SuperScalar Processors

CS/COE 1541 (Fall 2020)
Wonsun Ahn

University of Pittsburgh

In-order vs. Out-of-order superscalars

- **Superscalar**: a wide-issue processor that does dynamic scheduling
  - Extracts instruction level parallelism (ILP) within the processor

- **In-order** superscalar: **does not reorder** instructions
  - Only detects hazards between instructions to insert bubbles
  - Only extracts ILP that arises from given ordering of instructions
  - The processor simulated in Project 1
- **Out-of-order** superscalar: **does reorder** instructions
  - Reorders instructions to remove hazards and increase utilization
  - Typically results in higher performance compared to in-order
  - But dynamic reordering consumes lots of power

- Out-of-order sounds more exciting so let's talk about that

Name of the game is still ILP

- The processor internally constructs the data dependency graph
- The processor tries to take advantage of ILP as much as possible
  - By executing the red nodes in parallel with the blue nodes

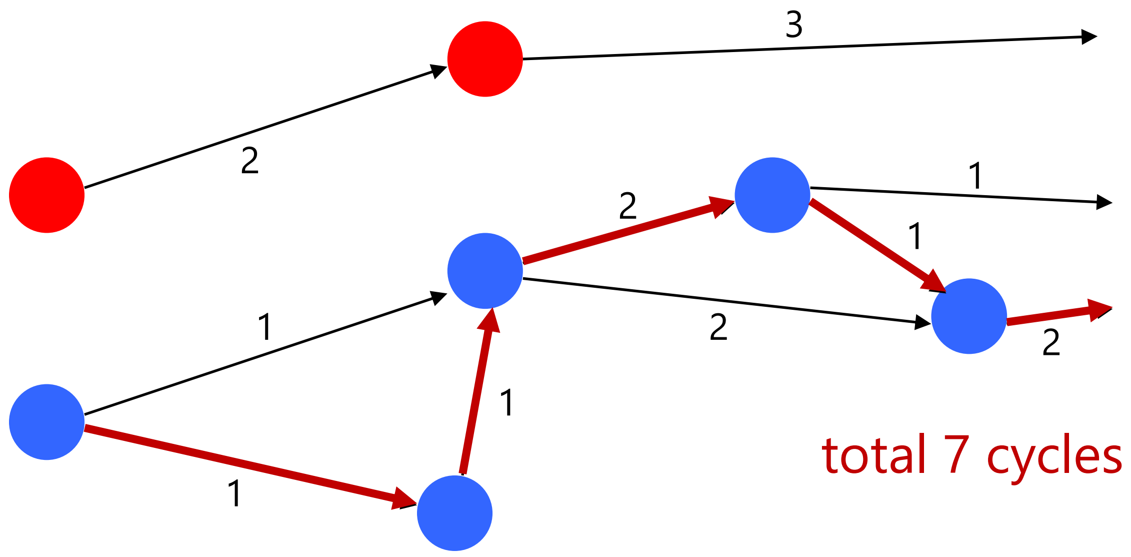
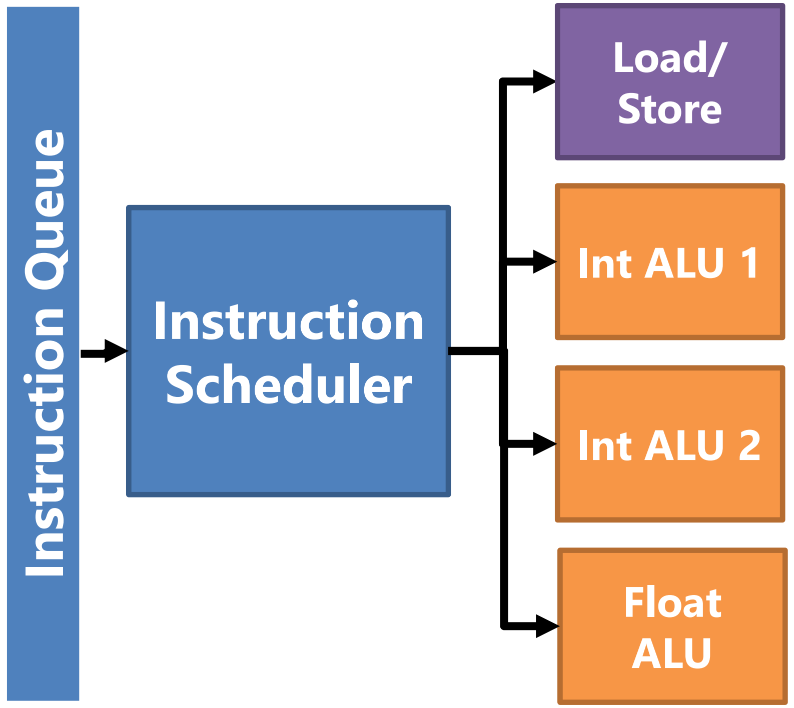


illustration courtesy of Dr. Melhem

University of Pittsburgh

Instruction Queue

- In order to expose ILP, superscalars need a big **instruction window**
  - Just like the compiler did for VLIWs
  - HW structure for storing instructions is called ***instruction queue***

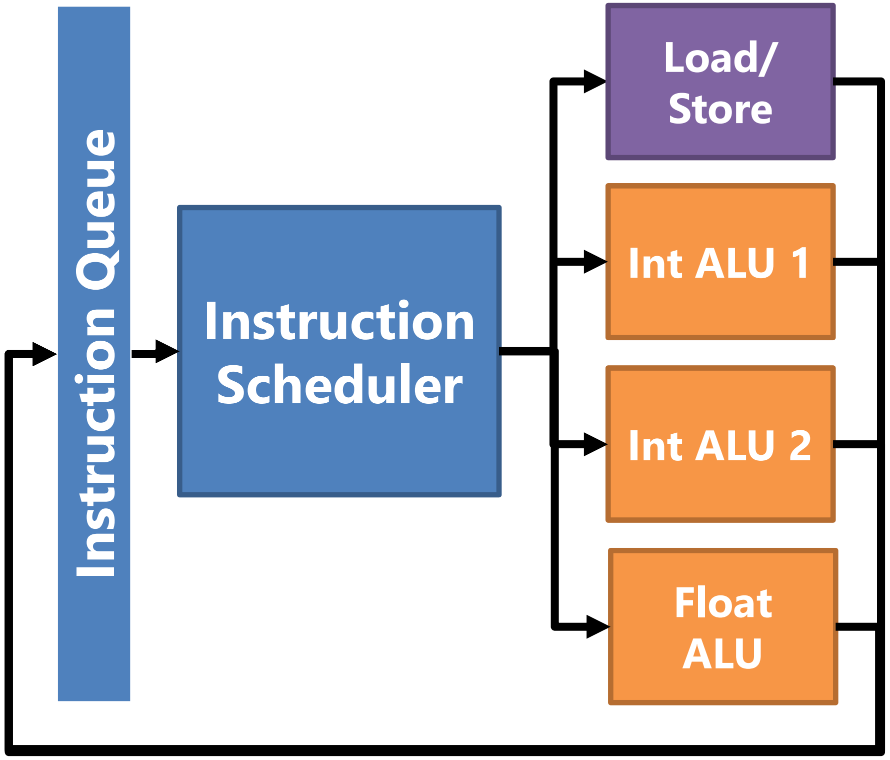


- At **ID**, instructions are decoded
  - And **dispatched** to the i-queue
- At **EX**, ready instructions are chosen from the instruction queue
  - Ready as in operands are available
  - And **issued** to an EX unit
- Insts start queueing up when insts fail to issue at a given cycle
  - Typically queue is always full
  - Pool of instructions to schedule

Scheduling the Instruction Queue

- Now we have pool of instructions. When do they become ready?
  - Ready operands and instructions are in `green`
  - Not ready operands and instructions are in `red`

```
  [Dispatched]
li t0, 1
li t1, 2
add t2, t0, t1
```

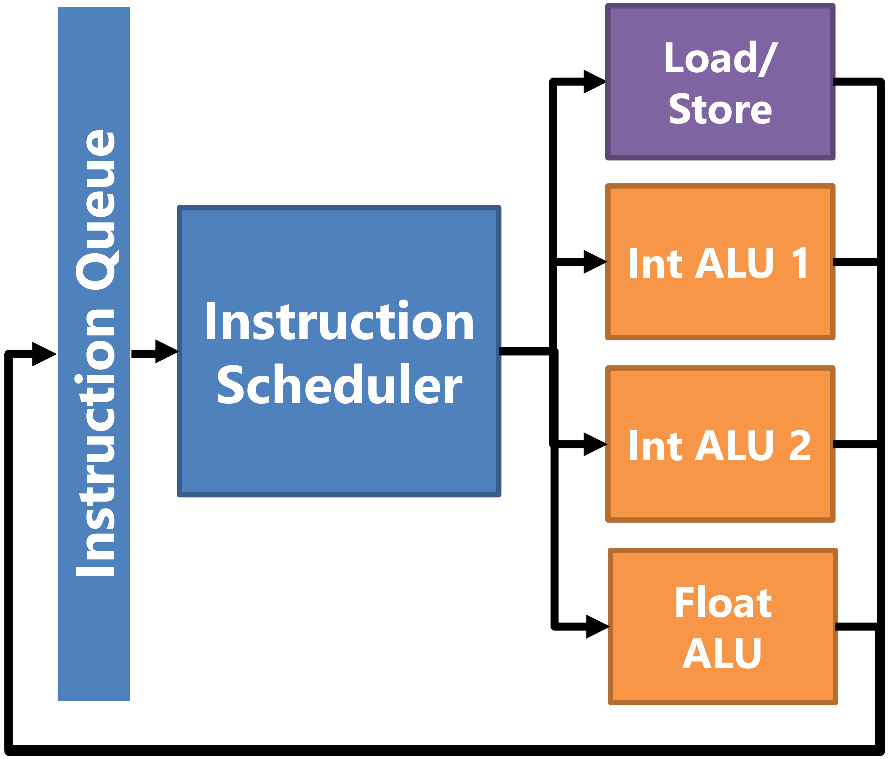


```
[Issued]
```

Scheduling the Instruction Queue

- Initially both `li t0, 1` and `li t1, 2` are ready
  - The `li` instruction does not have any register operands
  - Instruction scheduler has a choice of what to issue

```
[Dispatched]
li t0, 1
li t1, 2
add t2, t0, t1
```

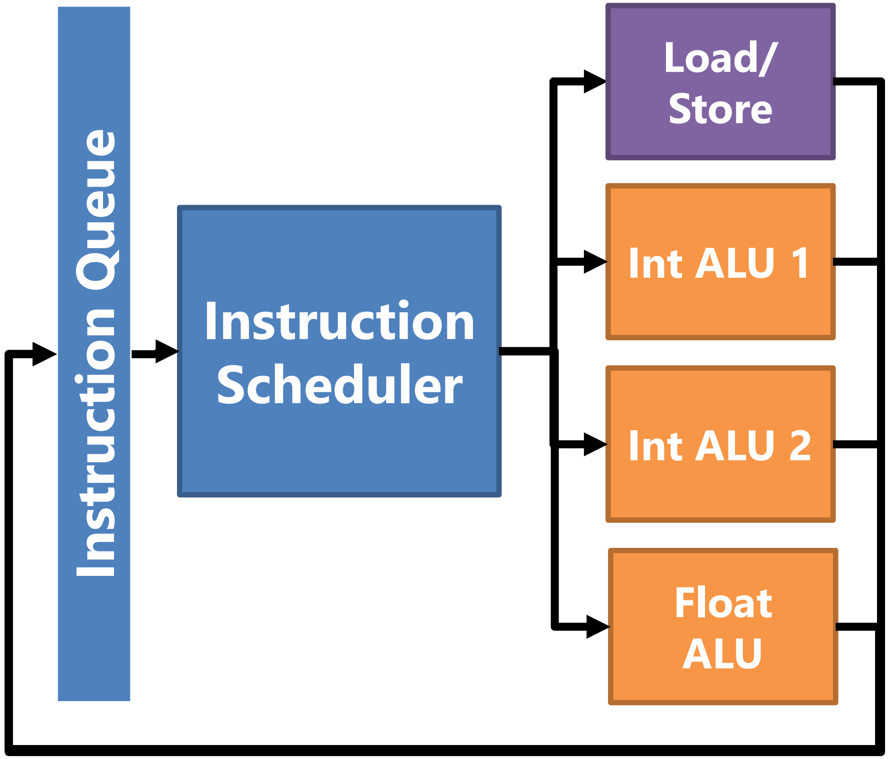


```
[Issued]
```

Scheduling the Instruction Queue

- Let's say the scheduler issues `li t1, 2` first
- Then the `t1` operand becomes ready after it completes

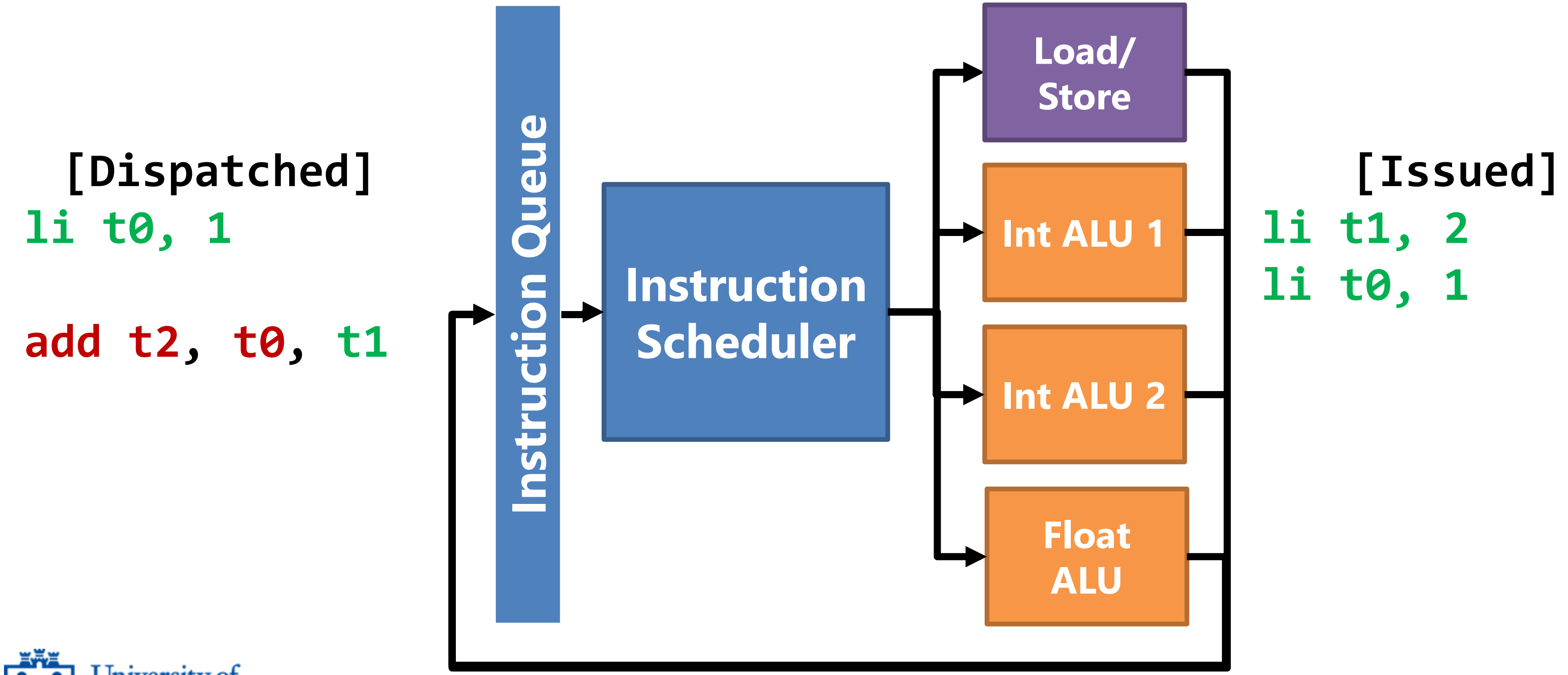
```
[Dispatched]
li t0, 1
li t1, 2
add t2, t0, t1
```



```
[Issued]
li t1, 2
```

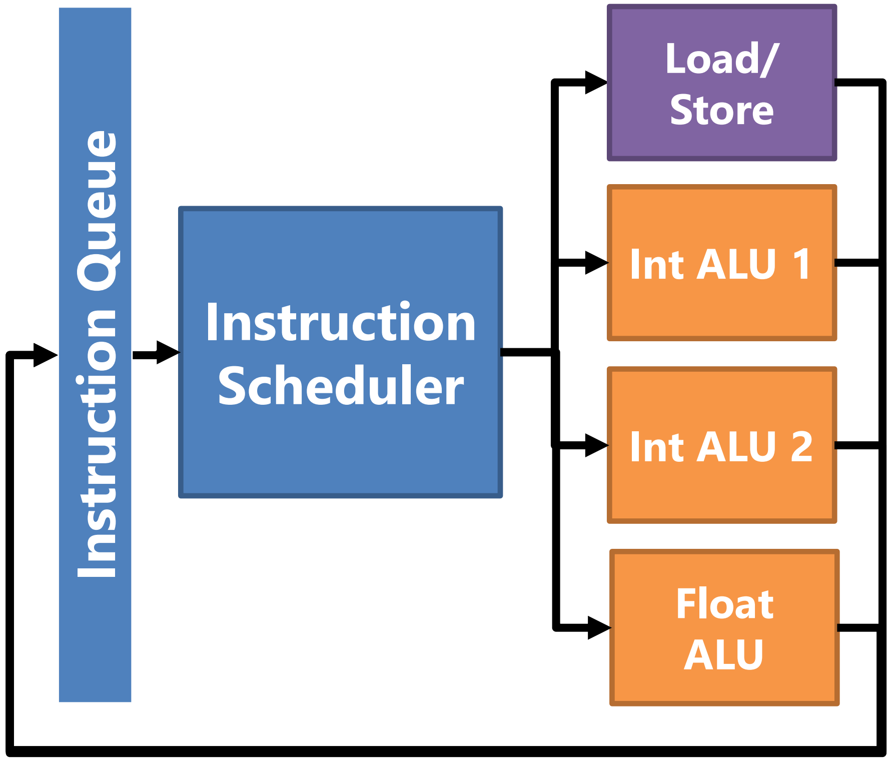
Scheduling the Instruction Queue

- Now the only ready instruction `li t0, 1` issues
- Then the `t0` operand becomes ready after it completes
- Now `add t2, t0, t1` is finally ready to issue



Scheduling the Instruction Queue

- And we are done!



[Dispatched]

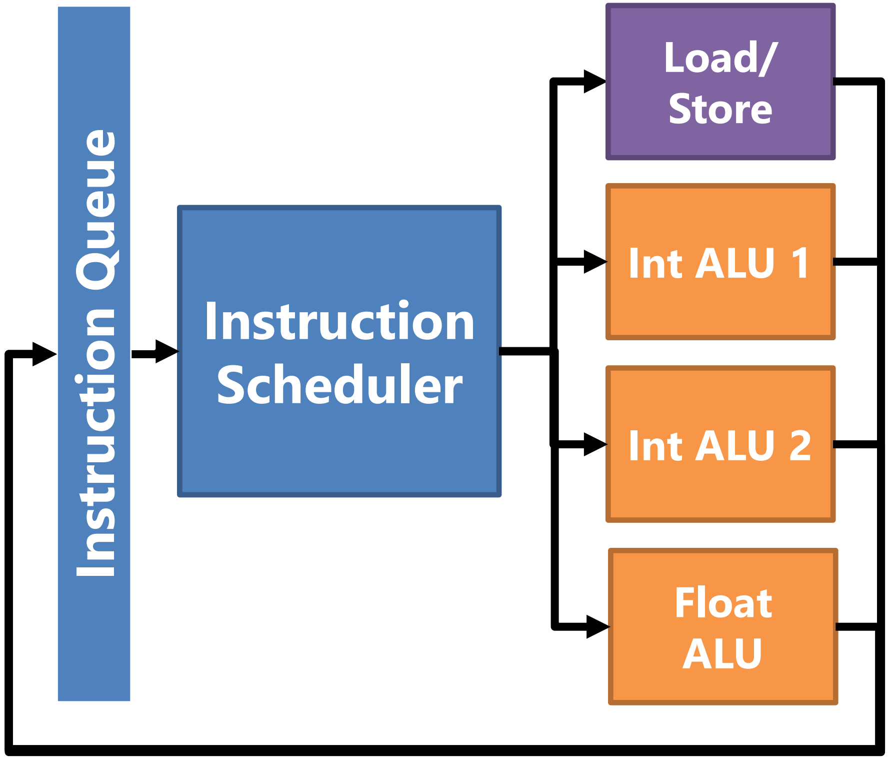
[Issued]

```
li t1, 2
li t0, 1
add t2, t0, t1
```

Scheduling the Instruction Queue

- Note how we reordered `li t0, 1` and `li t1, 2`
  - There are no dependencies between the two, so no issues
  - Also, RAW dependency with `add t2, t0, t1` was enforced

```
  [Dispatched]
li t0, 1
li t1, 2
add t2, t0, t1
```



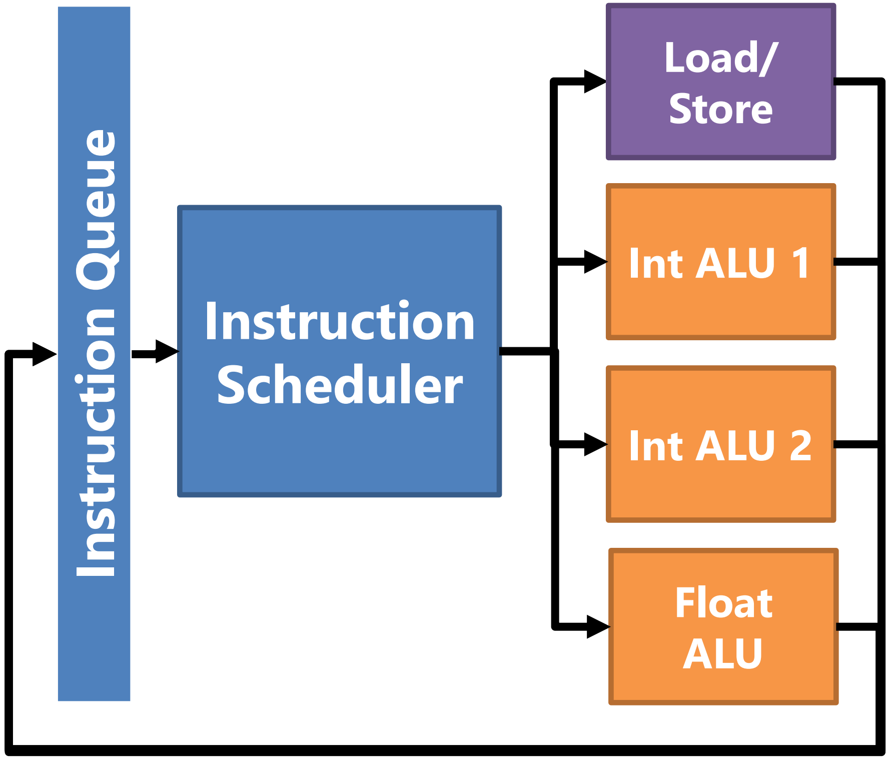
```
    [Issued]
li t1, 2
li t0, 1
add t2, t0, t1
```

What if we had a WAW dependency?

- Reordering `li t0, 1` and `li t0, 2` still allowed (both are ready)
    - Now `t2 = 4` in original code, but `t2 = 2` during execution!
    - How do we disallow this from happening?

```
[Dispatched]
li t0, 1
li t0, 2
add t2, t0, t0
```

WAW!



```
[Issued]
li t0, 2
li t0, 1
add t2, t0, t0
```

WAW and WAR dependencies are tricky

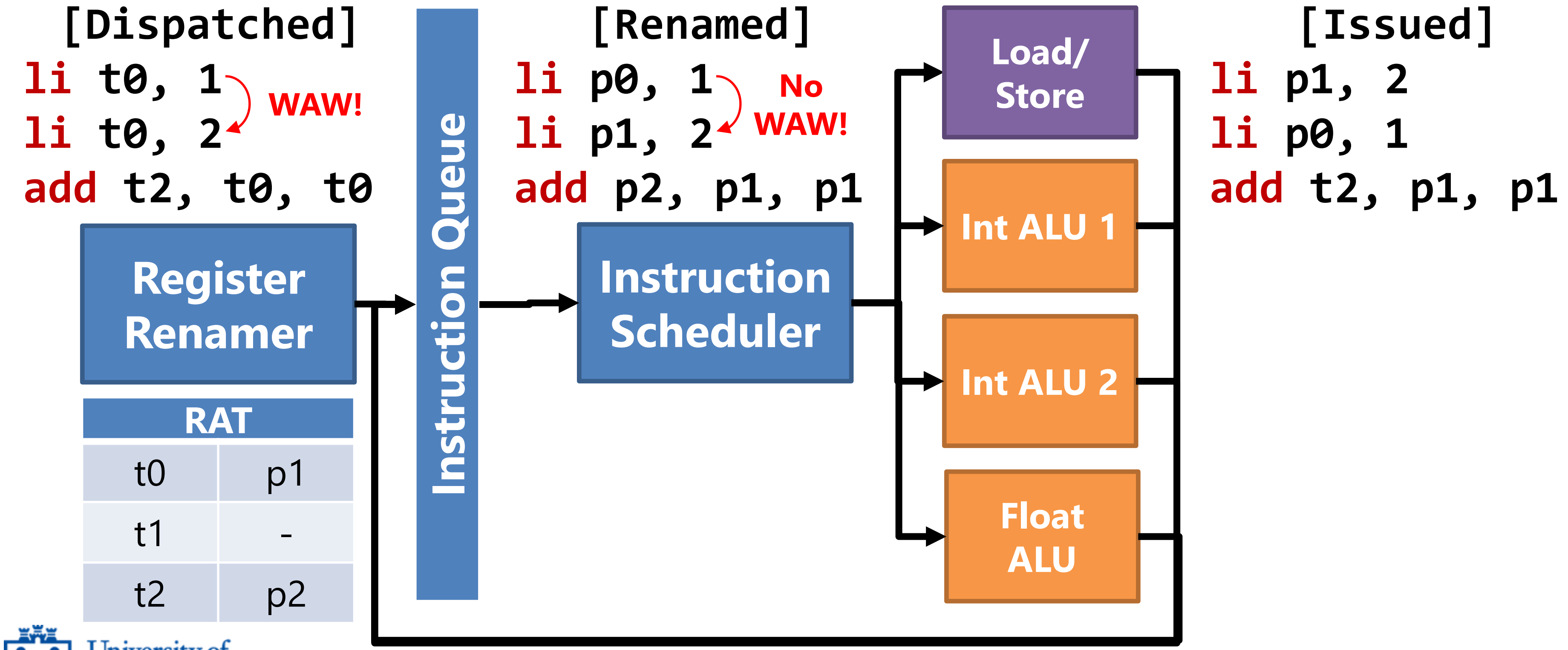
- RAW (true) dependencies are automatically enforced
  - Instructions cannot issue until all operands are ready (written)

- WAW and WAR dependencies are not enforced
  - There is no data passing between the two instructions
  - The two instructions can become ready in any order

- We could somehow enforce WAW and WAR dependencies
  - But there is a better solution: **register renaming**!
  - Remember? That's what the compiler did to remove WAW/WAR.

Register Renamer

- As soon as decode, **Register Renamer** renames all registers
  - Registers t0 and t2 are renamed registers p0, p1, p2
  - Done with the help of the **Register Alias Table (RAT)**



```
[Dispatched]
li t0, 1
li t0, 2
add t2, t0, t0
```

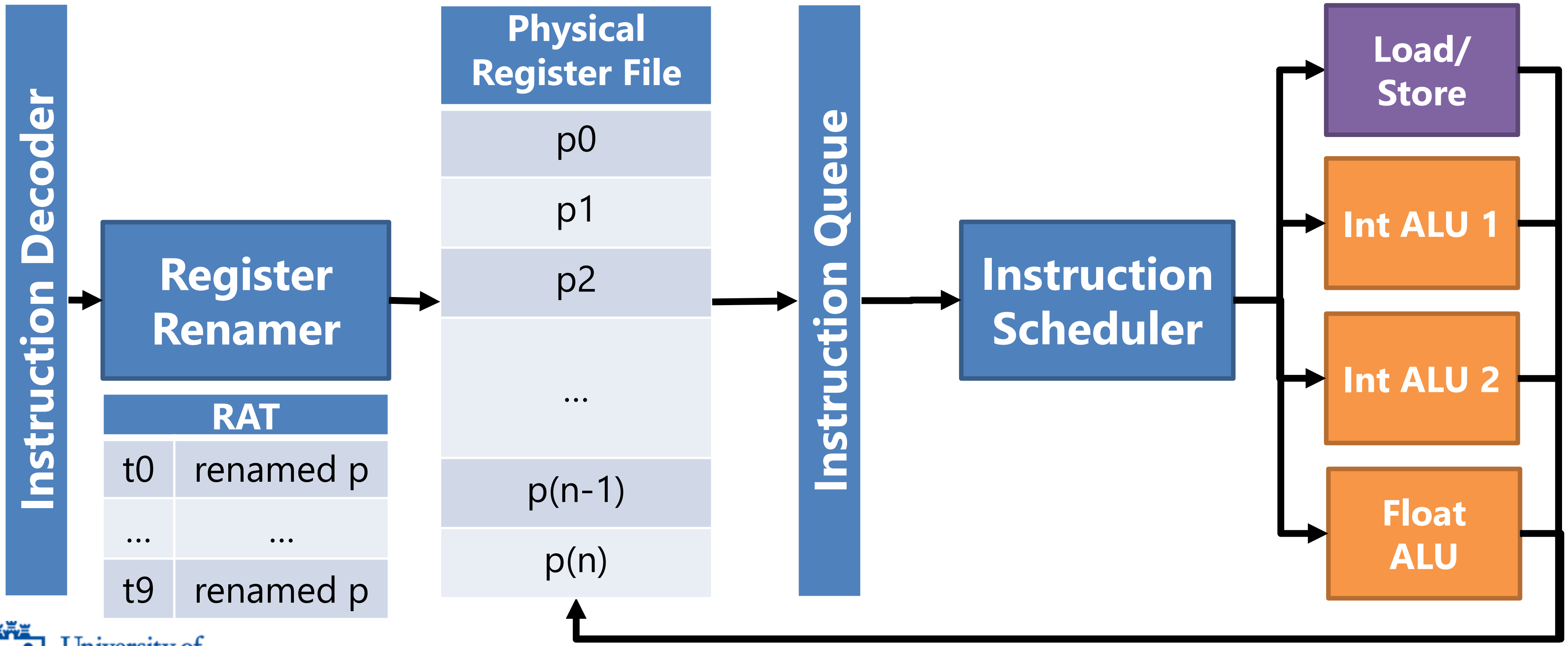
```
[Renamed]
li p0, 1
li p1, 2
add p2, p1, p1
```

```
[Issued]
li p1, 2
li p0, 1
add t2, p1, p1
```

RAT	
t0	p1
t1	-
t2	p2

Register Renaming and the Physical Register File

- Registers in ISA (t0, t1, t2) are called **architectural registers**
- Renamed registers (p0, p1, p2) are called **physical registers**
- Now the register file is filled with physical registers!



Physical Register File Management

- Number of physical registers is much larger than registers in ISA
  - To allow renaming of the same architectural register

- Instruction dispatch must stop when physical registers run out
  - Larger physical register file allows larger instruction window
  - Larger instruction window leads to more ILP through reordering

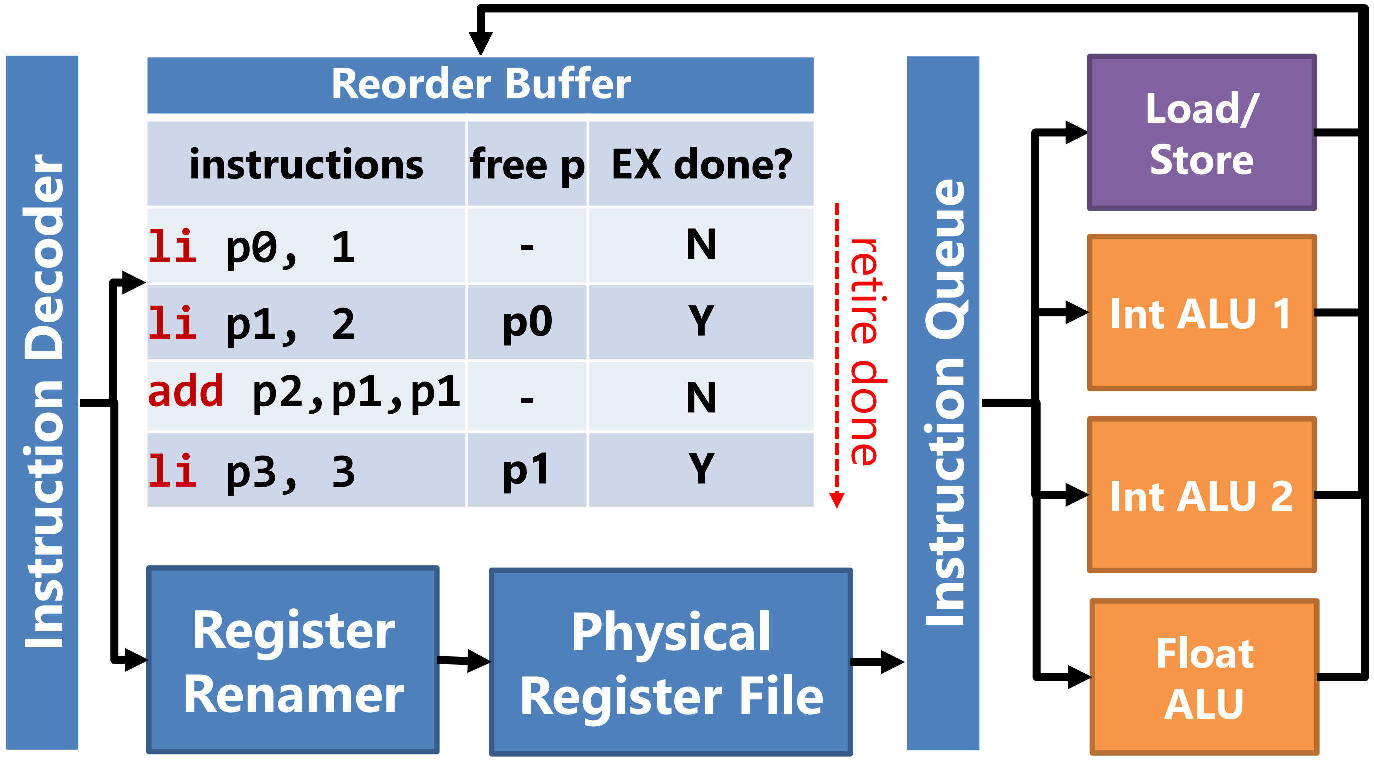
- Important to recycle physical registers efficiently
  - After use, physical registers must be returned to register pool
  - So that they can be allocated for the next renaming
  - How is this done? Through the **reorder buffer**.

University of Pittsburgh

Reorder Buffer

- Decoded instructions are **stored** in reorder buffer **in-order**
- Completed instructions are also **retired** from reorder buffer **in-order**
- Register in **free p** column is previous register used for the destination
  - When instruction retires, all previous instructions are done, so recycle p!

```
[Original Code]
li t0, 1
li t0, 2
add t2, t0, t0
li t0, 3
```



instructions	free p	EX done?
li p0, 1	-	N
li p1, 2	p0	Y
add p2,p1,p1	-	N
li p3, 3	p1	Y

The ARM Cortex-A8 architecture

- The ARM Cortex-A8 is an **in-order superscalar** processor
  - Notice the use of the **architectural register file**

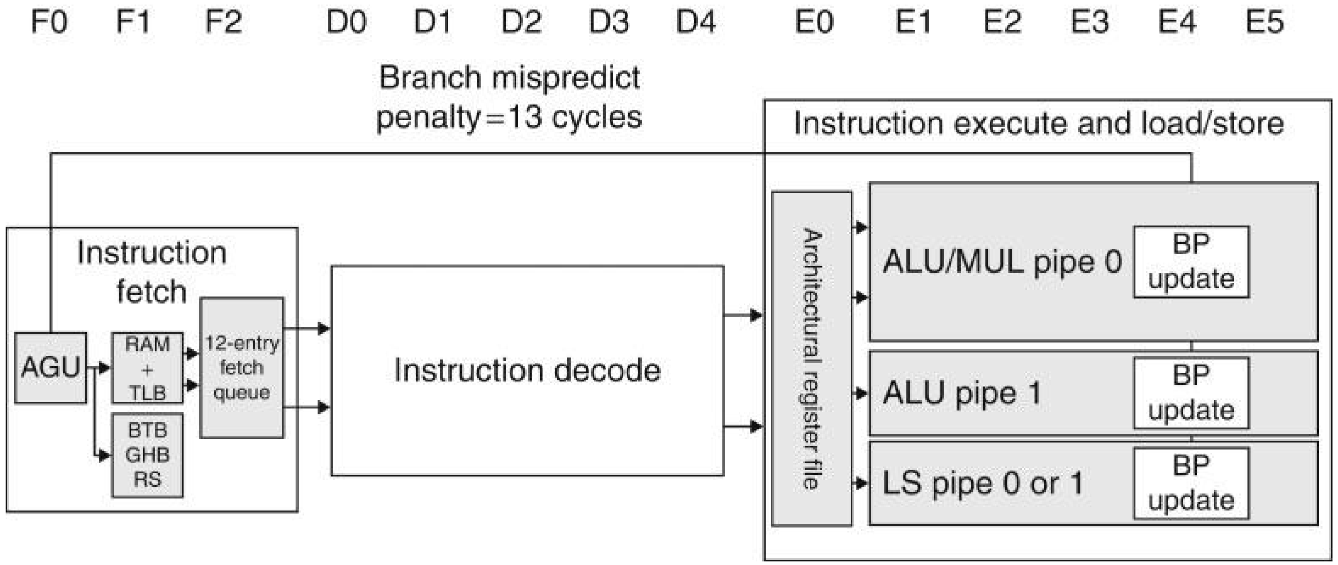
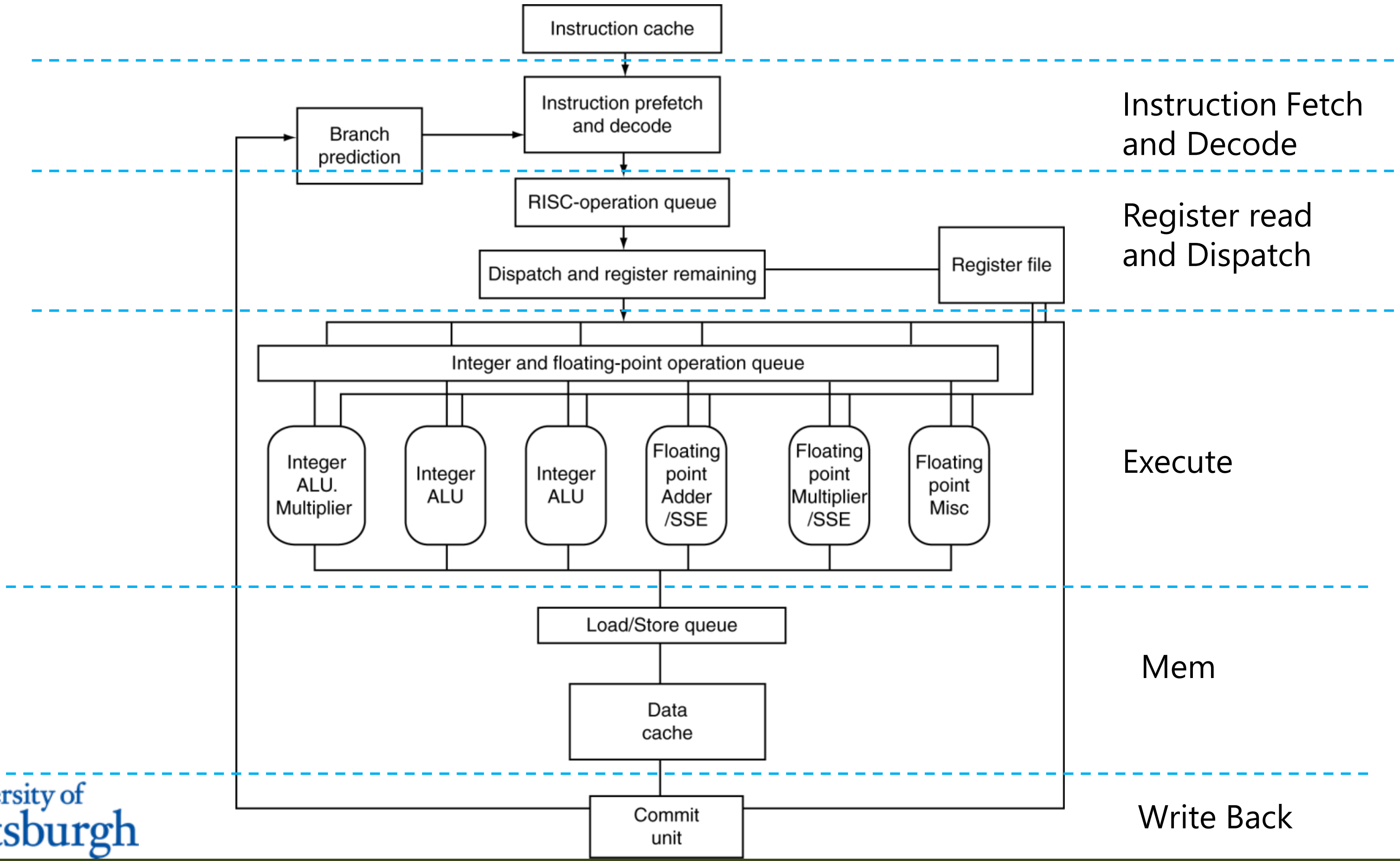


FIGURE 4.75 The A8 pipeline. The first three stages fetch instructions into a 12-entry instruction fetch buffer. The *Address Generation Unit* (AGU) uses a *Branch Target Buffer* (BTB), *Global History Buffer* (GHB), and a *Return Stack* (RS) to predict branches to try to keep the fetch queue full. Instruction decode is five stages and instruction execution is six stages.

The AMD Opteron X4 Microarchitecture

- The AMD Opteron is an **out-of-order superscalar** processor
  - **Commit unit** oversees retiring instructions from reorder buffer


Instruction cache
Instruction prefetch and decode
Branch prediction
RISC-operation queue
Dispatch and register remaining
Register file
Integer and floating-point operation queue
Integer ALU. Multiplier
Integer ALU
Integer ALU
Floating point Adder /SSE
Floating point Multiplier /SSE
Floating point Misc
Load/Store queue
Data cache
Commit unit
Instruction Fetch and Decode
Register read and Dispatch
Execute
Mem
Write Back

The Intel Core i7 architecture

- The Intel Core i7 is another **out-of-order superscalar** processor

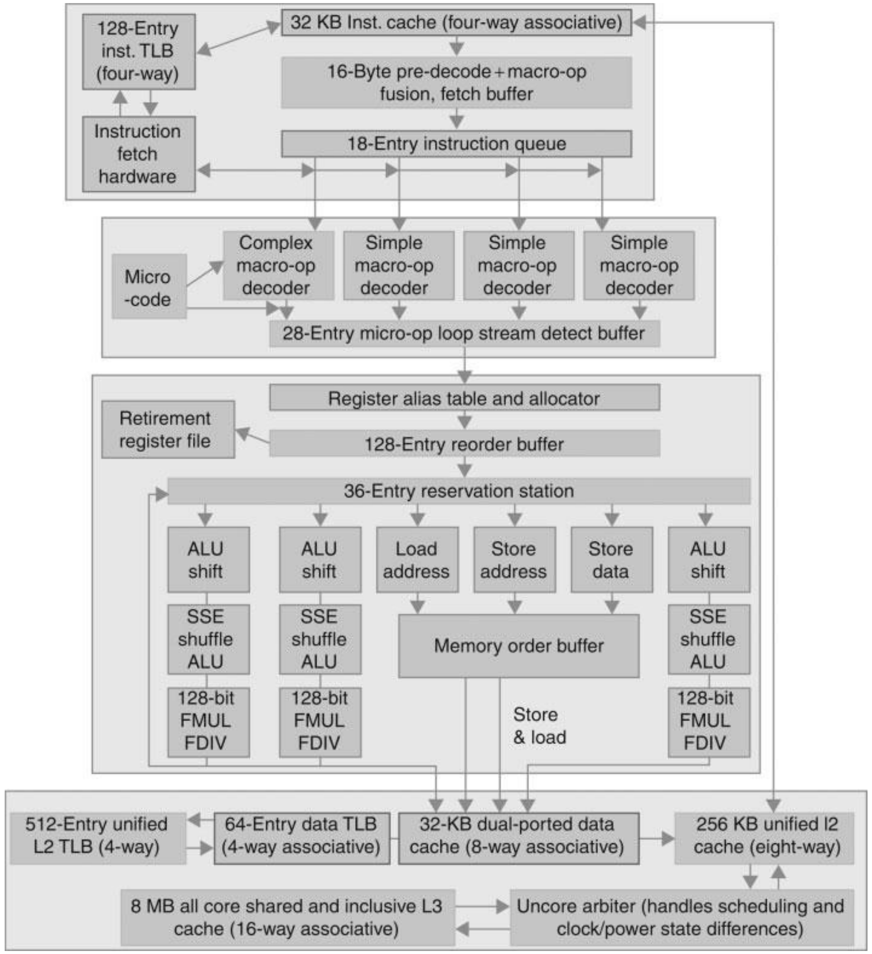

128-Entry inst. TLB (four-way)
32 KB Inst. cache (four-way associative)
16-Byte pre-decode + macro-op fusion, fetch buffer
Instruction fetch hardware
18-Entry instruction queue
Micro -code
Complex macro-op decoder
Simple macro-op decoder
Simple macro-op decoder
Simple macro-op decoder
28-Entry micro-op 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)

FIGURE 4.77 The Core i7 pipeline with memory components. The total pipeline depth is 14 stages, with branch mispredictions costing 17 clock cycles. This design can buffer 48 loads and 32 stores. The six independent units can begin execution of a ready RISC operation each clock cycle.

Static vs. Dynamic Scheduling