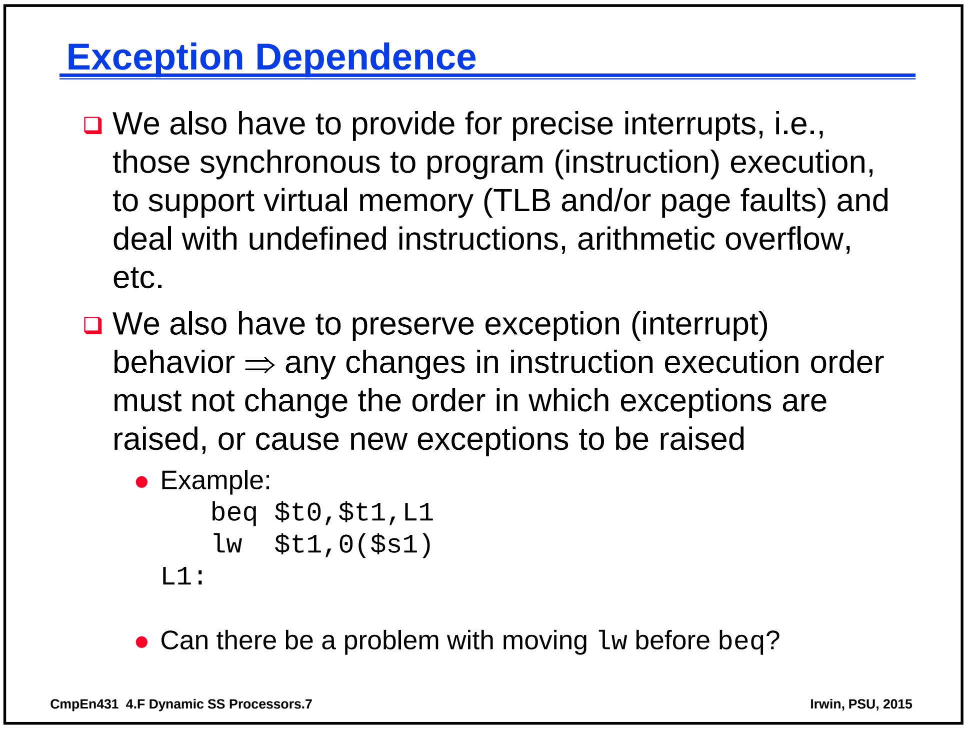
```
add  $t0,$t1,$t2
addi $t0,$t0,1
```

  - Must be respected: no way to avoid sequential execution

- WAR/WAW on registers
  - Two different things can happen when using the same name depending on instruction ordering
  - Can be eliminated by **register renaming**
- WAR/WAW on memory
  - Can't rename memory and don't know if there is an actual dependency until the effective address is known (in Exec)
  - Need to use something other than register renaming

Control Dependence

- Using branch prediction we may end up executing instructions that should **not** have been executed (i.e., the prediction is incorrect), thereby violating the control dependencies
  - But, as long as we don't change the visible machine state, it is still okay (we just used some energy doing work that has to be thrown away)
- The key is having a way to execute *past* predicted branches *without* changing the visible machine state until you know for sure that the branch prediction was correct

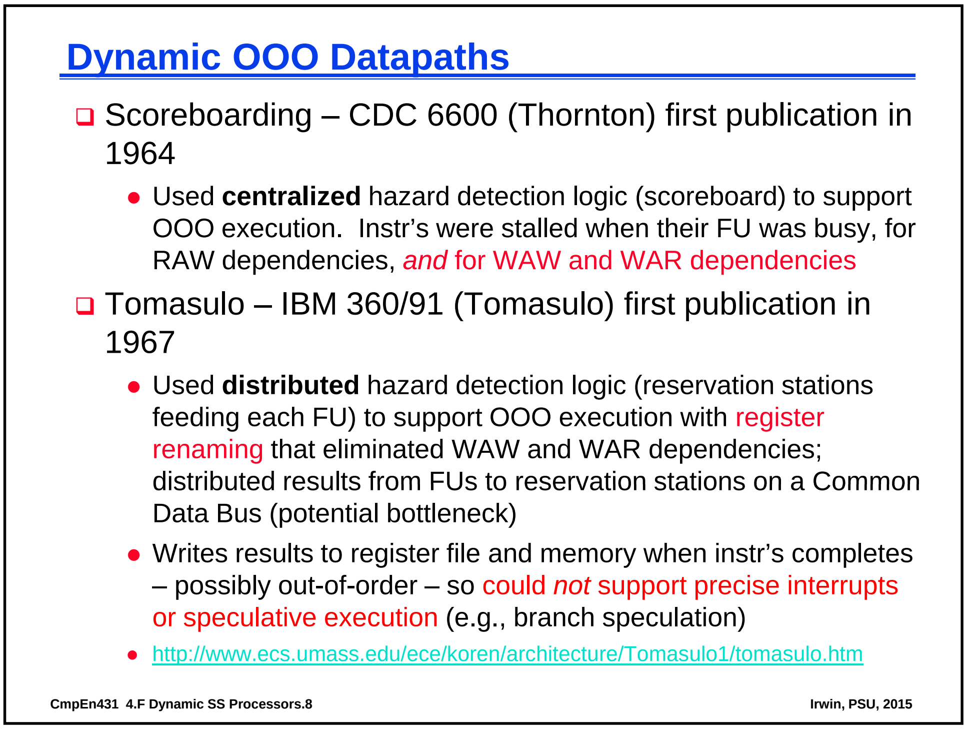


Exception Dependence

- We also have to provide for precise interrupts, i.e., those synchronous to program (instruction) execution, to support virtual memory (TLB and/or page faults) and deal with undefined instructions, arithmetic overflow, etc.
- We also have to preserve exception (interrupt) behavior ⇒ any changes in instruction execution order must not change the order in which exceptions are raised, or cause new exceptions to be raised
  - Example:

```
    beq $t0,$t1,L1
    lw  $t1,0($s1)
L1:
```

  - Can there be a problem with moving `lw` before `beq`?



Dynamic OOO Datapaths

- Scoreboarding – CDC 6600 (Thornton) first publication in 1964
  - Used **centralized** hazard detection logic (scoreboard) to support OOO execution. Instr's were stalled when their FU was busy, for RAW dependencies, *and* for WAW and WAR dependencies
- Tomasulo – IBM 360/91 (Tomasulo) first publication in 1967
  - Used **distributed** hazard detection logic (reservation stations feeding each FU) to support OOO execution with register renaming that eliminated WAW and WAR dependencies; distributed results from FUs to reservation stations on a Common Data Bus (potential bottleneck)
  - Writes results to register file and memory when instr's completes – possibly out-of-order – so could *not* support precise interrupts or speculative execution (e.g., branch speculation)
  - http://www.ecs.umass.edu/ece/koren/architecture/Tomasulo1/tomasulo.htm

More Recent Dynamic OOO Datapaths

- HPS – (Hwu, Patt, Shebanow) first publication in 1985
  - Used a register alias table and distributed node alias tables that fed each FUs (essentially reservation stations) to support OOO execution with register renaming; distributed results from FUs to reservation stations on multiple distribution buses (one per FU)
  - Supported precise interrupts and speculative execution with a checkpoint repair mechanism
- RUU – (Sohi) first publication in 1987
  - Uses a centralized Register Update Unit (RUU) that 1) receives new instr’s from decode, 2) renames registers, 3) monitors the (single) result bus to resolve dependencies, 4) determines when instr’s are ready to issue (send for execution), and 5) holds completed instr’s until they can commit
  - Supports precise interrupts and speculative execution with in-order **commit** out of the RUU
  - Basis of SimpleScalar’s datapath architecture

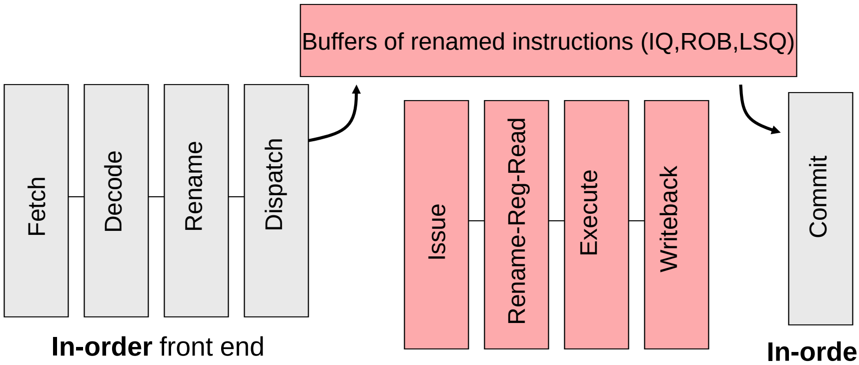
Basic OOO Instruction Flow Overview

1. Fetch (in program order): Fetch multiple sequential instructions in parallel from the IM (I$)
2. Decode, Rename, & Dispatch (in program order):
   - In parallel, decode all of the instr’s just fetched, rename the architected registers (ArchitectedRegFile) with rename registers (PhysicalRegFile), and schedule renamed instr’s for execution by dispatching them to the IQ (Instruction Queue) and the ROB (ReOrder Buffer) (combined in the RUU in SimpleScalar)
   - Loads and stores are dispatched as two (micro)instr’s – one to the IQ to compute the addr and one to LSQ (LoadStoreQueue) for the memory operation
3. Issue (Out Of Order - OOO): When an instr in the IQ has all of its source data and the FU (Functional Unit) it needs is free, it is issued for execution

Basic OOO Instruction Flow Overview, Con’t

4. Writeback (OOO): When the dst value has been computed it is written back to the PhysicalRegFile, the IQ, ROB and LSQ are updated – the instr completes
5. Commit (in program order): Only commit the instr’s result data to the state locations (i.e., update DM (D$), ArchitectedRegFile) when it is the oldest completed instr in the ROB

Out-of-Order Pipeline



Buffers of renamed instructions (IQ,ROB,LSQ)

Fetch

Decode

Rename

Dispatch

Issue

Rename-Reg-Read

Execute

Writeback

Commit

In-order front end

Out-of-order execution

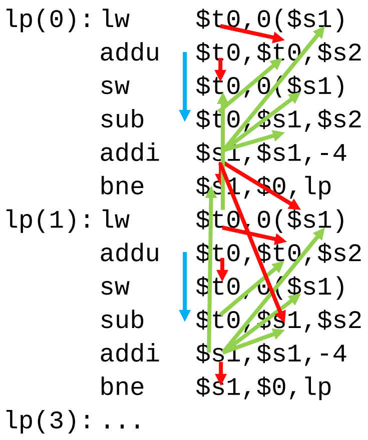
In-order commit

Our Code Example

```
lp(0): lw     $t0,0($s1)   #cache miss, 3 cycle stall
       addu   $t0,$t0,$s2
       sw     $t0,0($s1)
       sub    $t0,$s1,$s2  #provides WAW hazard
       addi   $s1,$s1,-4
       bne    $s1,$0,lp    #predict taken (and is)
lp(1): lw     $t0,0($s1)   #cache hit (from here on)
       addu   $t0,$t0,$s2
       sw     $t0,0($s1)
       sub    $t0,$s1,$s2
       addi   $s1,$s1,-4
       bne    $s1,$0,lp
lp(3): ...
```

Code Dependency Observations

- Lots of both true and false dependencies
- sub instr independent of other instr’s (has no true dependencies)
  - So can execute in parallel with another instr
  - Are there others?
- Registers re-used
  - Just as in static SS, the register names get in the way
  - How can the hardware get around this?



```
lp(0): lw     $t0,0($s1)
       addu   $t0,$t0,$s2
       sw     $t0,0($s1)
       sub    $t0,$s1,$s2
       addi   $s1,$s1,-4
       bne    $s1,$0,lp
lp(1): lw     $t0,0($s1)
       addu   $t0,$t0,$s2
       sw     $t0,0($s1)
       sub    $t0,$s1,$s2
       addi   $s1,$s1,-4
       bne    $s1,$0,lp
lp(3): ...
```

Register Renaming

- ❑ Can use register renaming to eliminate (WAW, WAR) (register) data dependencies – conceptually write each register once
  - + Removes **false** dependences (WAW and WAR)
  - + Leaves **true** dependences (RAW) intact
- ❑ “Architected” vs “Physical” registers
  - ● Architected (ISA) register names: **`$t0,$s1,$s1,$s2,`** etc
  - ● Physical register names: **`p1,p2,p3,p4,p5,p6,p7`**
- ❑ Need two hardware structures to enable renaming
  - ● A Map Table showing the architected register that the physical register is currently “impersonating”
  - ● A Free List of physical registers not currently in use
- ❑ When can a physical register be put back on the Free List?

Which Register to Free at Commit ?

- ❑ The over-written (physical) register can be freed at Commit (i.e., added back to the Free List), so we have to keep track of it during Rename

- ❑ We also need to keep track of the over-written (physical) register so that it can be restored in the Map Table on a recovery from mis-predicted branches and recovery from exceptions

Renaming Example: Initial State

```
lw    $t0,0($s1)
addu  $t0,$t0,$s2
sw    $t0,0($s1)
sub   $t0,$s1,$s2
addi  $s1,$s1,-4
bne   $s1,$0,lp
```

$s1	p1
$s2	p2
$t0	p3

Map Table

p4
p5
p6
p7
p8

Free List

Renaming Example: `lw` Renaming

Over-written Reg

```
lw    $t0,0($s1)  →
addu  $t0,$t0,$s2
sw    $t0,0($s1)
sub   $t0,$s1,$s2
addi  $s1,$s1,-4
bne   $s1,$0,lp
```

$s1	p1
$s2	p2
$t0	p3

Map Table

p4
p5
p6
p7
p8

Free List

Renaming Example: addu Renaming

Over-written Reg

```
lw    $t0,0($s1)
addu  $t0,$t0,$s2 →
sw    $t0,0($s1)
sub   $t0,$s1,$s2
addi  $s1,$s1,-4
bne   $s1,$0,lp
```

$s1	p1
$s2	p2
$t0	p4

Map Table

p5

p6

p7

p8

Free List

Renaming Example: sw Renaming

Over-written Reg

```
lw    $t0,0($s1)
addu  $t0,$t0,$s2
sw    $t0,0($s1)  →
sub   $t0,$s1,$s2
addi  $s1,$s1,-4
bne   $s1,$0,lp
```

$s1	p1
$s2	p2
$t0	p5

Map Table

p6

p7

p8

Free List

Renaming Example: `sub` Renaming

Over-written Reg

```
lw    $t0,0($s1)
addu  $t0,$t0,$s2
sw    $t0,0($s1)
sub   $t0,$s1,$s2 →
addi  $s1,$s1,-4
bne   $s1,$0,lp
```

$s1	p1
$s2	p2
$t0	p5

Map Table

p6
p7
p8

Free List

Renaming Example: `addi` Renaming

Over-written Reg

```
lw    $t0,0($s1)
addu  $t0,$t0,$s2
sw    $t0,0($s1)
sub   $t0,$s1,$s2
addi  $s1,$s1,-4  →
bne   $s1,$0,lp
```

$s1	p1
$s2	p2
$t0	p6

Map Table

p7
p8

Free List

Renaming Example: bne Renaming

Over-written Reg

```
lw    $t0,0($s1)        lw    p4,0(p1)      [p3]
addu  $t0,$t0,$s2       addu  p5,p4,p2      [p4]
sw    $t0,0($s1)        sw    p5,0(p1)
sub   $t0,$s1,$s2       sub   p6,p1,p2      [p5]
addi  $s1,$s1,-4        addi  p7,p1,-4      [p1]
bne   $s1,$0,lp    →    bne   p7,p0,lp
```

$s1	p7
$s2	p2
$t0	p6

Map Table

p8

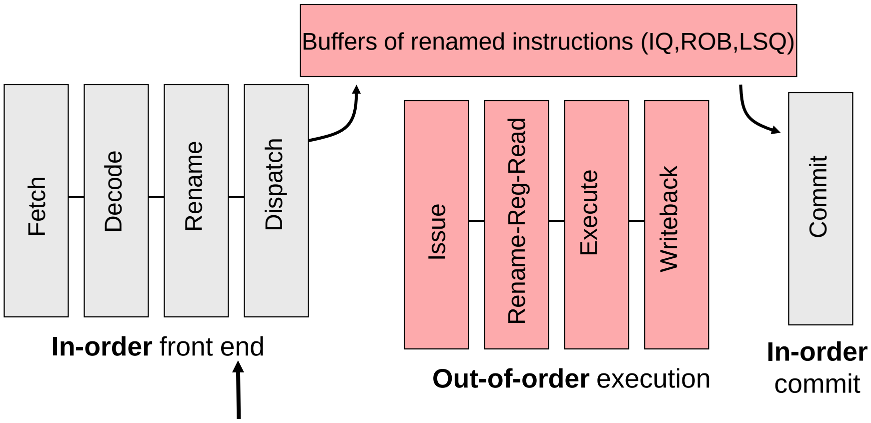
Free List

Our Code Example After Renaming

```
lp(0): lw    p4,0(p1)     #[p3];cache miss, 3 cycle stall
       addu  p5,p4,p2     #[p4]
       sw    p5,0(p1)
       sub   p6,p1,p2     #[p5]
       addi  p7,p1,-4     #[p1]
       bne   p7,p0,lp     #predict taken (and is)
lp(1): lw    p8,0(p7)     #[p6];cache hit
       addu  p9,p8,p2     #[p8]
       sw    p9,0(p7)
       sub   p10,p7,p2    #[p9]
       addi  p11,p7,-4    #[p7]
       bne   p11,p0,lp
lp(3): ...
```

Out-of-Order Pipeline Progress

- Have completed Fetch, Decode, Rename (in program order) and are ready to Dispatch



Buffers of renamed instructions (IQ,ROB,LSQ)

Fetch

Decode

Rename

Dispatch

Issue

Rename-Reg-Read

Execute

Writeback

Commit

In-order front end

Out-of-order execution

In-order commit

Instr's now have unique register names, so can now put into OOO execution structures

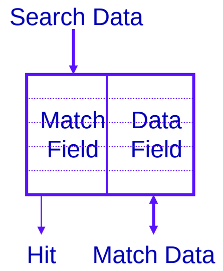
Dispatch

- Renamed instructions are placed into OOO hardware data structures

1. Issue Queue (IQ) (SimpleScalar's RUU)
   - Central piece of scheduling logic holding un-executed instr's
   - Accessible as both a RAM and a CAM (Content Addr Memory)
2. Re-order buffer (ROB) (SimpleScalar's RUU)
   - Holds all instructions (in order) until Commit time
   - Keeps track of the over-written register so they can be returned to the free list and to support recovery from mispredicted branches and exceptions
3. Load-Store Queue (LSQ)
   - Loads and stores dispatched in two parts - one going to the IQ for effective address calculation and the other to the LSQ for loads and stores going to the DM
   - Stores not sent to DM until Commit time, what about loads ?

Aside: Content Addressable Memories (CAMs)

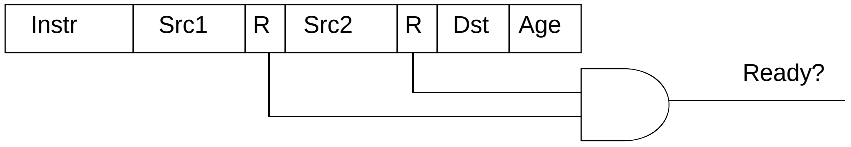
- ❑ Storage hardware that is addressed by its content. Typical applications include IQ source tag fields comparison logic, cache tags, and TLBs (translation lookaside buffers)
  - ● Hardware that compares the Search Data to the Match Field entries for *each* word in the CAM in *parallel* !
  - ● On a match the Hit bit is set and the Data Field for that entry is output to Match Data on read or the Match Data is written into the Data Field on write
  - ● If no match occurs, the Hit bit is reset
  - ● CAMs can be designed to accommodate multiple hits



- ❑ A storage structure can have ports of both types (RAM & CAM)

Issue Queue (IQ)

- ❑ Holds un-executed instructions
  - ● Instruction op and instruction "age"
- ❑ Tracks status of source inputs (ready, not ready)
  - ● Physical (renamed) source register names + a ready bit for each source operand
    - AND the ready bits to tell if the instruction is ready to issue (send for execution)
- ❑ Physical (renamed) destination register



Dispatch Steps

- Allocate IQ (and ROB) slot
  - Full? Stall
  - Not full? Find an empty slot in the IQ
- Read **ready bits** of inputs (source registers) from a Ready Table
  - Ready Table: 1-bit per physical register indicating whether or not that physical register value has been produced
- Clear **ready bit** of output (destination register) in Ready Table
  - Instruction has not produced value yet
- Write instruction data in the allocated IQ slot
- Recall that `lw` and `sw` go into both the IQ (for computing the effective address) and the LSQ (which interfaces with the DM)

Dispatch Example, `lw` Dispatch

```
lp(0): lw    p4,0(p1)    #[p3]
       addu  p5,p4,p2    #[p4]
       sw    p5,0(p1)
       sub   p6,p1,p2    #[p5]
       addi  p7,p1,-4    #[p1]
       bne   p7,p0,lp    #
```

Issue Queue

Instr	Src1	R	Src2	R	Dst	Age
lw		**y**	**0+p1**	**y**	**p4**	0

Ready Table

p0	y
p1	y
p2	y
p3	y
p4	**n**
p5	y
p6	y
p7	y
p8	y
p9	y

Dispatch Example, addu Dispatch

```
lp(0): lw    p4,0(p1)    #[p3]
       addu  p5,p4,p2    #[p4]
       sw    p5,0(p1)
       sub   p6,p1,p2    #[p5]
       addi  p7,p1,-4    #[p1]
       bne   p7,p0,lp    #
```

Issue Queue

Instr	Src1	R	Src2	R	Dst	Age
lw		y	0+p1	y	p4	0

Ready Table

p0	y
p1	y
p2	y
p3	y
p4	n
p5	y
p6	y
p7	y
p8	y
p9	y

Dispatch Example, sw Dispatch

```
lp(0): lw    p4,0(p1)    #[p3]
       addu  p5,p4,p2    #[p4]
       sw    p5,0(p1)
       sub   p6,p1,p2    #[p5]
       addi  p7,p1,-4    #[p1]
       bne   p7,p0,lp    #
```

Issue Queue

Instr	Src1	R	Src2	R	Dst	Age
lw		y	0+p1	y	p4	0
addu	p4	n	p2	y	p5	1

Ready Table

p0	y
p1	y
p2	y
p3	y
p4	n
p5	n
p6	y
p7	y
p8	y
p9	y

Dispatch Example, `sub` Dispatch

```
lp(0): lw     p4,0(p1)      #[p3]
       addu   p5,p4,p2      #[p4]
       sw     p5,0(p1)
       sub    p6,p1,p2      #[p5]
       addi   p7,p1,-4      #[p1]
       bne    p7,p0,lp      #
```

Issue Queue

Instr	Src1	R	Src2	R	Dst	Age
lw		y	0+p1	y	p4	0
addu	p4	n	p2	y	p5	1
sw	p5	n	0+p1	y		2

Ready Table

p0	y
p1	y
p2	y
p3	y
p4	n
p5	n
p6	y
p7	y
p8	y
p9	y

Dispatch Example, `addi` Dispatch

```
lp(0): lw     p4,0(p1)      #[p3]
       addu   p5,p4,p2      #[p4]
       sw     p5,0(p1)
       sub    p6,p1,p2      #[p5]
       addi   p7,p1,-4      #[p1]
       bne    p7,p0,lp      #
```

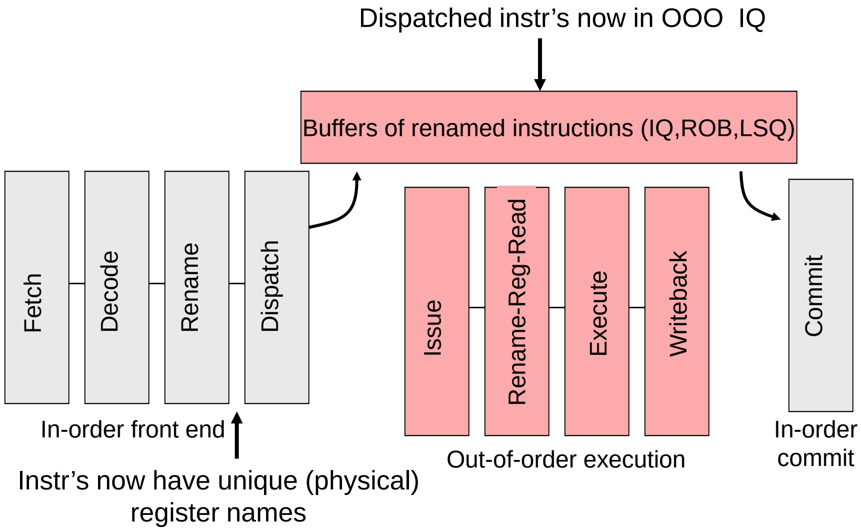
Issue Queue

Instr	Src1	R	Src2	R	Dst	Age
lw		y	0+p1	y	p4	0
addu	p4	n	p2	y	p5	1
sw	p5	n	0+p1	y		2
sub	p1	y	p2	y	p6	3

Ready Table

p0	y
p1	y
p2	y
p3	y
p4	n
p5	n
p6	n
p7	y
p8	y
p9	y

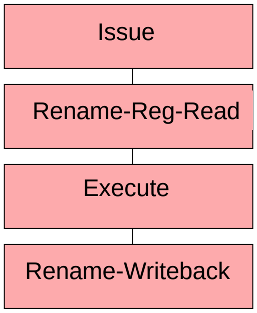
Out-of-Order Pipeline Progress


Dispatched instr's now in OOO IQ
Buffers of renamed instructions (IQ,ROB,LSQ)
Fetch
Decode
Rename
Dispatch
Issue
Rename-Reg-Read
Execute
Writeback
Commit
In-order front end
Out-of-order execution
In-order commit
Instr's now have unique (physical) register names

Out-of-Order Execution Pipeline Stages

- Execution (out-of-order) stages
- Issue

1. **Select** ready instructions
   - Send (Issue) them for execution
2. **Wakeup** dependent instructions in the IQ



- OOO execution pipeline has necessary forwarding hardware and multiple FU's of different types (some of them with multiple pipeline stages)
- Remember, read and writeback are from/to the Physical (rename) RegFile

Issue = Select + Wakeup

- **Select** N oldest, ready instr's to send for execution (checking for structural hazards (e.g., FUs))
  - Assume `lw` has already been issued to memory and it's 3 cycle cache miss is still pending
  - sub and addi are the two oldest ready instr's

Issue Queue (IQ)

	Instr	Src1	R	Src2	R	Dst	Age
Issued	lw		y	0+p1	y	p4	0
	addu	p4	n	p2	y	p5	1
	sw	p5	n	0+p1	y		2
	sub	p1	y	p2	y	p6	3
	addi	p1	y	-4	y	p7	4
	bne	p7	n	p0	y		5

Ready Table

p0	y
p1	y
p2	y
p3	y
p4	n
p5	n
p6	n
p7	n
p8	y
p9	y

Issue = Select + Wakeup

- **Wakeup** dependent instr's
  - CAM search for dst addr in **Src1** and **Src2** and set ready bit (**R**) on match
  - Update Ready Table for Dispatch of future instr's

Assoc Search for p6 and p7 ↓

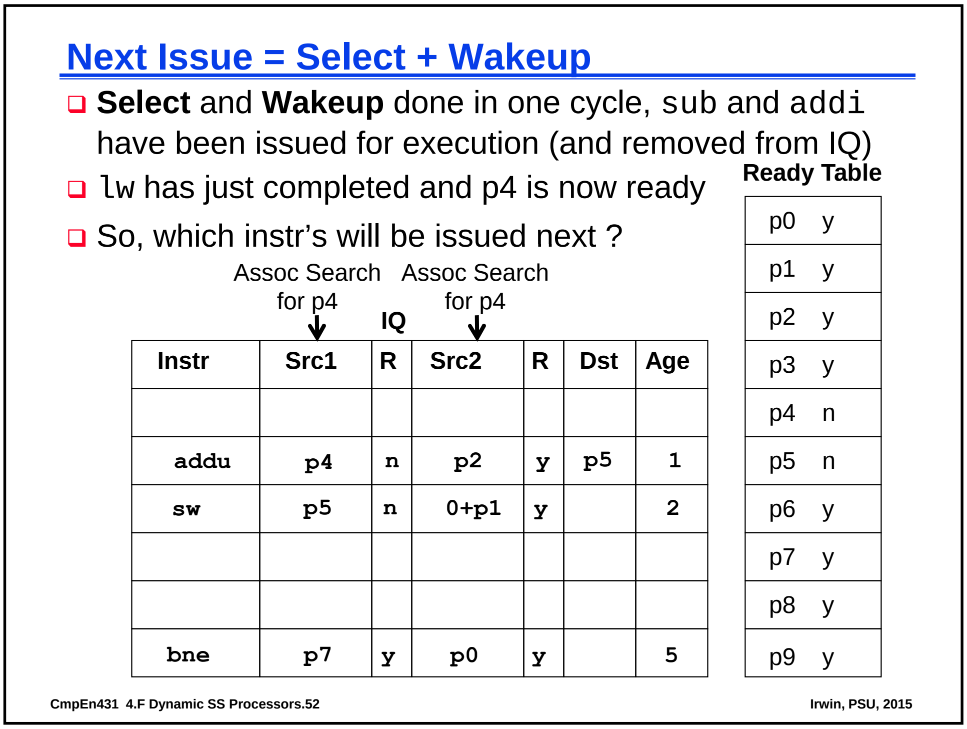
IQ

Assoc Search for p6 and p7 ↓

	Instr	Src1	R	Src2	R	Dst	Age
Issued	lw		y	0+p1	y	p4	0
	addu	p4	n	p2	y	p5	1
	sw	p5	n	0+p1	y		2
Ready!	sub	p1	y	p2	y	p6	3
Ready!	addi	p1	y	-4	y	p7	4
	bne	p7	y	p0	y		5

Ready Table

p0	y
p1	y
p2	y
p3	y
p4	n
p5	n
p6	**y**
p7	**y**
p8	y
p9	y



Next Issue = Select + Wakeup

- ❑ **Select** and **Wakeup** done in one cycle, `sub` and `addi` have been issued for execution (and removed from IQ)
- ❑ `lw` has just completed and p4 is now ready
- ❑ So, which instr's will be issued next ?

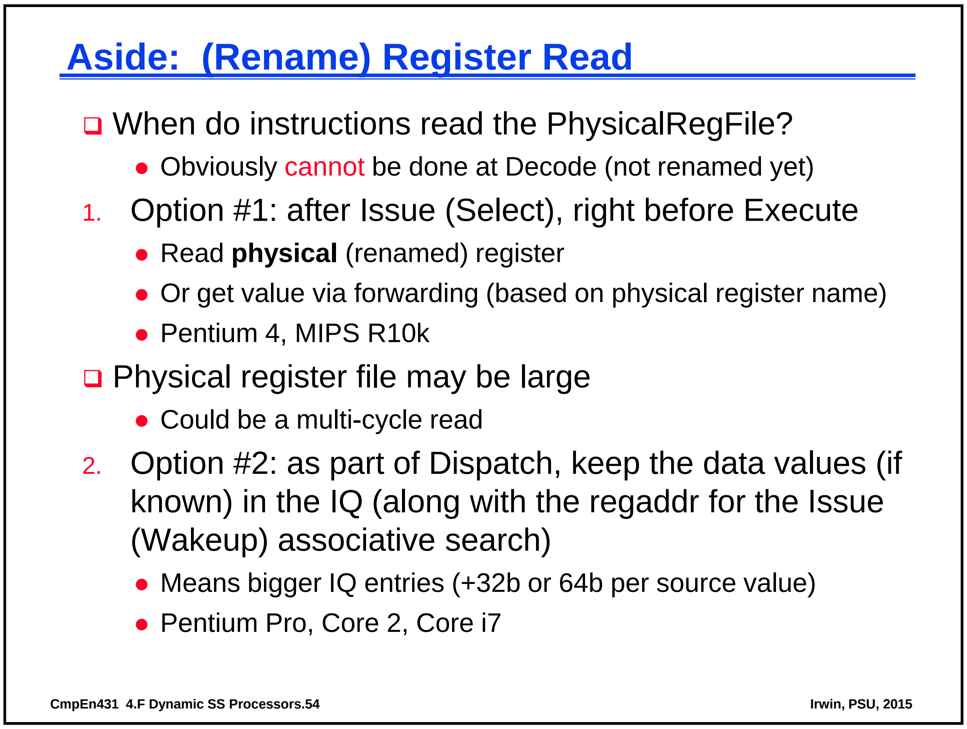
Assoc Search for p4 ↓   Assoc Search for p4 ↓

IQ

Instr	Src1	R	Src2	R	Dst	Age
`addu`	`p4`	`n`	`p2`	`y`	`p5`	`1`
`sw`	`p5`	`n`	`0+p1`	`y`		`2`
`bne`	`p7`	`y`	`p0`	`y`		`5`

Ready Table

p0	y
p1	y
p2	y
p3	y
p4	n
p5	n
p6	y
p7	y
p8	y
p9	y



Aside: (Rename) Register Read

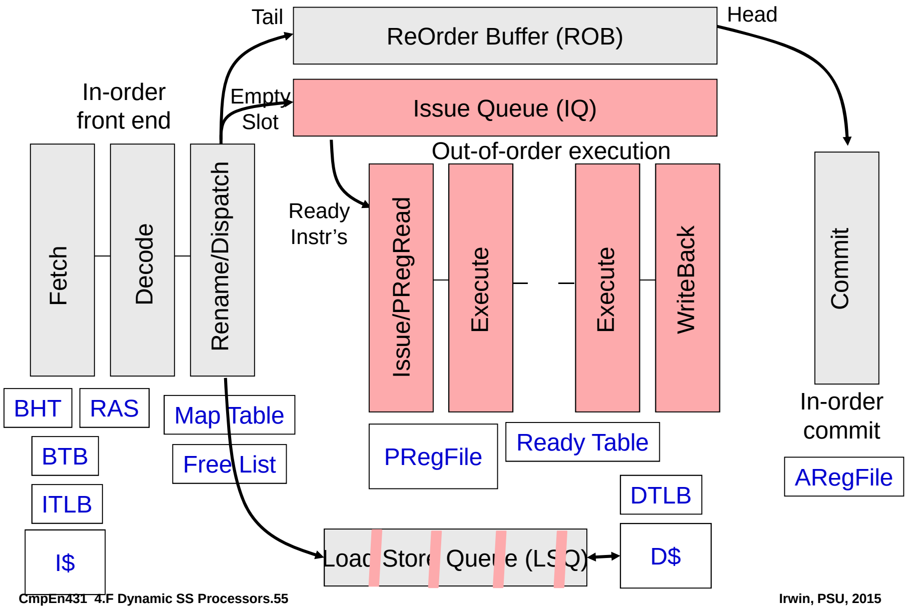
- ❑ When do instructions read the PhysicalRegFile?
  - Obviously cannot be done at Decode (not renamed yet)

1. Option #1: after Issue (Select), right before Execute
   - Read **physical** (renamed) register
   - Or get value via forwarding (based on physical register name)
   - Pentium 4, MIPS R10k

- ❑ Physical register file may be large
  - Could be a multi-cycle read

2. Option #2: as part of Dispatch, keep the data values (if known) in the IQ (along with the regaddr for the Issue (Wakeup) associative search)
   - Means bigger IQ entries (+32b or 64b per source value)
   - Pentium Pro, Core 2, Core i7

Out-of-order Pipeline – The Detailed View



Re-Order Buffer (ROB)

- All instructions Commit in order
  - At commit write the Physical (rename) register value to the Architected (ISA) register and free the **overwritten** Physical register (add it back to the Free List), for store instr's write the data in the LSQ to the D$ (more on this soon), and free the LSQ and ROB entries for reuse
- Two other purposes
  - To support recovery from branch mis-prediction, load mis-speculation and to support precise (synchronous) interrupts
    - Flush the ROB, IQ, and LSQ, restore Map Table and Ready Table to before mis-prediction/interrupt, and free the Physical registers (update Free List) (wasted time, wasted power – why accurate branch prediction is **sooo** important for OOO datapaths (not as bad for interrupts since they are relatively infrequent)) and …
    - On mis-predicted branch at ROB **head**, update BHT, BTB, restart the pipeline at the branch (with the correct prediction this time)
    - On interrupt of instruction at ROB **head**, service the interrupt, restart the pipeline at the interrupting instr

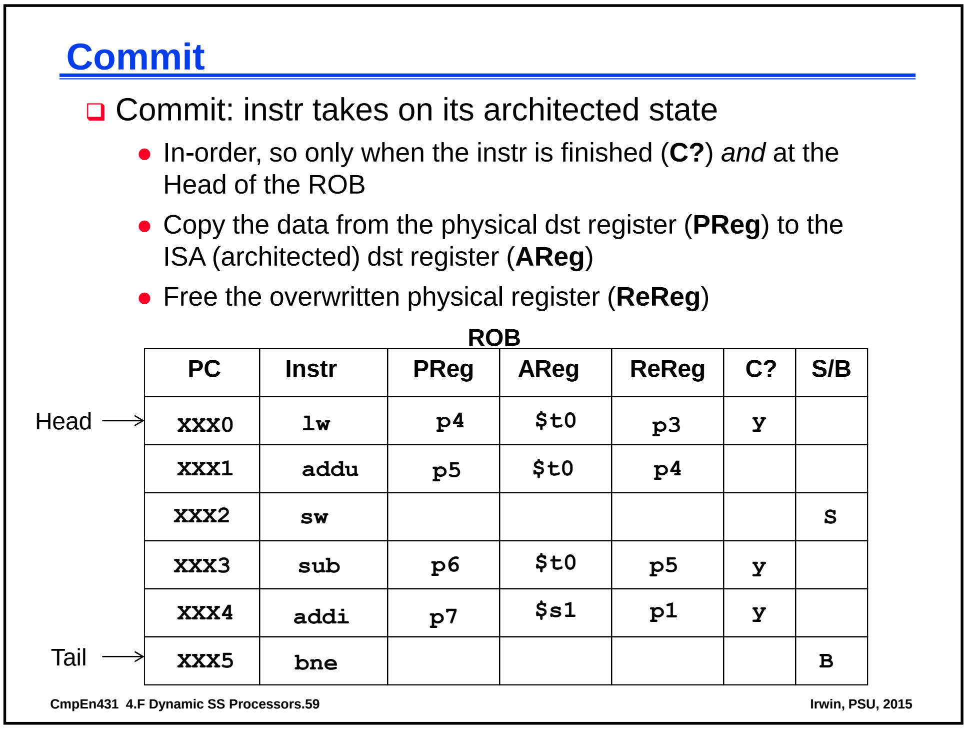
Speculation in OOO Machines

- Speculation allows execution of future instr's that (may) depend on the speculated instruction
  - Speculate on the outcome of a conditional branch (branch prediction) just don't **commit** until the branch outcome is known
  - Speculate that a store (for which we don't yet know the address) that precedes a load does not refer to the same address, allowing the load to be done before the store (load speculation) once again not **committing** the load until the speculation has cleared
- Must have hardware mechanisms for
  - Checking to see if the guess was correct
  - Recovering from incorrect speculation – only **commit** out of the ROB when are sure speculation is correct
- Ignore and/or buffer exceptions created by speculatively executed instructions until it is clear that they should really occur (i.e., not allowed to change the machine state until **commit** time)

Re-Order Buffer (ROB) Data

- ROB entry has to keep track of all of the info needed for Commit & Recover
  - Physical (renamed) dst register addr and its architectural (ISA) equivalent (so can update ARegFile on completion)
  - Overwritten physical register name (for release and recovery)
  - Instruction address (PC) and type (in particular store, branch)
  - A way of determining when the instr completes execution
  - Exception (interrupt) and branch outcome information
- On Dispatch: insert at tail
  - Full? Stall
- Commit: remove from head
  - Instr at head not completed? No instr to commit this cycle
  - Multiple instr's at head completed … commit multiple instr's this cycle (if have hardware to support it)

http://www.ecs.umass.edu/ece/koren/architecture/ROB/rob_simulator.htm

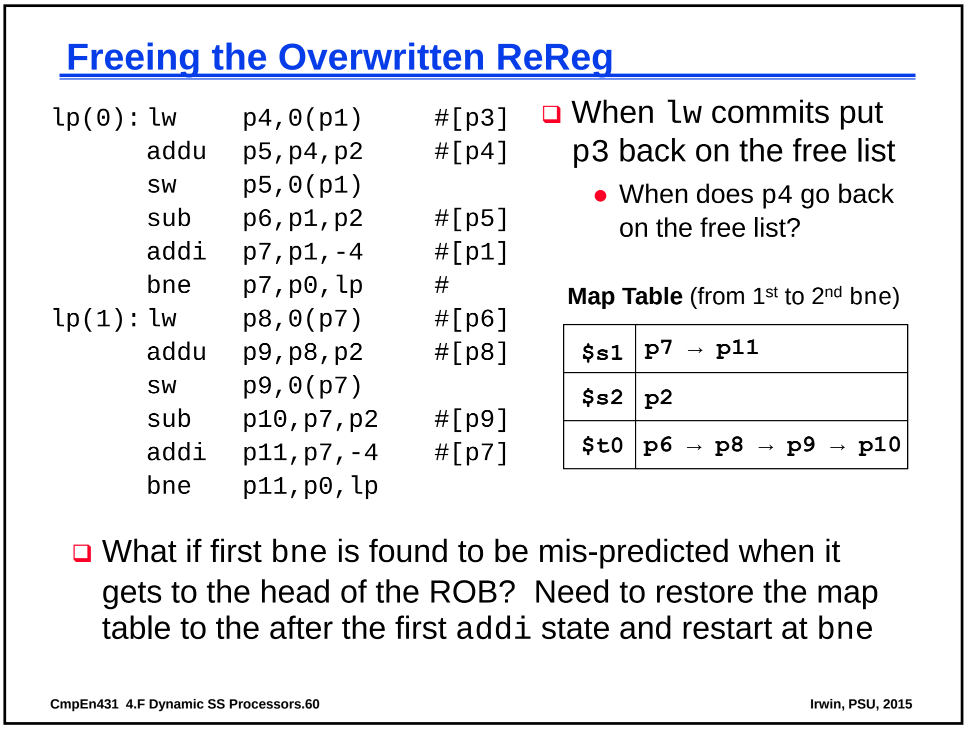


Commit

- Commit: instr takes on its architected state
  - In-order, so only when the instr is finished (**C?**) *and* at the Head of the ROB
  - Copy the data from the physical dst register (**PReg**) to the ISA (architected) dst register (**AReg**)
  - Free the overwritten physical register (**ReReg**)

ROB

	PC	Instr	PReg	AReg	ReReg	C?	S/B
Head →	xxx0	lw	p4	$t0	p3	y	
	xxx1	addu	p5	$t0	p4		
	xxx2	sw					S
	xxx3	sub	p6	$t0	p5	y	
	xxx4	addi	p7	$s1	p1	y	
Tail →	xxx5	bne					B



Freeing the Overwritten ReReg

```
lp(0): lw     p4,0(p1)      #[p3]
       addu   p5,p4,p2      #[p4]
       sw     p5,0(p1)
       sub    p6,p1,p2      #[p5]
       addi   p7,p1,-4      #[p1]
       bne    p7,p0,lp      #
lp(1): lw     p8,0(p7)      #[p6]
       addu   p9,p8,p2      #[p8]
       sw     p9,0(p7)
       sub    p10,p7,p2     #[p9]
       addi   p11,p7,-4     #[p7]
       bne    p11,p0,lp
```

- When `lw` commits put p3 back on the free list
  - When does p4 go back on the free list?

Map Table (from 1st to 2nd bne)

$s1	p7 → p11
$s2	p2
$t0	p6 → p8 → p9 → p10

- What if first bne is found to be mis-predicted when it gets to the head of the ROB? Need to restore the map table to the after the first `addi` state and restart at bne

Load Store Queue (LSQ)

- Loads and stores are dispatched to the IQ, to the LSQ (the interface to the DM), and to the ROB
  - When ready, loads and stores are issued (for effective address calculation) and their IQ entries are released
  - When the effective address or store source has been calculated, it is compared to find the matching EAddr / Src entries in the LSQ

Assoc Search for p5

Assoc Search for 0+p7

LSQ

	Instr	Src	R	EAddr	R	Dst	Age
Issued	lw (1)		y	0+p1	y	p4	0
Ready!	sw (1)	p5	y	0+p1	y		2
Ready!	lw (2)		y	0+p7	y	p8	6
	sw (2)	p9	n	0+p7	y		8

Memory Location Data Dependencies

- RAW, WAR and WAW memory data dependencies
  - Memory storage conflicts are less frequent since memory locations are not used (and reused) in the same way that registers are
- Stores are committed to the DM from the LSQ **in program order** at commit time (when they are at the head of the ROB); since stores commit **in order** there are **no** WAW hazards. There are also **no** WAR hazards since there are also no older loads (they have already been committed).
  - RAW, true dependence (cannot reorder, what to do?)

```
sw    $t0,0($s1)
lw    $t1,0($s1)
```

  - WAR, anti-dependence (write commit in order **fixes**, `lw` will have already been committed)

```
lw    $t0,0($s1)
sw    $t1,0($s1)
```

  - WAW, memory output dependence (write commit in order **fixes**)

```
sw    $t0,0($s1)
sw    $t1,0($s1)
```

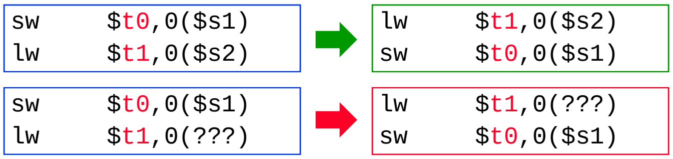
Loads from Memory

- When an issued load instruction completes execution
  1. the load data from memory is written to the PRegFile and the Ready Table is updated
  2. the load source register addr is compared (associatively) to see if it matches the Src addr's of instr's in the IQ and the Src addr of stores in the LSQ
  3. the load's LSQ entry is released

- Note that the oldest load is issued for execution out of the LSQ to the DM
  - If there is a EAddr match with another (younger) load, that younger load may not need to be executed since the current load **may** have loaded the data the younger load needs
  - However, it there is an intervening store between the issuing load and a younger load **all** with the same effective address then the store has the data the younger load needs (store->load forwarding)

Load Bypassing

- For better performance younger loads can bypass (be issued before) older loads and stores in the LSQ under certain conditions
  - Loads bypassing stores - Ready loads **can** bypass previous (older) stores as long as their effective addresses are known and different (so there is no RAW hazard)



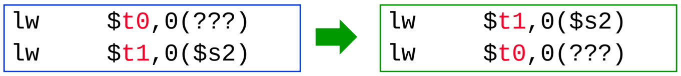
```
sw      $t0,0($s1)
lw      $t1,0($s2)
```

```
lw      $t1,0($s2)
sw      $t0,0($s1)
```

```
sw      $t0,0($s1)
lw      $t1,0(???)
```

```
lw      $t1,0(???)
sw      $t0,0($s1)
```

- Loads bypassing loads - Ready (EAddr has been calculated) loads in the LSQ **can** bypass previous (older) unready loads
  - What if they are to the same EAddr? Who cares, no harm done.



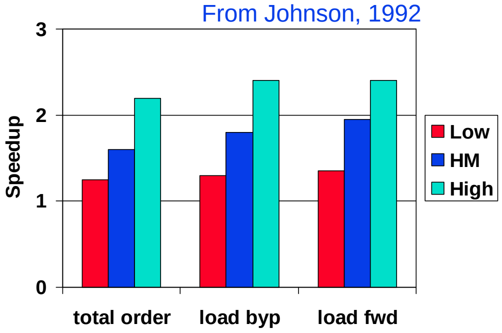
```
lw      $t0,0(???)
lw      $t1,0($s2)
```

```
lw      $t1,0($s2)
lw      $t0,0(???)
```

Load Bypassing with Load Forwarding

- Load forwarding – when a load's data is supplied directly from an older store in the LSQ
  - The most recent older matching LSQ store data value is supplied to the load (beware! there could be more than one matching store)
- Load bypassing gives 19% speedup improvement (for a 4-way OOO datapath)
- Load forwarding gives an additional 4% speedup improvement

From Johnson, 1992



Stores to Memory

- Stores are held in the LSQ until the store is ready to commit (in program order - when the store is at the head of the ROB); on Commit the LSQ and ROB entries are released
- In addition to the associative search for matching EAddr's, when a PReg becomes ready that address is compared (associatively) with the LSQ's Src fields (stores' data value PReg addresses)
  - If there is **also** an effective addr match (EAddr) in the LSQ with a load and the stores data is ready, then the store can provide the load's dst data **if** the store is the most recent store older than the load (again, store->load forwarding)

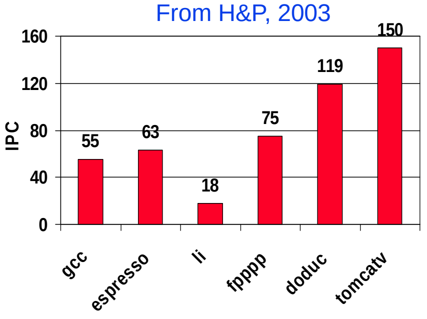
OOO Scheduling Scope (Exposing More ILP)

- Scheduling scope = OOO window size
  - Larger = better
  1. Constrained by the number of physical registers (PhyRegFile)
     - ROB roughly limited by the number of physical registers
     - Big register file = expensive (area) and slow
  2. Constrained by size of Issue Queue
     - Limits number of un-executed instructions
     - CAMs = can't make too big (power + area)
  3. Constrained by size of Load+Store Queue
     - Limits number of loads/stores
     - CAMs = can't make too big (power + area)
- Usefulness of large window: limited by branch prediction
  - 95% branch mis-prediction rate: 1 in 20 branches, or 1 in 100 instr's

ILP in a "Perfect" Dynamic SS Datapath

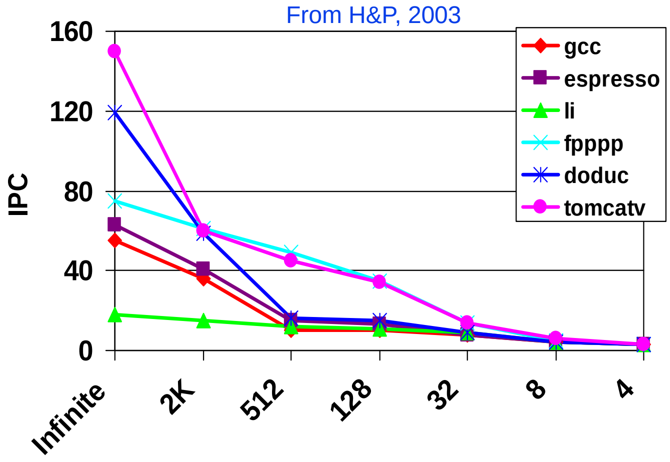
- The perfect dynamic SS datapath has
  - An infinite number of rename registers that eliminates all WAR, WAW data hazards
  - Infinite IQ, LSQ, and ROB (so never full)
  - No (fetch, decode, dispatch, issue, FU, buses, ports) limit on the number of instr's that can begin execution simultaneously (as long as RAW (true) data hazards are not present)
  - Perfect branch prediction
  - Perfect caches
  - Loads can be moved before stores as long as there are no RAW data hazards
  - All FU's have a 1 cycle latency

From H&P, 2003



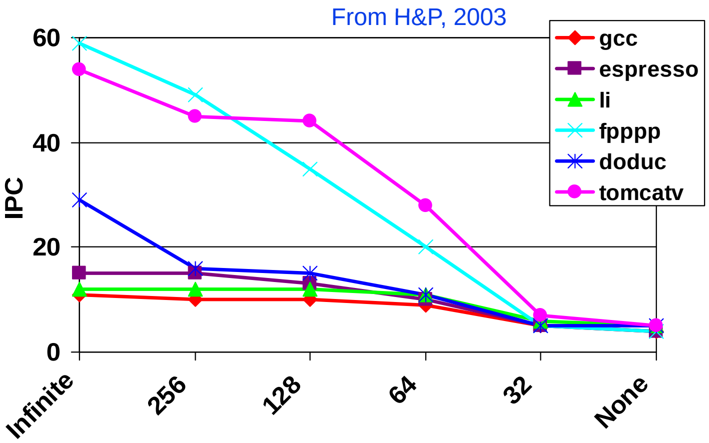
Effect of IQ size on ILP

- Instruction window (IQ) – the set of instructions that are examined simultaneously for execution


From H&P, 2003

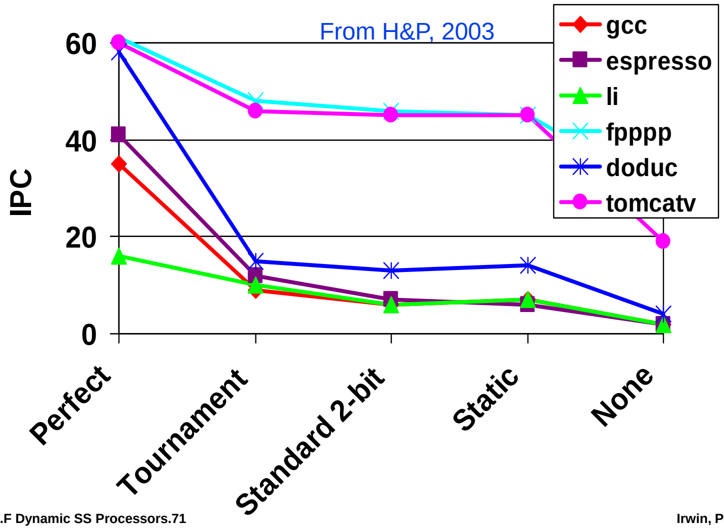
Effect of Finite Rename Registers

- On a processor with an IQ of 2K, a maximum 64-way issue capability, and a tournament branch predictor with 8K entries


From H&P, 2003

Effect of Realistic Branch Prediction on ILP

- On a processor with an IQ of 2K and maximum 64-way issue capability


From H&P, 2003

Summary: Dynamic (OOO) Scheduling

- Dynamic scheduling
  - Totally in the hardware; compiler can help (e.g., loop unrolling)
- Fetch many instr's into instruction window
  - Use branch prediction to speculate past (multiple) branches
  - Flush pipeline queues on branch misprediction
- Rename to avoid false dependencies
- Execute instructions as soon as possible
  - Register dependencies are known
  - Handling memory dependencies more tricky
- Commit instr's in order
  - Anything strange happens before commit, just flush pipeline queues and restart
- Current machines: 100+ instruction scheduling window

Out Of Order: Top 5 Things to Know

1. Register renaming
   - How to perform it and how to recover it
2. Issue/Select
   - Wakeup: CAM
   - Choose N oldest ready instructions
3. Stores
   - Write at commit
   - Forward to loads via LSQ
4. Loads
   - Possibility for load bypassing and load forwarding
5. Commit
   - Precise state maintained in the ROB
   - How/when physical registers are freed

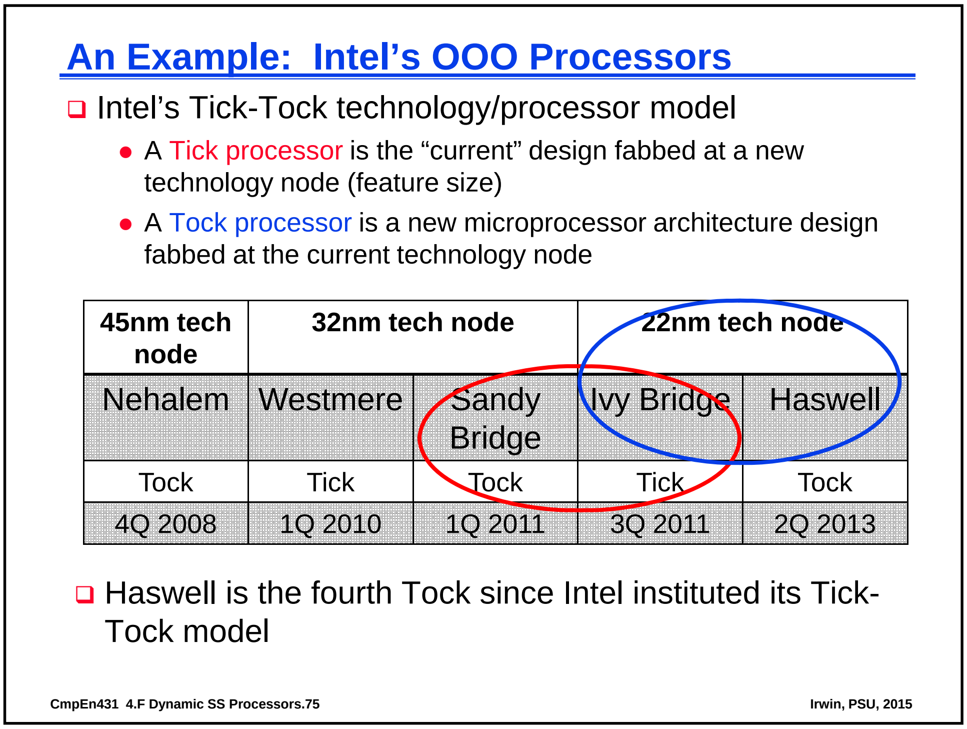
Power Costs of OOO Execution

❑ Complexity of dynamic scheduling and recovering from mis-speculation requires more power

❑ Multiple simpler cores may be better (power-wise)

- Power*Delay product may be a better measure

Microprocessor	Year	Clock Rate	Pipeline Stages	Issue width	Out-of-order/ Speculation	Cores	Power
i486	1989	25MHz	5	1	No	1	5W
Pentium	1993	66MHz	5	2	No	1	10W
Pentium Pro	1997	200MHz	10	3	Yes	1	29W
P4 Willamette	2001	2000MHz	22	3	Yes	1	75W
P4 Prescott	2004	3600MHz	31	3	Yes	1	103W
Core	2006	2930MHz	14	4	Yes	2	75W
Nehalem	2010	3300MHz	14	4	Yes	1	87W
Ivy Bridge	2012	3400MHz	14	4	Yes	8	77W

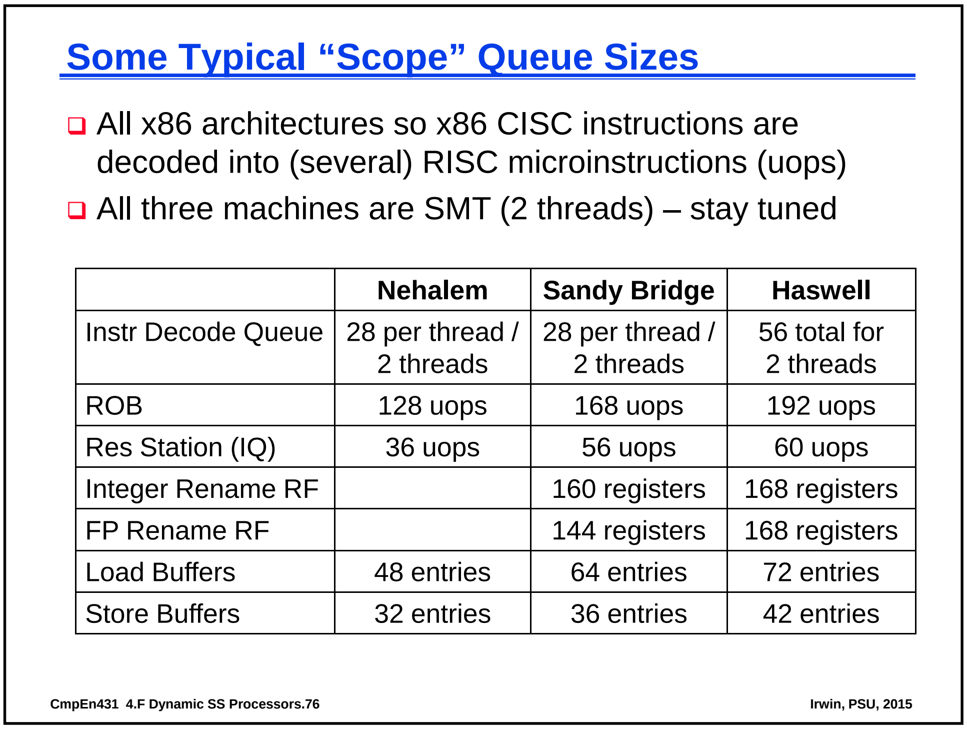


An Example: Intel's OOO Processors

- Intel's Tick-Tock technology/processor model
  - A Tick processor is the "current" design fabbed at a new technology node (feature size)
  - A Tock processor is a new microprocessor architecture design fabbed at the current technology node

45nm tech node	32nm tech node		22nm tech node	
Nehalem	Westmere	Sandy Bridge	Ivy Bridge	Haswell
Tock	Tick	Tock	Tick	Tock
4Q 2008	1Q 2010	1Q 2011	3Q 2011	2Q 2013

- Haswell is the fourth Tock since Intel instituted its Tick-Tock model

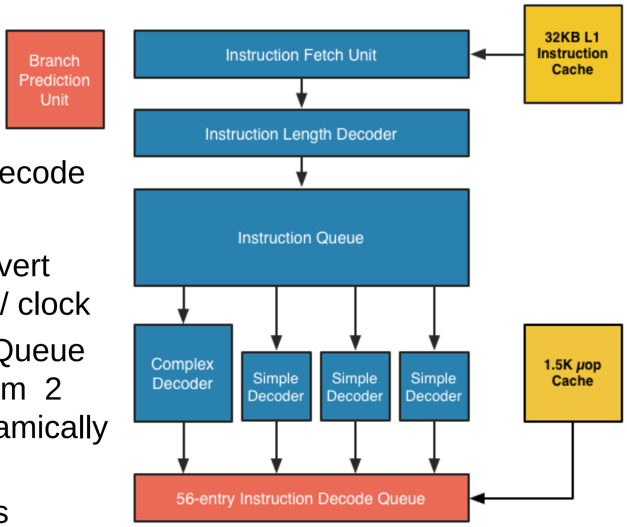


Some Typical "Scope" Queue Sizes

- All x86 architectures so x86 CISC instructions are decoded into (several) RISC microinstructions (uops)
- All three machines are SMT (2 threads) – stay tuned

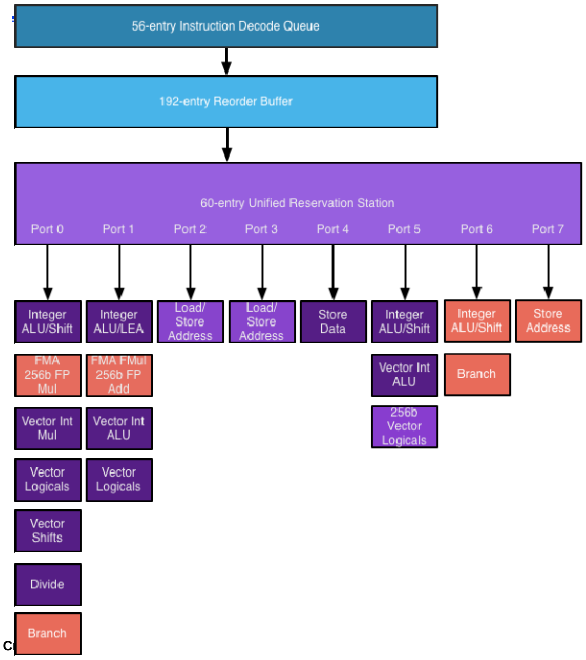
	Nehalem	Sandy Bridge	Haswell
Instr Decode Queue	28 per thread / 2 threads	28 per thread / 2 threads	56 total for 2 threads
ROB	128 uops	168 uops	192 uops
Res Station (IQ)	36 uops	56 uops	60 uops
Integer Rename RF		160 registers	168 registers
FP Rename RF		144 registers	168 registers
Load Buffers	48 entries	64 entries	72 entries
Store Buffers	32 entries	36 entries	42 entries

Intel Haswell Front End



- 4-wide fetch/decode
- 2-way SMT
- Decoders convert x86 to 4 uops / clock
- Instr Decode Queue holds uops from 2 threads – dynamically partitioned
- (Red is what is changed over Sandy Bridge)

Intel Haswell Execution Engine



- IQ work done by the Reservation Station
- 8 execution ports (only 6 on Sandy Bridge)

Haswell’s Cache Architecture

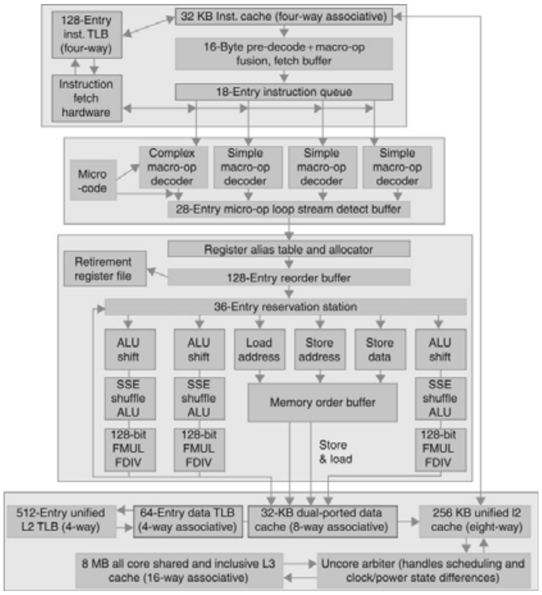
- ❑ All caches have 64B blocks; L1s and L2 private, L3 shared

Metric	Nehalem	Sandy Bridge	Haswell
L1 I$	32K, 4-way	32K, 8-way	32K, 8-way
L1 D$	32K, 8-way	32K, 8-way	32K, 8-way
Ld-to-use	4 cycles	4 cycles	4 cycles
Ld bdwdth	16B/cycle	32B/cycle (banked)	64B/cycle
St bdwdth	16B/cycle	16B/cycle	32B/cycle
UL2	256K, 8-way	256K, 8-way	256K, 8-way
Ld-to-use	10 cycles	11 cycles	11 cycles
Bdwdth L1	32B/cycle	32B/cycle	64B/cycle
L1 iTLB	128, 4-way	128, 4-way	128, 4-way
L1 dTLB	64, 4-way	64, 4-way	64, 4-way
L2 uTLB	512, 4-way	512, 4-way	1024, 8-way

Cortex A8 versus Intel i7

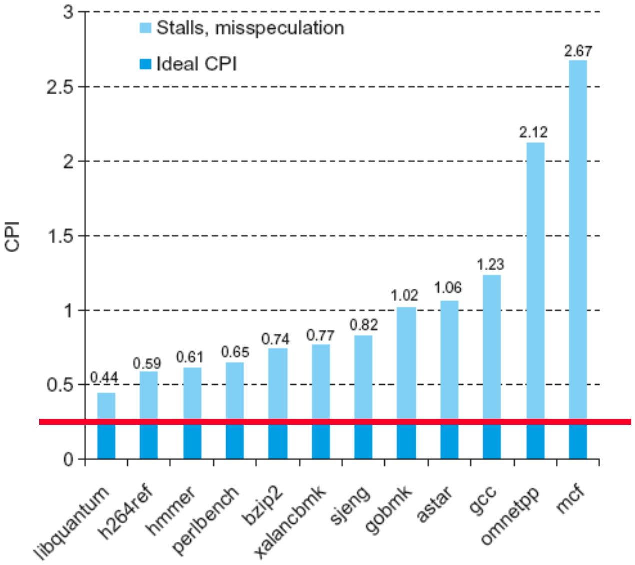
Processor	ARM A8	Intel Core i7 920
Market	Personal Mobile Device	Server, cloud
Thermal design power	2 Watts	130 Watts
Clock rate	1 GHz	2.66 GHz
Cores/Chip	1	4
Floating point?	No	Yes
Multiple issue?	Yes	Yes
Peak instructions/clock cycle	2	4
Pipeline stages	14	14
Pipeline schedule	Static in-order	Dynamic out-of-order with speculation
Branch prediction	2-level	2-level
1st level caches/core	32 KiB I, 32 KiB D	32 KiB I, 32 KiB D
2nd level caches/core	128-1024 KiB	256 KiB
3rd level caches (shared)	-	2- 8 MB

Core i7 Pipeline

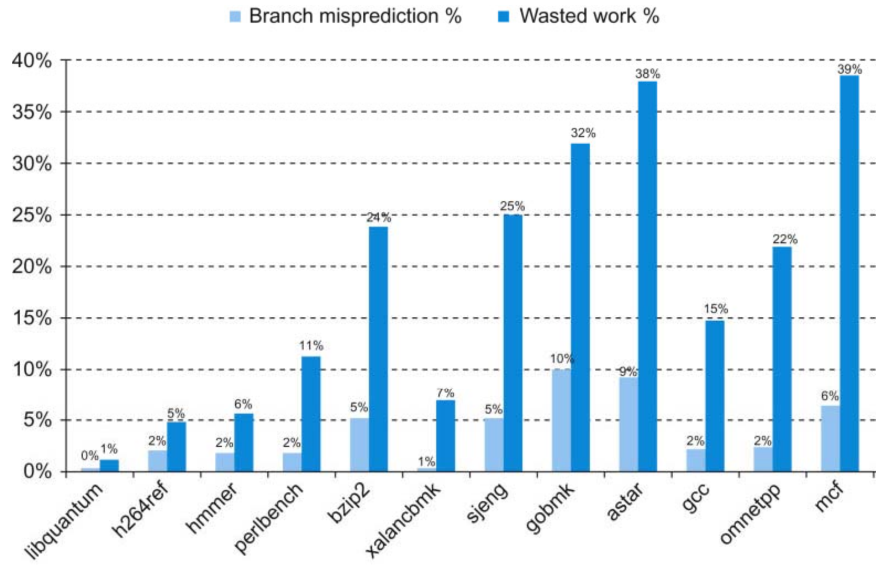


128-Entry inst. TLB (four-way)
32 KB Inst. cache (four-way associative)
16-Byte pre-decode + macro-op fusion, fetch buffer
Instruction fetch hardware
18-Entry instruction queue
Micro -code
Complex macro-op decoder
Simple macro-op decoder
Simple macro-op decoder
Simple macro-op decoder
28-Entry micro-op loop stream detect buffer
Register alias table and allocator
Retirement register file
128-Entry reorder buffer
36-Entry reservation station
ALU shift
ALU shift
Load address
Store address
Store data
ALU shift
SSE shuffle ALU
SSE shuffle ALU
Memory order buffer
SSE shuffle ALU
128-bit FMUL FDIV
128-bit FMUL FDIV
Store & load
128-bit FMUL FDIV
512-Entry unified L2 TLB (4-way)
64-Entry data TLB (4-way associative)
32-KB dual-ported data cache (8-way associative)
256 KB unified l2 cache (eight-way)
8 MB all core shared and inclusive L3 cache (16-way associative)
Uncore arbiter (handles scheduling and clock/power state differences)

Core i7 Performance



Core i7 Branch Speculation Performance

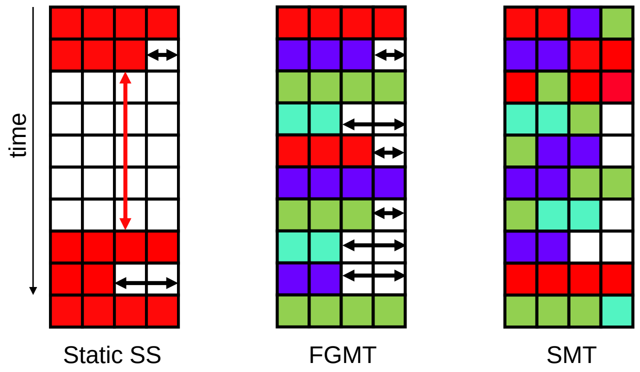


Review: Multithreaded Implementations

- MT trades (single-thread) latency for throughput
  - Sharing the datapath degrades the latency of individual threads, but improves the aggregate latency of both threads
  - And it improves utilization of the datapath hardware
- Main questions: **thread scheduling policy and pipeline partitioning**
  - When to switch from one thread to another?
  - How exactly do threads share the pipelined datapath itself?
- Choices depends on what kind of latencies you want to tolerate and how much single thread performance you are willing to sacrifice
  - Coarse-grain multithreading (**CGMT**)
  - Fine-grain multithreading (**FGMT**)
  - Simultaneous multithreading (**SMT**)

Vertical and Horizontal Under-Utilization

- FGMT reduces **vertical under-utilization**
  - Loss of all slots in an issue cycle
- Does not help with **horizontal under-utilization**
  - Loss of some slots in an issue cycle (in a static SS)



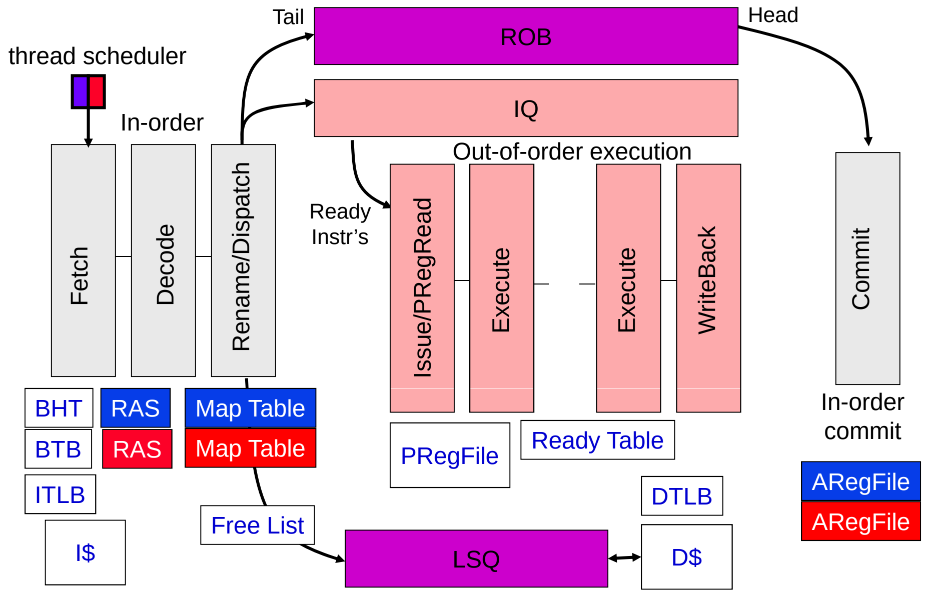
Simultaneous MultiThreading (SMT)

- What can issue instr’s from multiple threads in one cycle?
  - Same thing that issues instr’s from multiple parts of same thread …
  - …out-of-order execution !!
- **Simultaneous multithreading (SMT)**: OOO + FGMT
  - Aka (by Intel) **“hyper-threading”**
  - Once instr’s are renamed, issuer doesn’t care which thread they come from (well, for non-loads at least)
  - Some examples
    - IBM Power5: 4-way, 2 threads; IBM Power7: 4-way, 4 threads
    - Intel Pentium4: 3-way, 2 threads; Intel Core i7: 4-way, 2 threads
    - AMD Bulldozer: 4-way, 2 threads
    - Alpha 21464: 8-way issue, 4 threads (canceled)
    - Notice a pattern? #threads (T) * 2 = #issue width (N)

SMT Resource Partitioning

- Each thread must have its own persistent hard state structures
  - Per-thread PC (thread scheduler)
  - Map Table
  - ARegFile
- No-state (combinational) structures (e.g., ALU) can be dynamically shared
- As with FGMT, TLBs, caches, bpred tables (BHT,BTB) are already dynamically partitioned (persistent soft state) so can be shared
  - Some structures, e.g., TLBs, BTBs, will need thread ids
  - Some ordered “soft” state structures (e.g., RAS) will have to be replicated

SMT Out-of-order Pipeline



SMT Resource Partitioning, con't

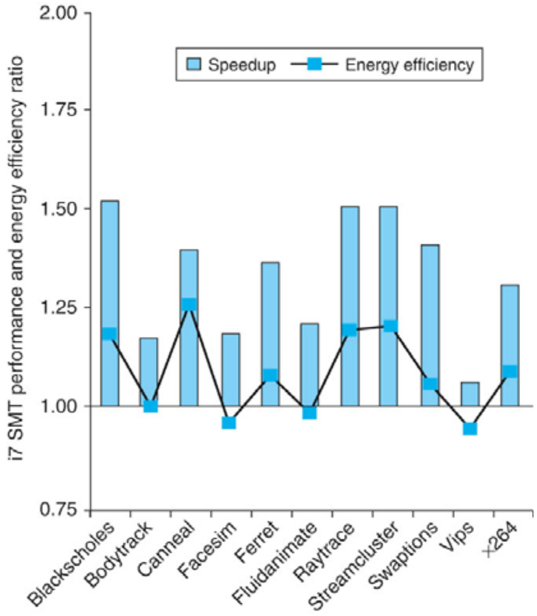
Transient state structures will need to be partitioned

- ❑ Execution pipeline latches shared as with FGMT
- ❑ Free List, PRegFile, Ready Table, and IQ entries can be partitioned (shared) at the fine grain (entry) level
  - Physically unordered and so fine-grain sharing is possible
  - Probably want a bigger PRegFile and IQ
    - # physical registers = (**#threads** * #arch-regs) + #in-flight instr's
    - # Map Table entries = (**#threads** * #arch-regs)
- ❑ How are physically ordered structures (ROB, LSQ) shared?
  - Fine-grain sharing (as with IQ) would entangle commit (and squash on branch misprediction, interrupts)
  - Allowing threads to commit independently is important, so …

Static vs Dynamic ROB & LSQ Partitioning

- ❑ **Static partitioning** (basically one per thread)
  - T equal-sized contiguous partitions in the ROB and LSQ
    - T is the number of threads
    - Essentially equivalent to having a ROB and LSQ for each thread
  - Could have sub-optimal utilization (fragmentation) as some ROBs could fill up while others are almost empty
  - But no starvation (as in dynamic partitioning)
- ❑ **Dynamic partitioning**
  - #partitions > #T, available partitions assigned on need basis
  - Better utilization
  - Possible starvation (one thread grabs most/all the partitions, so other threads are "starved")
  - Couple with a fetch policy that gives a preference to threads with fewest in-flight instr's
- ❑ Both need a larger ROB and LSQ

Multithreading Speed-Ups on the Core i7



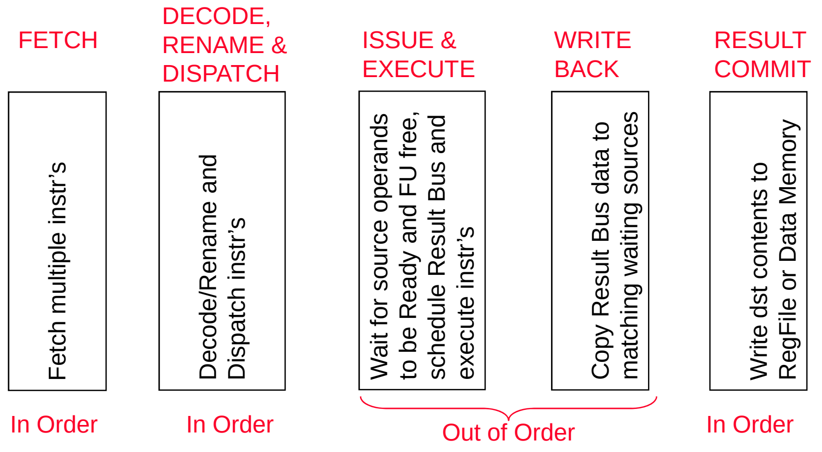
Multithreading vs Multicore

- ❑ If you wanted to run multiple threads would you build a
  - A multicore: multiple separate pipelines?
  - A multithreaded processor: a single larger pipeline?
- ❑ **Both will get you throughput on multiple threads**
  - A multicore core will be simpler, possibly faster clock
    - Multicore is mainly a TLP (thread-level parallelism) engine
  - SMT will get you better performance (IPC) on a single thread
    - SMT is basically an ILP engine that converts TLP to ILP
- ❑ **Do both**
  - Intel’s Sandy (Ivy) Bridge and Haswell, IBM’s Power7 & 8
  - 4 to 8 OOO 4-way cores each of which supports 2 to 4 threads (SMT)
  - Private L1 and L2 caches, shared L3 cache
  - 3+ GHz clock rate

SimpleScalar Structure

- `sim-outorder`: supports out-of-order execution (with in-order commit) with a Register Update Unit (RUU)
  - Uses a RUU for register renaming and to hold the results of pending instructions (our IQ). The RUU also retires (i.e., commits) completed instructions (so our ROB) in program order to the RegFile
  - Uses a LSQ for store instructions not ready to commit and load instructions waiting for access to the D$
  - Loads are satisfied by either the memory or by an earlier store value residing in the LSQ if their addresses match
    - Loads are issued to the memory system only when addresses of *all* previous (older) loads and stores are known

SimpleScalar Pipeline Stage Functions



FETCH

Fetch multiple instr's

In Order

DECODE, RENAME & DISPATCH

Decode/Rename and Dispatch instr's

In Order

ISSUE & EXECUTE

Wait for source operands to be Ready and FU free, schedule Result Bus and execute instr's

WRITE BACK

Copy Result Bus data to matching waiting sources

Out of Order

RESULT COMMIT

Write dst contents to RegFile or Data Memory

In Order

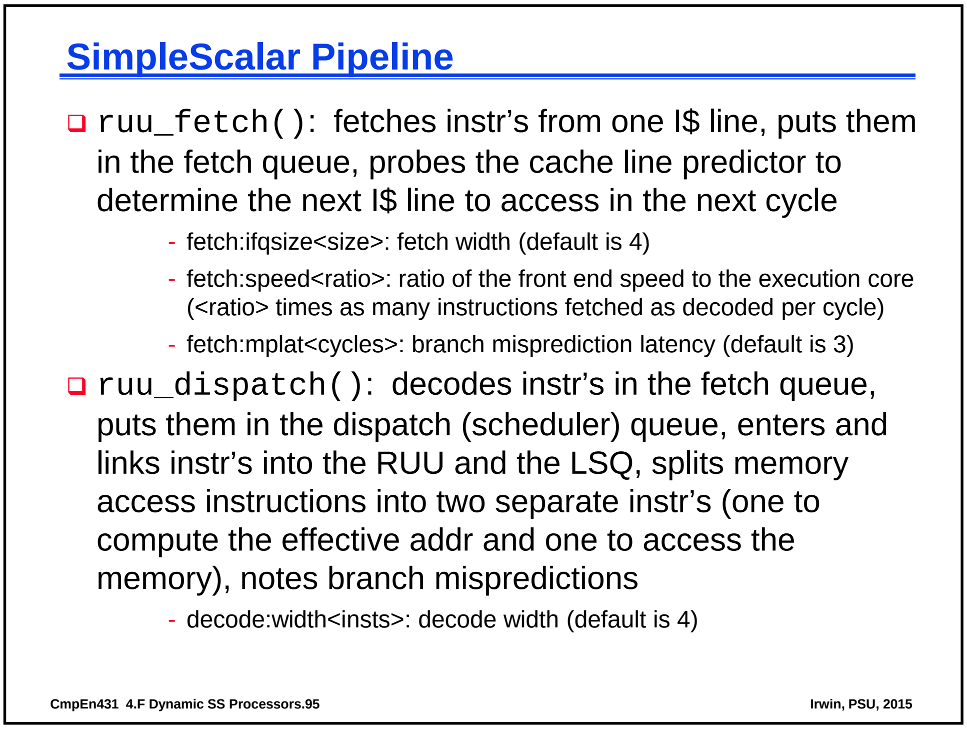
`ruu_fetch()`

`ruu_dispatch()`

`ruu_issue()`
`lsq_refresh()`

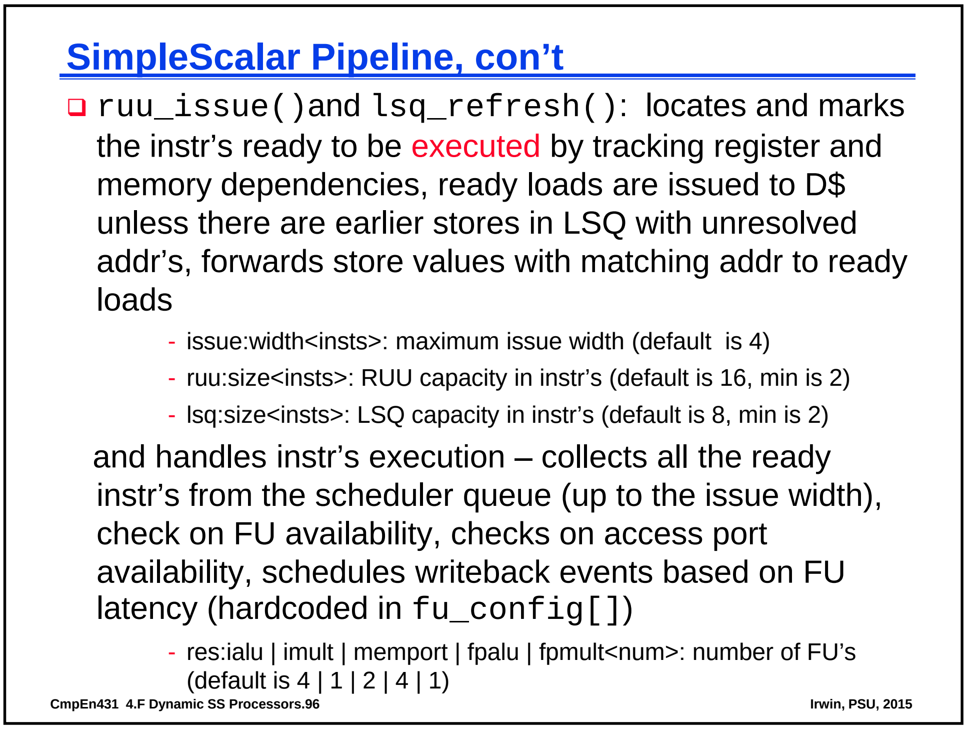
`ruu_writeback()`

`ruu_commit()`



SimpleScalar Pipeline

- `ruu_fetch()`: fetches instr's from one I$ line, puts them in the fetch queue, probes the cache line predictor to determine the next I$ line to access in the next cycle
  - fetch:ifqsize<size>: fetch width (default is 4)
  - fetch:speed<ratio>: ratio of the front end speed to the execution core (<ratio> times as many instructions fetched as decoded per cycle)
  - fetch:mplat<cycles>: branch misprediction latency (default is 3)
- `ruu_dispatch()`: decodes instr's in the fetch queue, puts them in the dispatch (scheduler) queue, enters and links instr's into the RUU and the LSQ, splits memory access instructions into two separate instr's (one to compute the effective addr and one to access the memory), notes branch mispredictions
  - decode:width<insts>: decode width (default is 4)



SimpleScalar Pipeline, con't

- `ruu_issue()`and `lsq_refresh()`: locates and marks the instr's ready to be executed by tracking register and memory dependencies, ready loads are issued to D$ unless there are earlier stores in LSQ with unresolved addr's, forwards store values with matching addr to ready loads
  - issue:width<insts>: maximum issue width (default is 4)
  - ruu:size<insts>: RUU capacity in instr's (default is 16, min is 2)
  - lsq:size<insts>: LSQ capacity in instr's (default is 8, min is 2)

and handles instr's execution – collects all the ready instr's from the scheduler queue (up to the issue width), check on FU availability, checks on access port availability, schedules writeback events based on FU latency (hardcoded in `fu_config[]`)

  - res:ialu | imult | memport | fpalu | fpmult<num>: number of FU's (default is 4 | 1 | 2 | 4 | 1)

SimpleScalar Pipeline, con't

- `ruu_writeback()`: determines completed instr's, does data forwarding to dependent waiting instr's, detects branch misprediction and on misprediction rolls the machine state back to the checkpoint and discards erroneously issued instructions

- `ruu_commit()`: in-order commits results for instr's (values copied from RUU to RegFile or LSQ to D$), RUU/LSQ entries for committed instr's freed; keeps retiring instructions at the head of RUU that are ready to commit until the head instr is one that is not ready

Reminders

- This week
  - Dynamic superscalar processors – out-of-order and SMT
    - PH 4.10 and PH 7.5
- Next week
  - Multicores and multicore (multiprocessor) cache coherence
    - PH 2.11, 5.8-5.9, 7.7
- Reminders
  - HW5 due Nov 12th, HW6 (Project) will come out Nov 5th and will be due Dec 14th
  - Quiz5 will close on Nov 2nd, Quiz 6 will close on Nov 16th
  - Second (and **last**) evening exam scheduled
    - Tuesday, November 17, 20:15 to 22:15, Location 22 Deike
    - If you have a conflict I **MUST** KNOW before October 31st
  - Note that (unlike what elion says) there is no final exam during the final exam period in this course