
CMPEN 431

Computer Architecture

Fall 2017

The Dynamic SuperScalar (OOO) Processor

Mahmut Taylan Kandemir (www.cse.psu.edu/~kandemir)

[Adapted from *Computer Organization and Design, 5th Edition*,

Patterson & Hennessy, © 2014, MK

With additional thanks/credits to Amir Roth, Milo Martin, Onur Mutlu

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 = .
. = a

lw \$t0, 0(\$s1)
addu \$t0, \$t0, \$s2

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

- WAR, anti-dependence (**renaming** allows reordering)

. = a
a = .

lw \$t0, 0(\$s1)
addu \$s1, \$s2, \$s3

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

- WAW, output dependence (**renaming** allows reordering)

a = .
a = .

lw \$t0, 0(\$s1)
addu \$t0, \$s2, \$s3

sw \$t0, 0(\$s1)
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 \Rightarrow 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
 - Named after CDC 6600 scoreboard
 - <http://www.computerhistory.org/revolution/supercomputers/10/33>
 - 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>

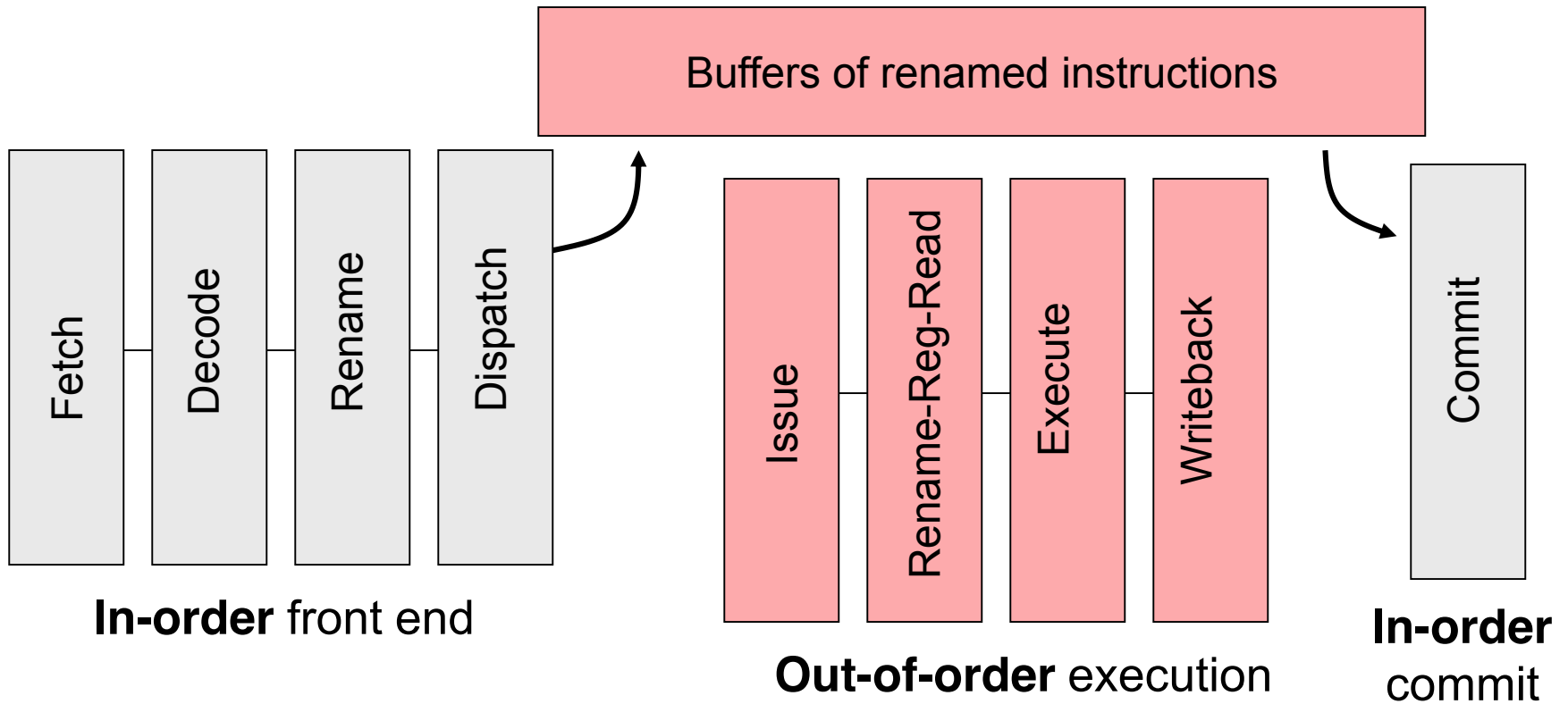
More Recent Dynamic OOO Datapaths

- ❑ 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)
3. **Issue** (Out Of Order -- **OOO**): When an instr has all of its source data and the FU (Functional Unit) it needs is free, it is issued for **execution**
4. Writeback (OOO): When the dst value has been computed it is written back to the PhysicalRegFile and other shared structures – the instr **completes**
5. Commit (in program order): Changes the state of the machine

Out-of-Order Pipeline



Code Example

RAW

WAR

WAW

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) : ...

The diagram illustrates data hazards between two code blocks. Blue arrows represent Read-After-Write (RAW) hazards, showing that the second block's instructions depend on the first block's instructions. Green arrows represent Write-After-Read (WAR) hazards, indicating that the first block's instructions depend on the second block's instructions. Red arrows represent Write-After-Write (WAW) hazards, showing that the first block's instructions depend on the second block's instructions. The first block's instructions are: lw \$t0, 0(\$s1); addu \$t0, \$t0, \$s2; sw \$t0, 0(\$s1); sub \$t0, \$s1, \$s2; addi \$s1, \$s1, -4; bne \$s1, \$0, lp. The second block's instructions are: lw \$t0, 0(\$s1); addu \$t0, \$t0, \$s2; sw \$t0, 0(\$s1); sub \$t0, \$s1, \$s2; addi \$s1, \$s1, -4; bne \$s1, \$0, lp. The third block's instruction is: 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 (create artificial dependencies)
 - 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) : ...
```

The diagram illustrates data flow dependencies between instructions in three loops. Red arrows represent true dependencies, green arrows represent false dependencies, and blue arrows represent control flow. In the first loop, the `sub` instruction is independent of the `lw` and `addu` instructions that precede it. The `addi` instruction has a true dependency on the `sub` instruction. The `bne` instruction has a true dependency on the `addi` instruction. In the second loop, the `sub` instruction is independent of the `lw` and `addu` instructions that precede it. The `addi` instruction has a true dependency on the `sub` instruction. The `bne` instruction has a true dependency on the `addi` instruction. The `sub` instruction in the second loop has a true dependency on the `sub` instruction in the first loop. The `addi` instruction in the second loop has a true dependency on the `addi` instruction in the first loop. The `bne` instruction in the second loop has a true dependency on the `bne` instruction in the first loop. The `sub` instruction in the third loop has a true dependency on the `sub` instruction in the second loop. The `addi` instruction in the third loop has a true dependency on the `addi` instruction in the second loop. The `bne` instruction in the third loop has a true dependency on the `bne` instruction in the second loop.

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: 1w Renaming

Over-written Reg

lw \$t0, 0(\$s1) → lw p4, 0(p1) [p3]
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

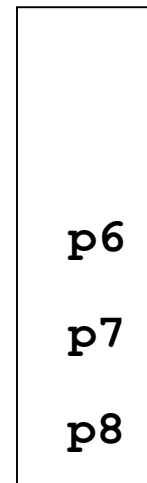
Renaming Example: addu Renaming

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)
sub   $t0, $s1, $s2
addi  $s1, $s1, -4
bne   $s1, $0, lp
```

\$s1	p1
\$s2	p2
\$t0	p5

Map Table



Free List

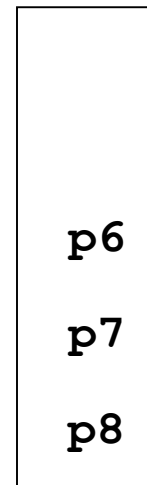
Renaming Example: sw Renaming

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				
addi	\$s1, \$s1, -4				
bne	\$s1, \$0, lp				

\$s1	p1
\$s2	p2
\$t0	p5

Map Table



Free List

Renaming Example: sub 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				
bne	\$s1, \$0, lp				

\$s1	p1
\$s2	p2
\$t0	p6

Map Table



Free List

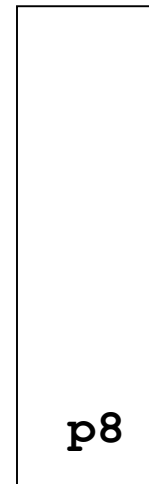
Renaming Example: addi Renaming

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			

\$s1	p7
\$s2	p2
\$t0	p6

Map Table



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



Free List

Code Example After Renaming

RAW

WAR - none

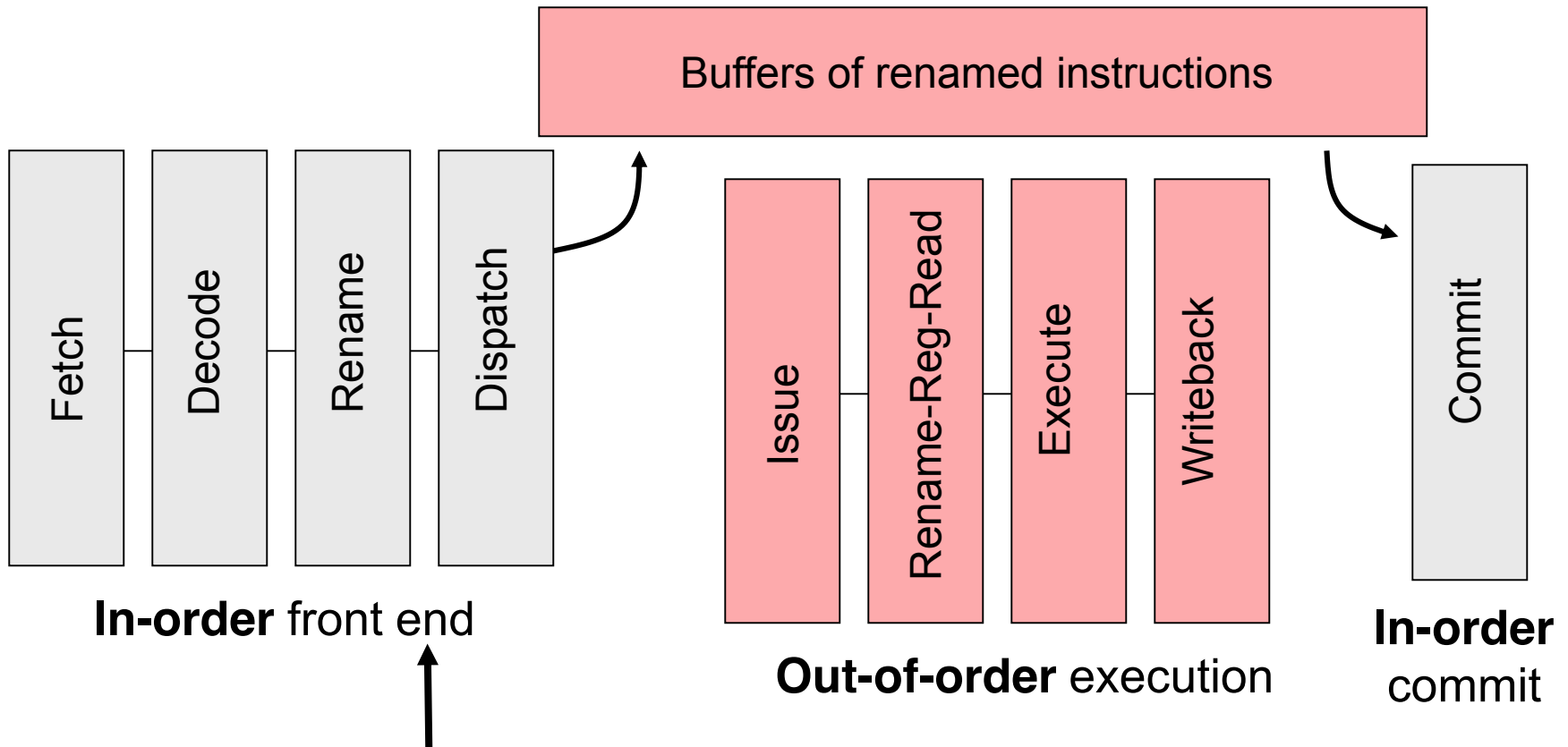
WAW - none

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	p1, 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) : ...		

- ❑ As promised, renaming **eliminated** false data dependencies (WAW, WAR) and left true data dependencies (RAW) **intact**

Out-of-Order Pipeline Progress

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



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

Why is Dynamic Scheduling?

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

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

```
LD    R3 ← R1 (0)
ADD  R3 ← R3, R1
ADD  R1 ← R6, R7
IMUL R5 ← R6, R8
ADD  R7 ← R3, R5
```

- ❑ Answer: First ADD stalls 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

Out-of-order Execution (Dynamic Scheduling)

- ❑ Idea: Move the dependent instructions out of the way of independent ones
 - Rest areas for dependent instructions: Reservation Stations
- ❑ Monitor the source “values” of each instruction in the resting area
- ❑ When all source “values” of an instruction are available, “fire” (i.e. issue) the instruction
 - Instructions dispatched in dataflow (not control-flow) order
- ❑ Benefit:
 - Latency tolerance: Allows independent instructions to execute and complete in the presence of a long latency operation

Advantages of Dynamic Scheduling

- ❑ Allows code that was compiled with one pipeline in mind to run efficiently on a different pipeline
 - Eliminates need to have multiple binaries and recompile
- ❑ Enables handling some cases when dependences are unknown at compile time
 - E.g., may involve memory reference or data-dependent branch
- ❑ Allows the processor to tolerate unpredictable delays
 - E.g., cache misses

Enabling OoO Execution

1. Need to link the consumer of a value to the producer
 - **Register renaming:** Associate a “tag” with each data value
2. Need to buffer instructions until they are ready
 - Insert instruction into **reservation stations** after renaming
3. Instructions need to keep track of readiness of source values
 - **Broadcast the “tag”** when the value is produced
 - Instructions **compare their “source tags”** to the broadcast tag → if match, source value becomes ready
4. When all source values of an instruction are ready, issue the instruction to functional unit (FU)
 - What if more instructions become ready than available FUs?

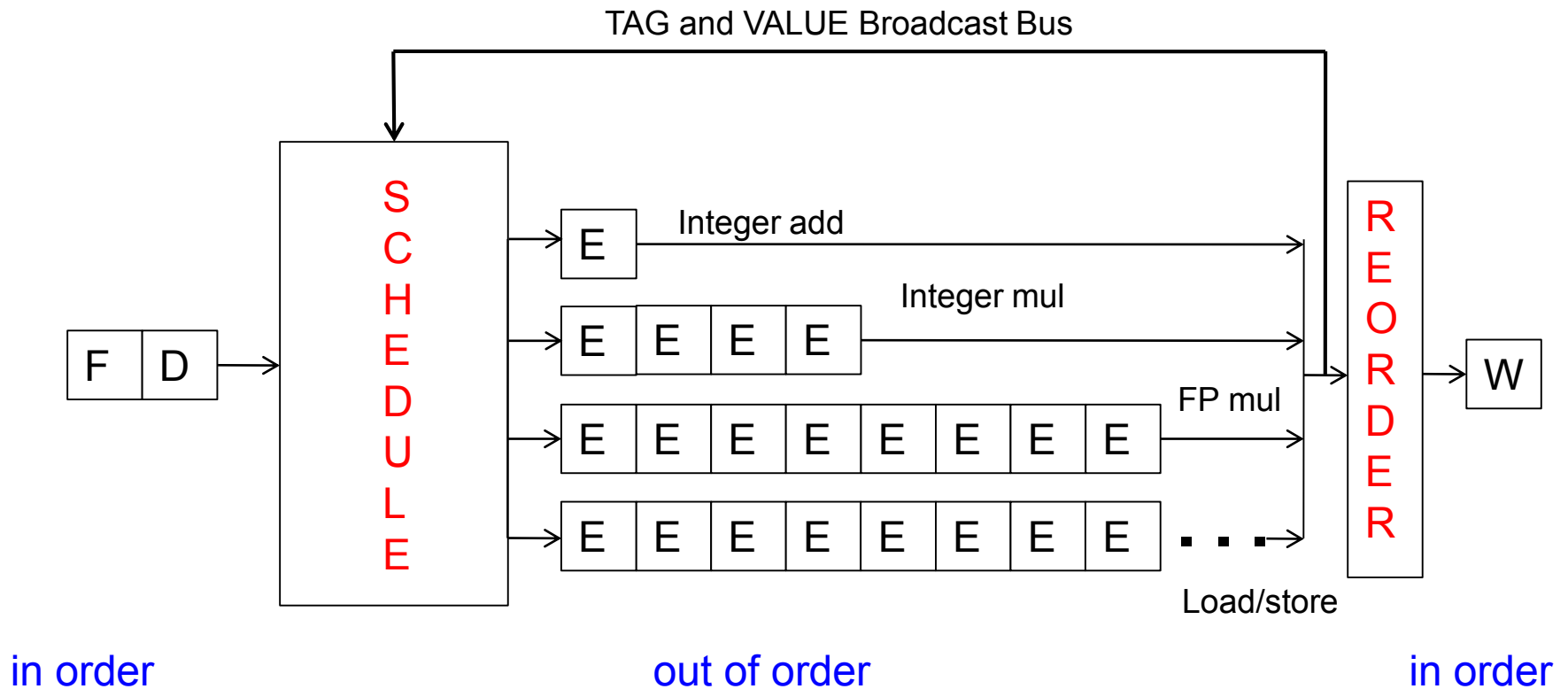
Tomasulo's Algorithm

- ❑ OoO with register renaming invented by Robert Tomasulo
 - Used in IBM 360/91 Floating Point Units
 - **Read:** 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.

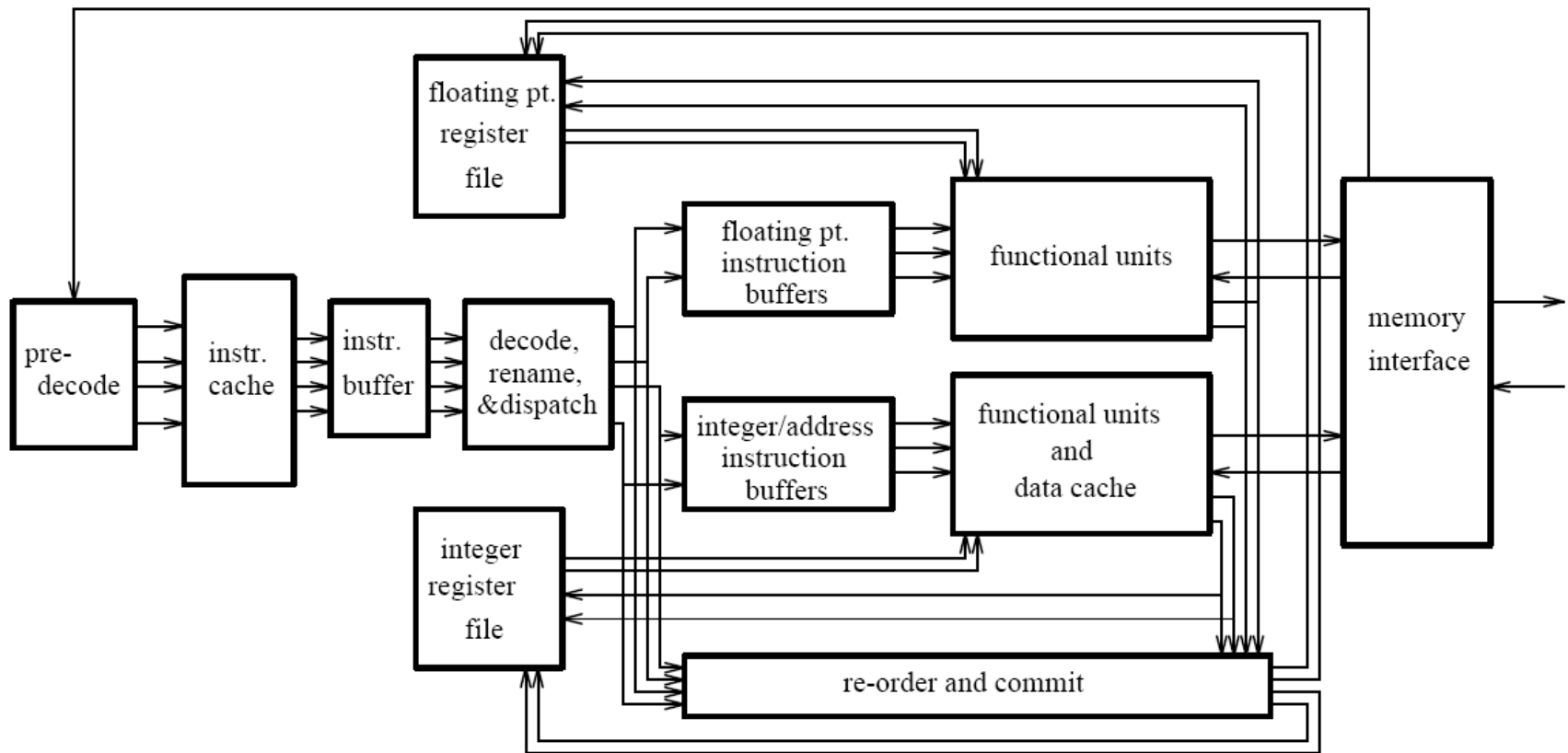
- ❑ Variants used in most high-performance processors
 - Most notably Pentium Pro, Pentium M, Intel Core(2), AMD K series,
 - Alpha 21264, MIPS R10000, IBM POWER5, Oracle UltraSPARC T4

Two Humps in the Pipeline



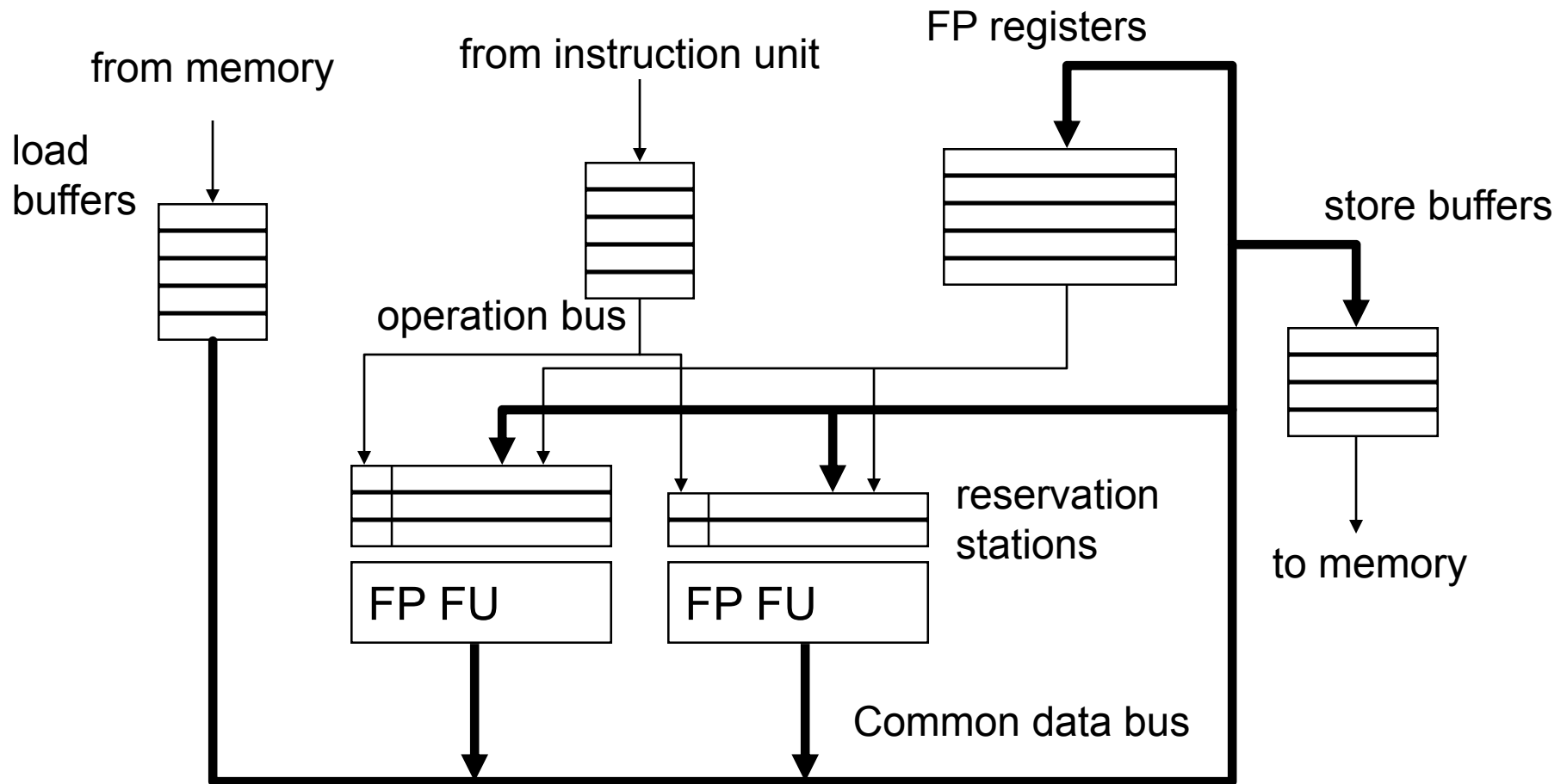
- ❑ **Hump 1:** Reservation stations (scheduling window)
- ❑ **Hump 2:** Reordering (reorder buffer, aka instruction window or active window)

General Organization of an OOO Processor



- Smith and Sohi, “**The Microarchitecture of Superscalar Processors**,” Proc. IEEE, Dec. 1995.

Tomasulo's Machine: IBM 360/91



Register Renaming in Tomasulo's Algorithm

- ❑ 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 because not enough register ID's (i.e., names) in the ISA
- ❑ The register ID is **renamed** to the reservation station entry that will hold the register's value
 - Register ID → RS entry ID
 - Architectural register ID → Physical register ID
 - After renaming, RS entry ID used to refer to the register
- ❑ This eliminates anti- and output- dependencies
 - Approximates the performance effect of a large number of registers even though ISA has a small number

Tomasulo's Algorithm

- ❑ If reservation station available before renaming
 - Instruction + renamed operands (source value/tag) inserted into the reservation station (dispatch)
 - Only rename if reservation station is available
- ❑ Else stall
- ❑ While in reservation station, each instruction:
 - Watches common data bus (CDB) for tag of its sources
 - When tag seen, grab value for the source and keep it in the reservation station
 - When both operands available, instruction ready to be issued
- ❑ Issue instruction to the Functional Unit when instruction is ready
- ❑ After instruction finishes in the Functional Unit
 - Arbitrate for CDB
 - Put tagged value onto CDB (tag broadcast)
 - Register file is connected to the CDB
 - Register contains a tag indicating the latest writer to the register
 - If the tag in the register file matches the broadcast tag, write broadcast value into register (and set valid bit)
 - Reclaim rename tag
 - no valid copy of tag in system!

Hardware-Based Speculation

- ❑ Execute instructions along predicted execution paths but only commit the results if prediction was correct
- ❑ Instruction commit: allowing an instruction to update the register file when instruction is no longer speculative
- ❑ Need an additional piece of hardware to prevent any irrevocable action until an instruction commits
 - i.e., updating state or taking an execution

Reorder Buffer

- ❑ Reorder buffer – holds the result of instruction between completion and commit

- ❑ Four fields:
 - Instruction type: branch/store/register
 - Destination field: register number
 - Value field: output value
 - Ready field: completed execution?

- ❑ Modify reservation stations:
 - Operand source is now reorder buffer instead of functional unit

Reorder Buffer

- ❑ Register values and memory values are not written until an instruction commits
- ❑ On misprediction:
 - Speculated entries in ROB are cleared
- ❑ Exceptions:
 - Not recognized until it is ready to commit

Dynamically scheduling memory ops

- ❑ Compilers must schedule memory ops conservatively
- ❑ Options for hardware:
 1. Don't execute any load until all prior stores execute (**conservative**)
 2. Execute loads as soon as possible, detect violations (**aggressive**)
 - When a store executes, it checks if any later loads executed too early (to the same address). If so, flush pipeline and restart
 3. Learn violations over time, selectively reorder (**predictive**)

Conservative

```
ld r2,4(sp)
ld r3,8(sp)
add r3,r2→r1
st r1,0(sp)
ld r5,0(r8) //stall
ld r6,4(r8)
sub r5,r6→r4
st r4,8(r8)
```

Aggressive (Wrong Outcome?)

```
ld r2,4(sp)
ld r3,8(sp)
ld r5,0(r8) //does r8==sp?
add r3,r2→r1
ld r6,4(r8) //does r8+4==sp?
st r1,0(sp)
sub r5,r6→r4
st r4,8(r8)
```

Memory forwarding

- ❑ Stores write cache at Commit (until then hold store value and address in a Store Queue)
 - Commit is in-order, so allows stores to (also) be “undone” on branch mis-predictions, etc.
- ❑ Loads read cache
 - Early execution of loads is critical
- ❑ Forwarding
 - Allow store to (following) load communication before store Commits (e.g., in the previous example forward p4 to p8)
 - Conceptually like register bypassing, but different implementation
 - Why? Addresses unknown until Execute

Store Queue with load forwarding

❑ Store Queue (SQ)

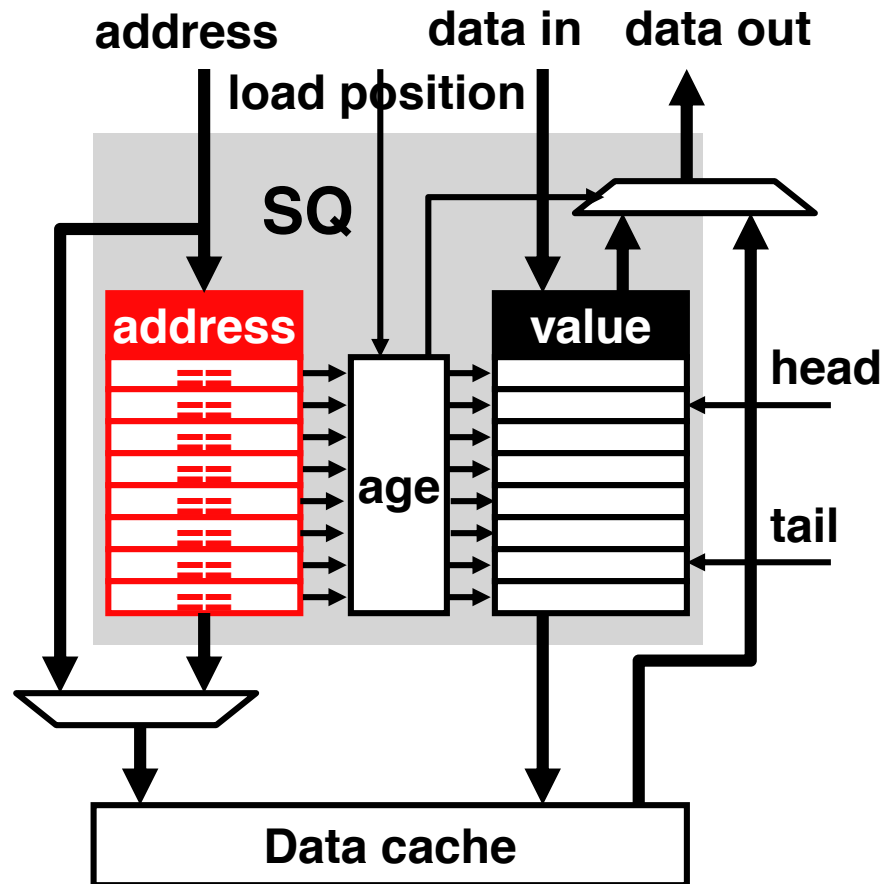
- Holds all in-flight stores
- **CAM**: searchable by address
- Age logic: determine *youngest* matching store that is *older* than the load

❑ Store execution

- Write into SQ
 - address + data value
- Store from SQ into D\$ at Commit

❑ Load execution

- Assoc search SQ addresses
 - Match? Forward youngest matching store value to load request
- Else read value from D\$



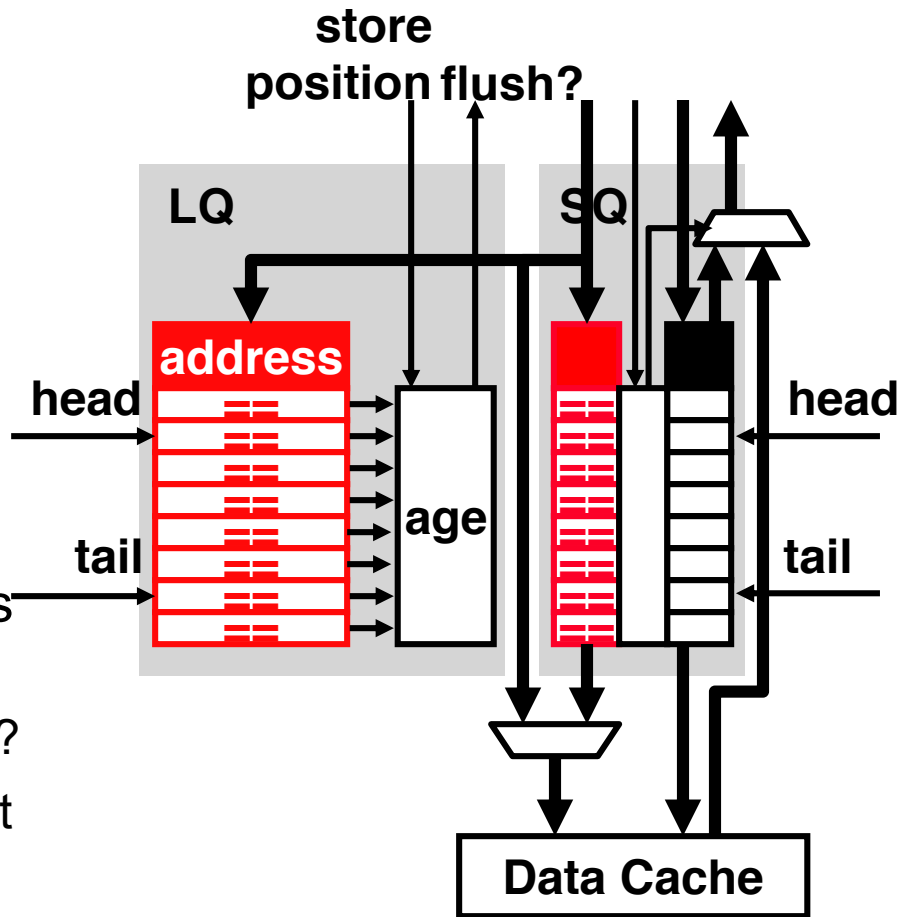
Load speculation

❑ Speculation requires two things.....

1. Detection of mis-speculations (guessed that memory addresses didn't match and it turns out that they did)
 - How can we do this?
2. Recovery from mis-speculations
 - Squash instr's after offending load (they may be using the load data)
 - Saw how to squash instr's after mispredicated branches: same method

Store Queue (SQ) + Load Queue (LQ)

- ❑ LQ detects load ordering violations
- ❑ Load execution
 - Write address into LQ
 - Also note store forwarded from if any
- ❑ Store execution
 - Assoc search LQ addresses
 - Found a younger (later in code) load with same addr?
 - If didn't forward data to that load from the youngest (older) store?
 - Then mis-speculated the load so initiate recovery



LSQ = Store Queue + Load Queue

- ❑ Store Queue: handles forwarding
 - Written into by stores (@ store execute)
 - Searched by loads (@ load execute) for store forwarding
 - Write to data cache (@ store Commit)
- ❑ Load Queue: detects ordering violations
 - Written into by loads (@ load execute)
 - Searched by stores (@ store execute) to detect load ordering violation
- ❑ Both together
 - Allow aggressive load scheduling
 - Stores don't constrain load execution
 - Help us improve performance significantly

Summary of OOO Execution Concepts

- ❑ Renaming eliminates false dependencies
- ❑ Tag broadcast enables value communication between instructions → dataflow
- ❑ An out-of-order engine dynamically builds the dataflow graph of a piece of the program
 - Which piece?
 - Limited to the instruction window
 - Can we do it for the whole program? Why would we like to?
 - How can we have a large instruction window efficiently?

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
UltraSparc III	2003	1950MHz	14	4	No	1	90W
UltraSparc T1	2005	1200MHz	6	1	No	8	70W

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

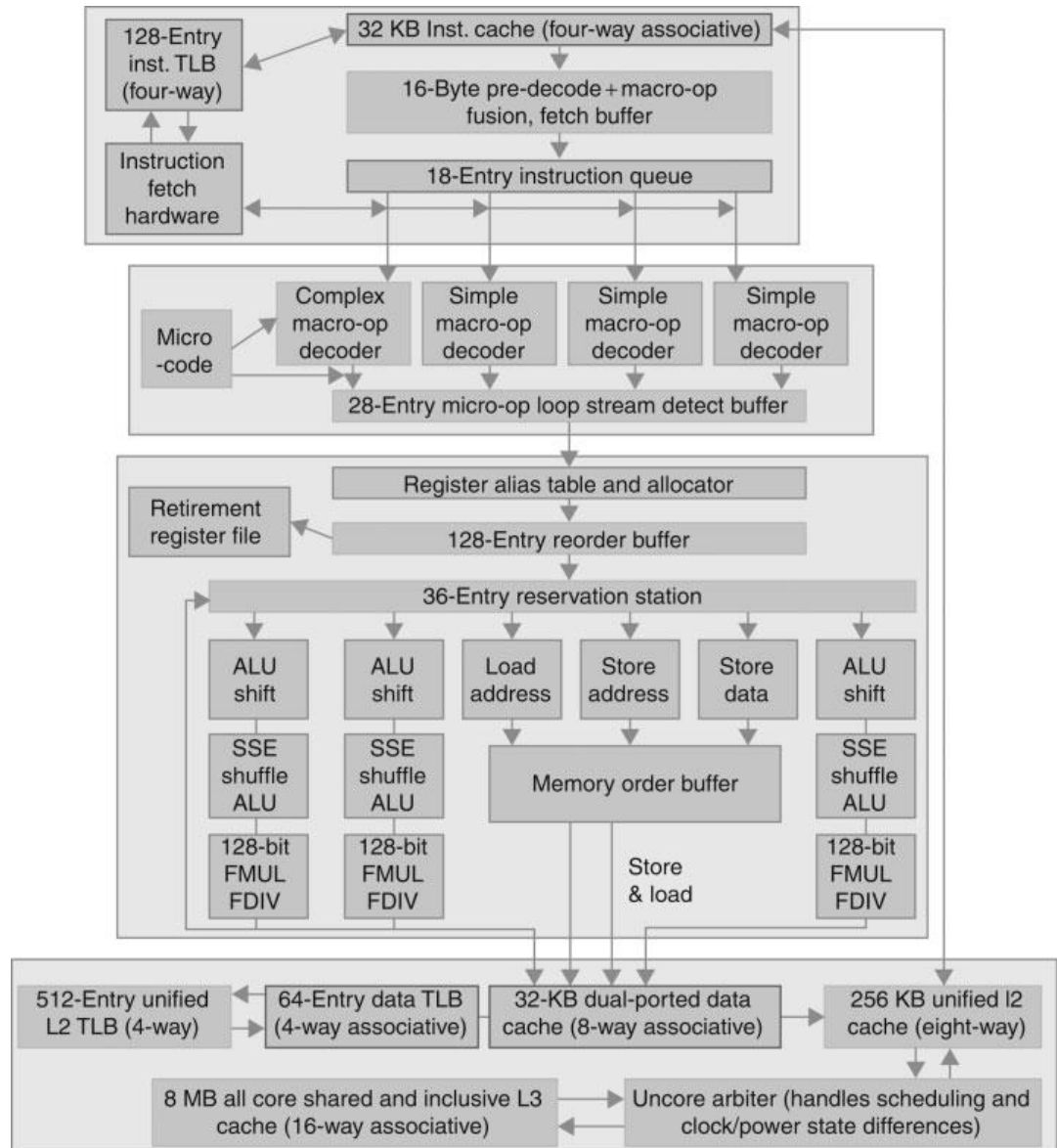
- ❑ Haswell is the fourth Tock since Intel instituted its Tick-Tock model
- ❑ After Haswell: Broadwell (14nm), Sky Lake (14nm), ...

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

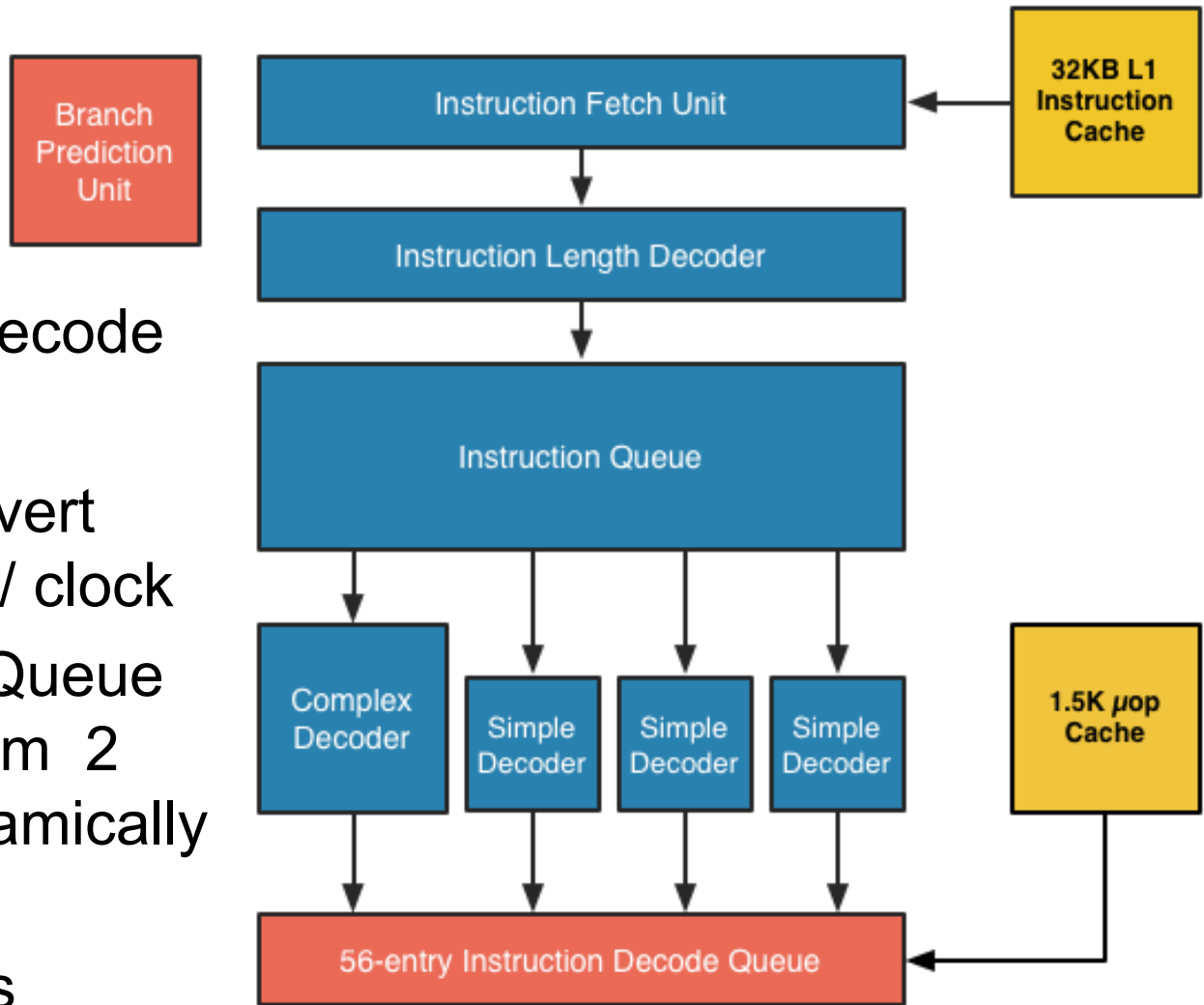
	Nehalem	Sandy Bridge	Haswell
Instr Decode Queue	28 per thread / 2 threads	28 per thread / 2 threads	56 total for 2 threads
ROB	128 uops	168 uops	192 uops
Res Station (IQ)	36 uops	56 uops	60 uops
Integer Rename RF		160 registers	168 registers
FP Rename RF		144 registers	168 registers
Load Buffers	48 entries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

Core i7 (Nehalem) Pipeline

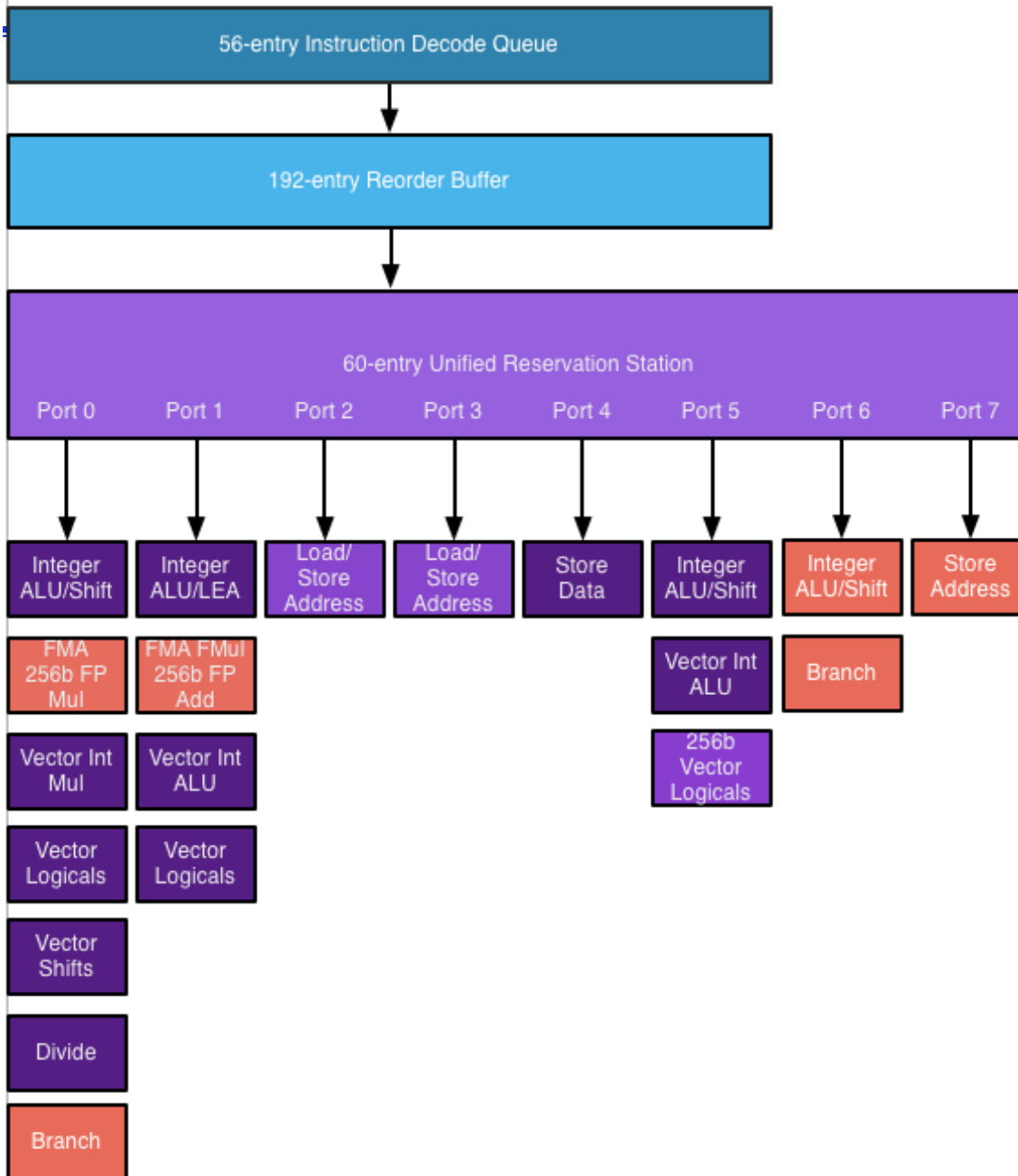


Intel Haswell Front End

- ❑ 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)

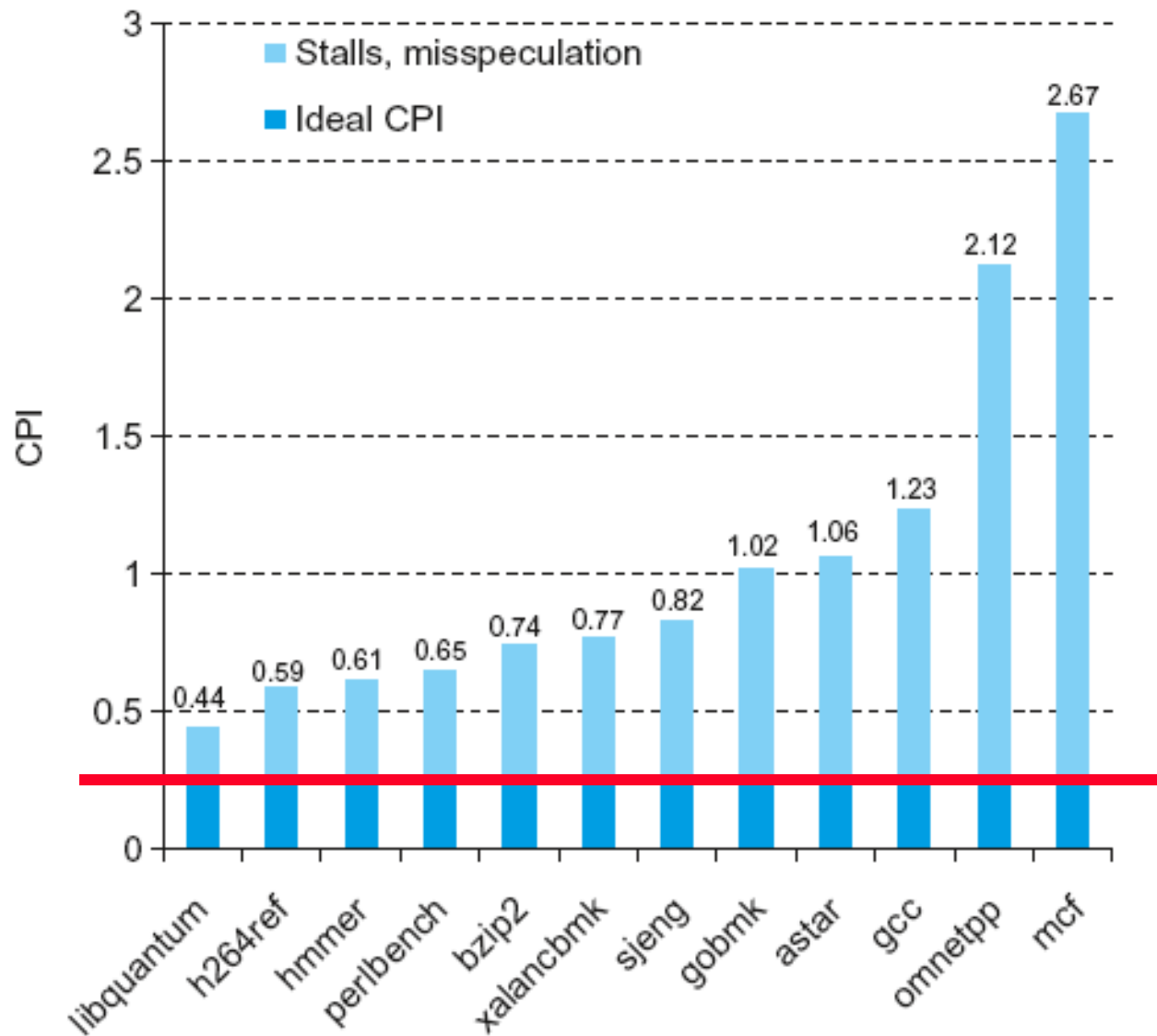


Intel Haswell Execution Engine

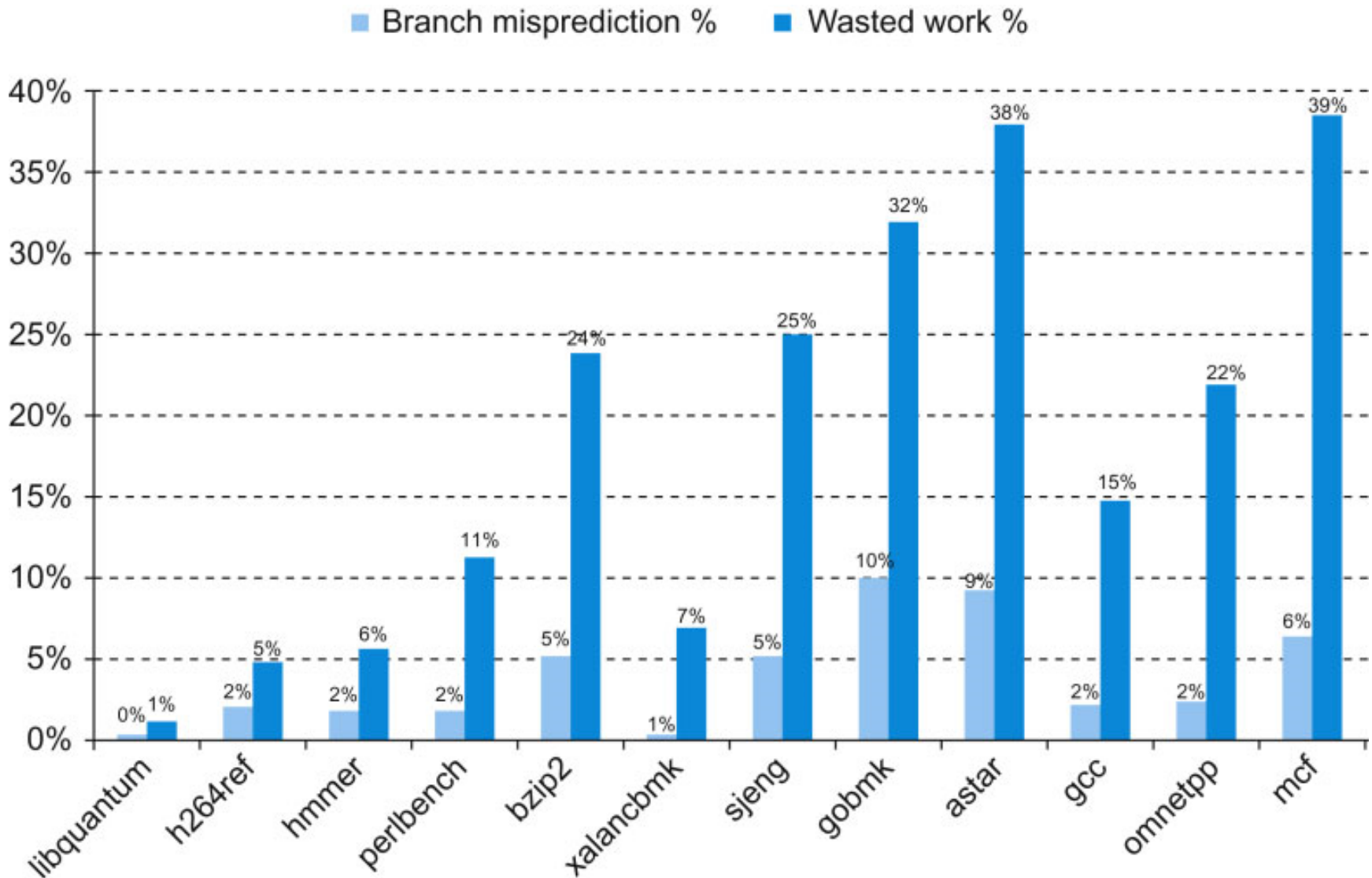


- ❑ IQ work done by the Reservation Station
- ❑ 8 execution ports (only 6 on Sandy Bridge)

Core i7 Performance




i7 Branch Speculation Performance



Cortex A8 versus Intel i7

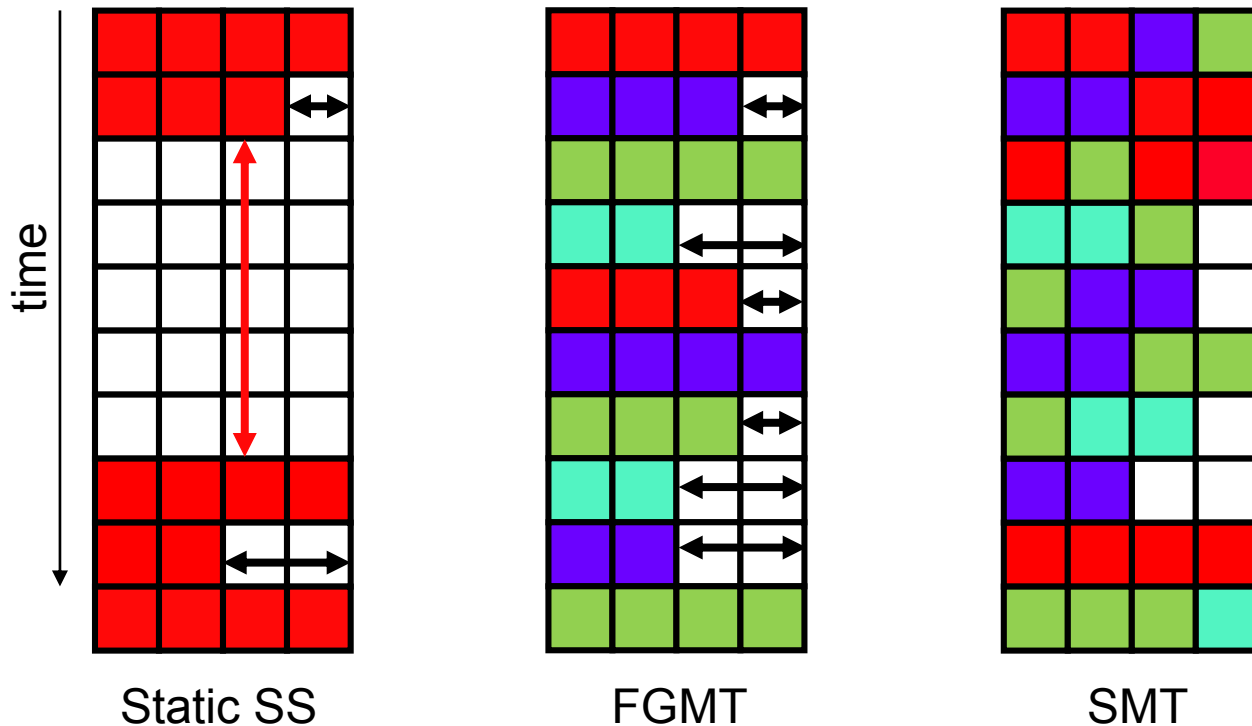
Processor	ARM A8	Intel Core i7 920
Market	Personal Mobile Device	Server, cloud
Thermal design power	2 Watts	130 Watts
Clock rate	1 GHz	2.66 GHz
Cores/Chip	1	4
Floating point?	No	Yes
Multiple issue?	Yes	Yes
Peak instructions/clock cycle	2	4
Pipeline stages	14	14
Pipeline schedule	Static in-order	Dynamic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

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

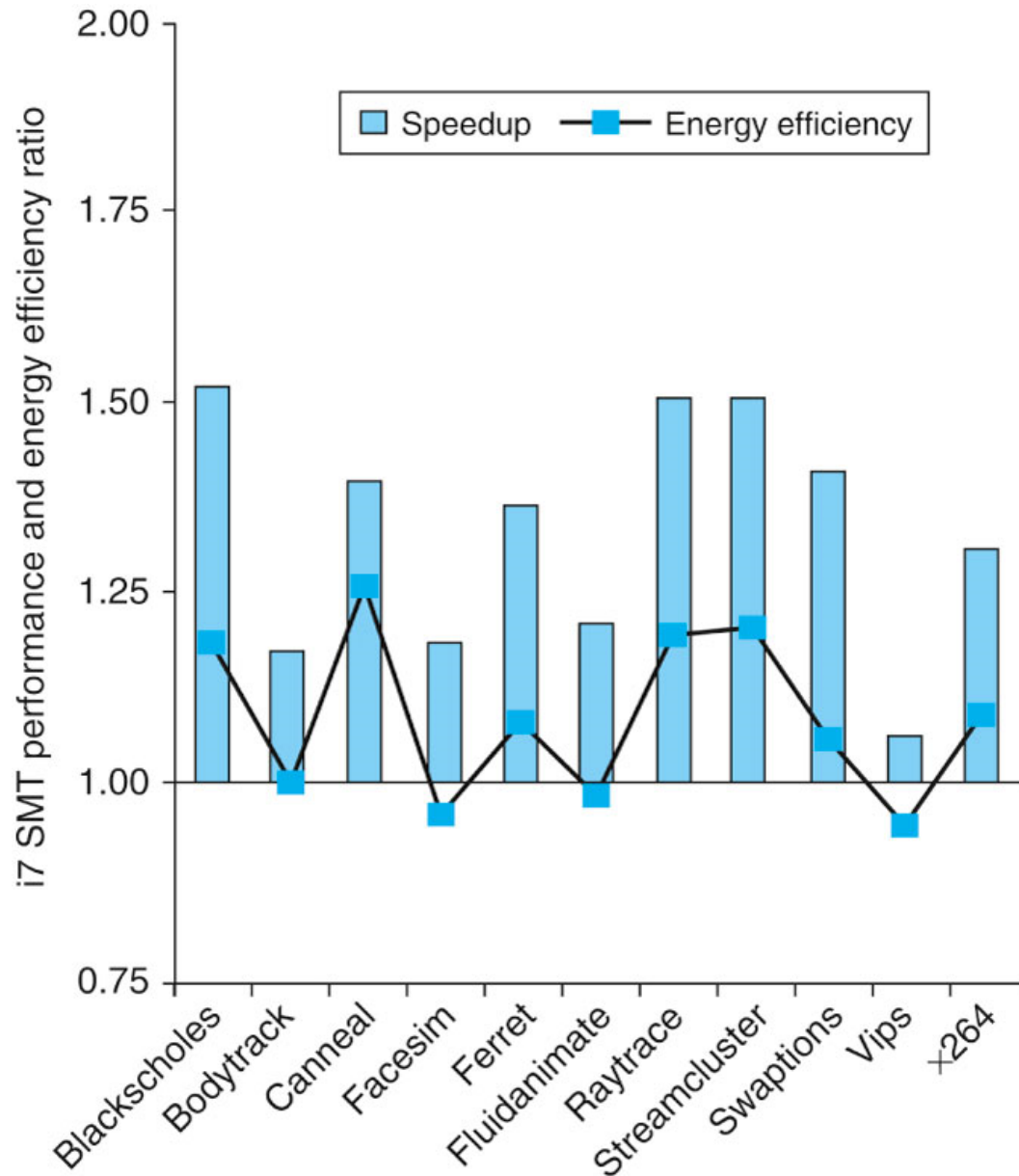
- ❑ FGMT reduces **vertical under-utilization**
 - Loss of all slots in an issue cycle
- ❑ Does not help with **horizontal under-utilization**
 - Loss of some slots in an issue cycle (in a static SS)



Simultaneous MultiThreading (SMT) omit

- ❑ 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? $\#threads (T) * 2 = \#issue\ width (N)$

Multithreading Speed-Ups on the 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

Check Yourself

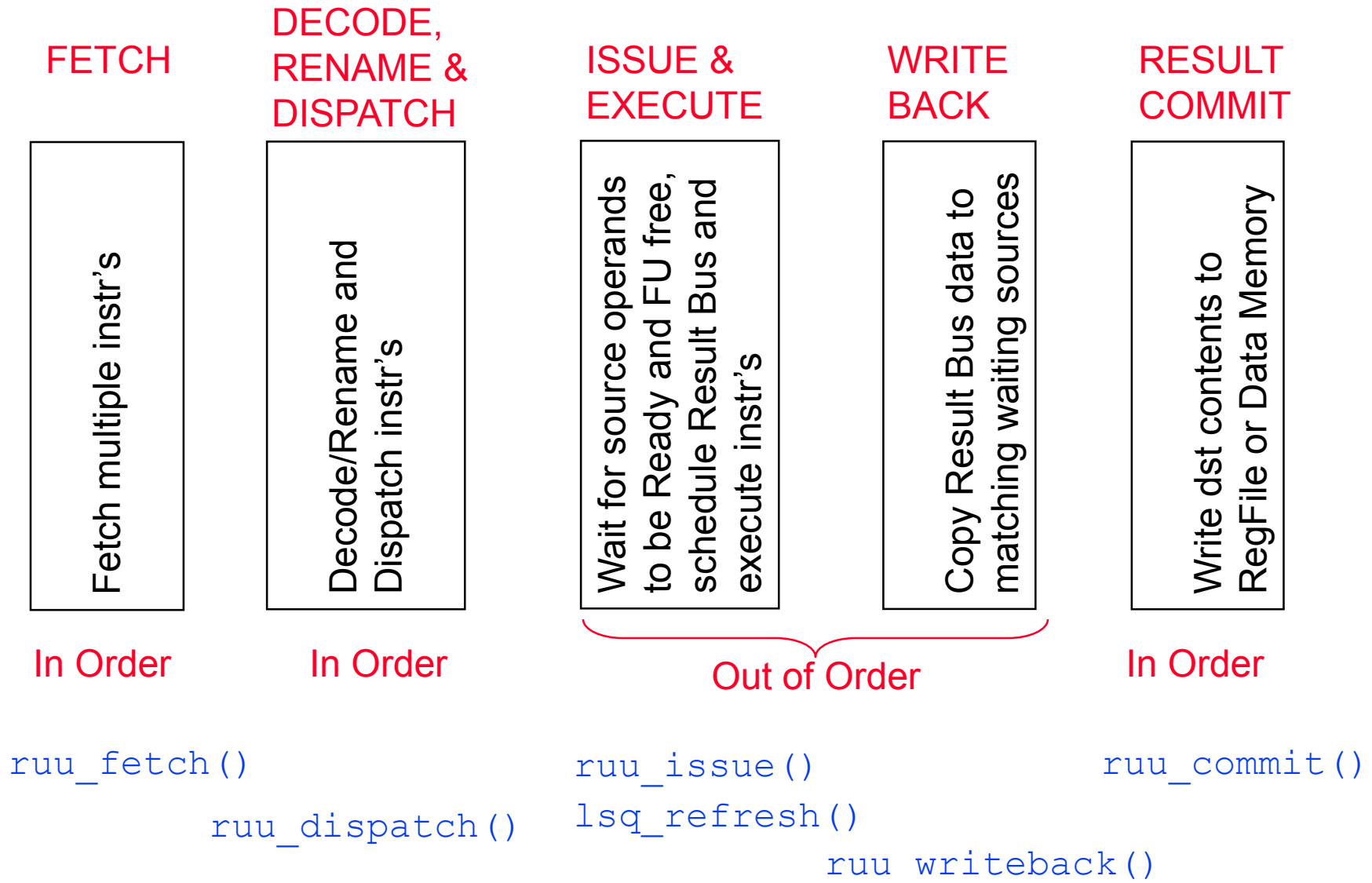
- ☐ Branch prediction
- ☐ Multiple issue
- ☐ VLIW
- ☐ Superscalar
- ☐ Static scheduling
- ☐ Dynamic scheduling
- ☐ In-order execution
- ☐ Out-of-order (OOO) execution
- ☐ Speculation
- ☐ Reorder buffer
- ☐ Register renaming

❑ BACKUP

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



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