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

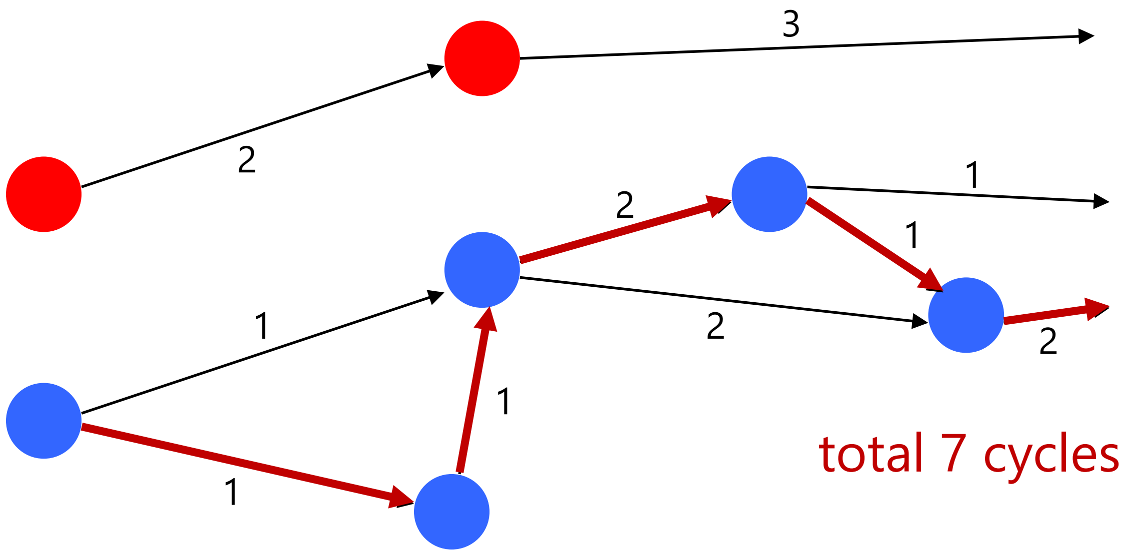
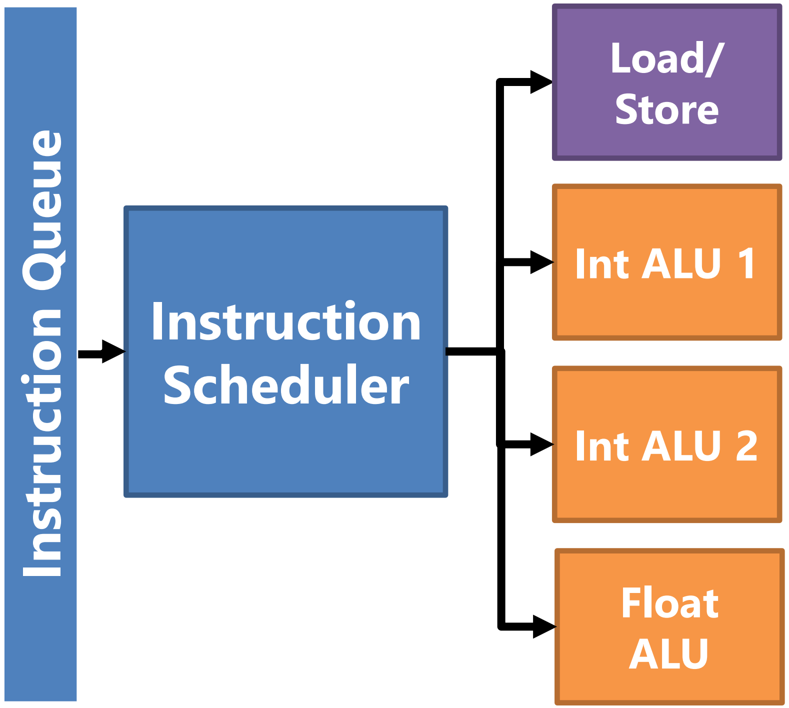


illustration courtesy of Dr. Melhem

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***



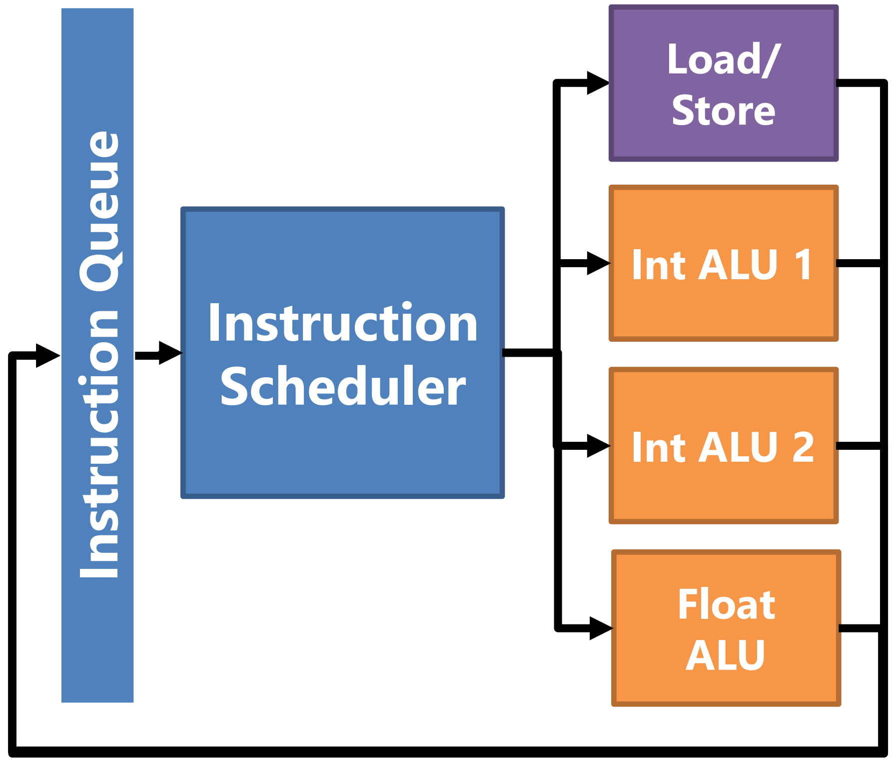
- At **ID**, instructions are decoded
  - And **dispatched** to the i-queue
- At **EX**, ready instructions are chosen from the instruction queue
  - Ready as in operands are available
  - And **issued** to an EX unit
- Insts start queueing up when insts fail to issue at a given cycle
  - Typically queue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

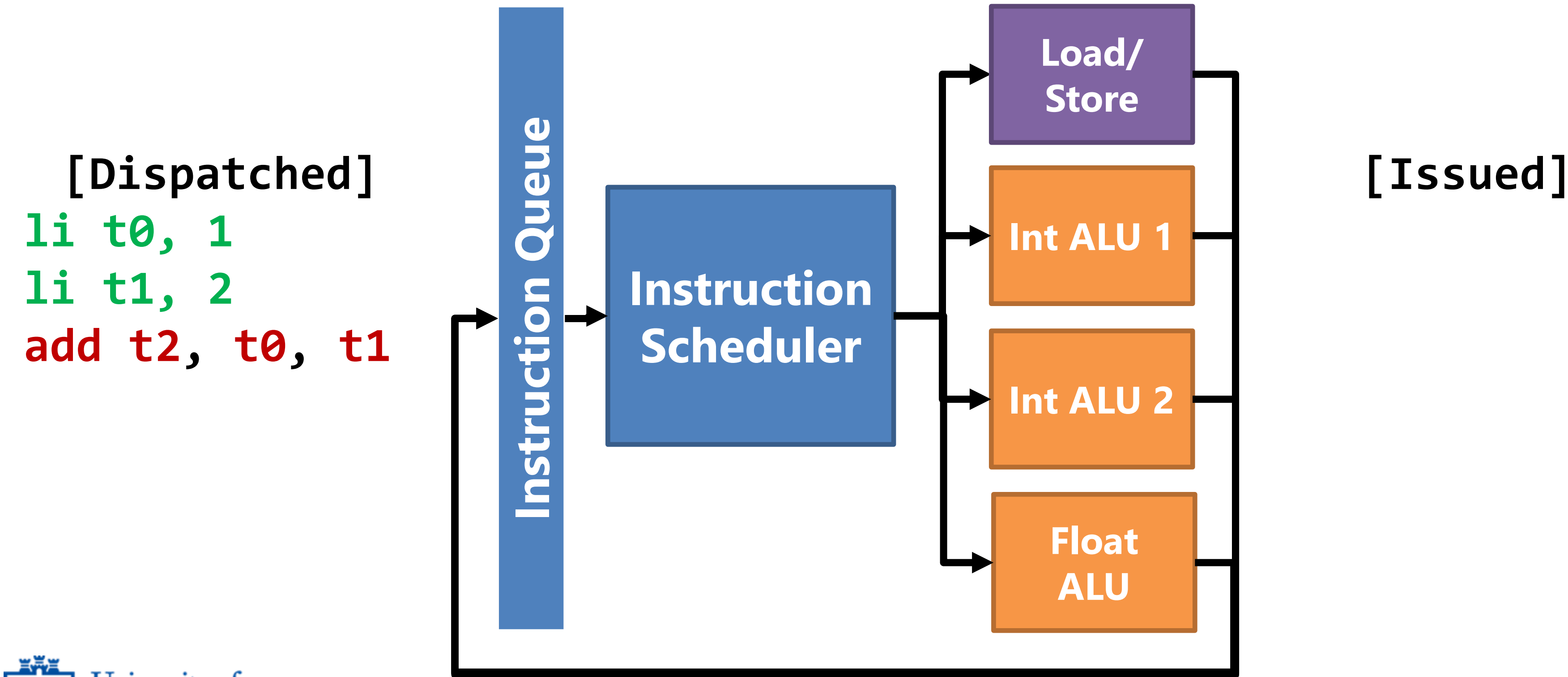


[Issued]

University of Pittsburgh

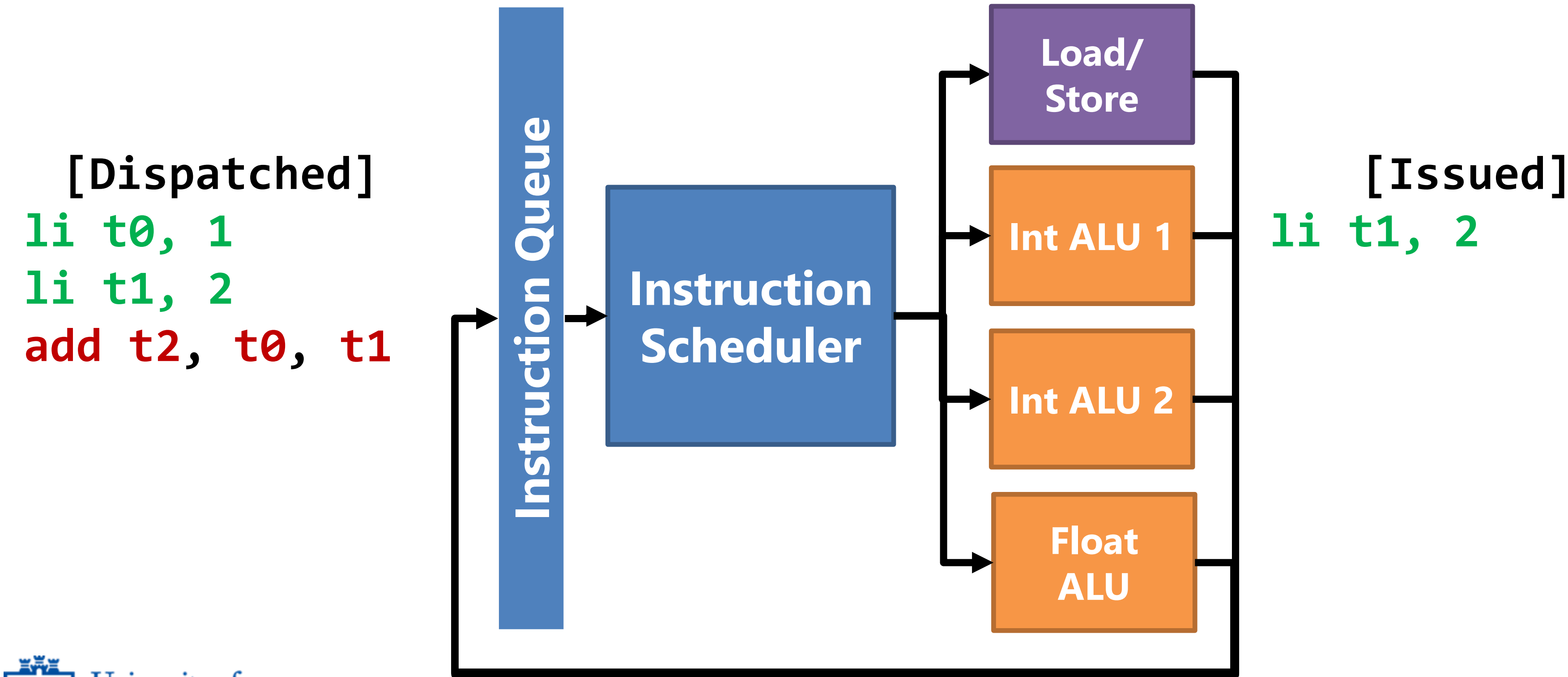
Scheduling the Instruction Queue

- 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



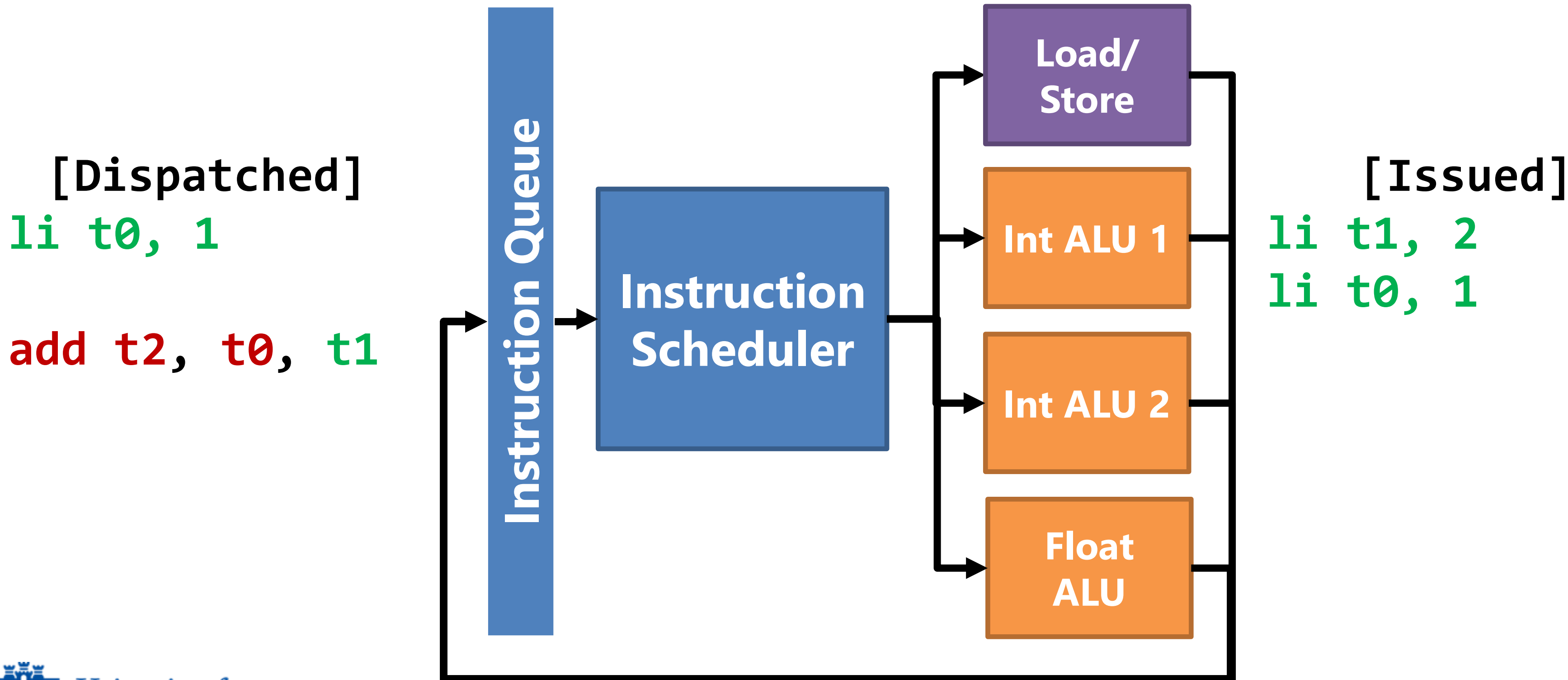
Scheduling the Instruction Queue

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



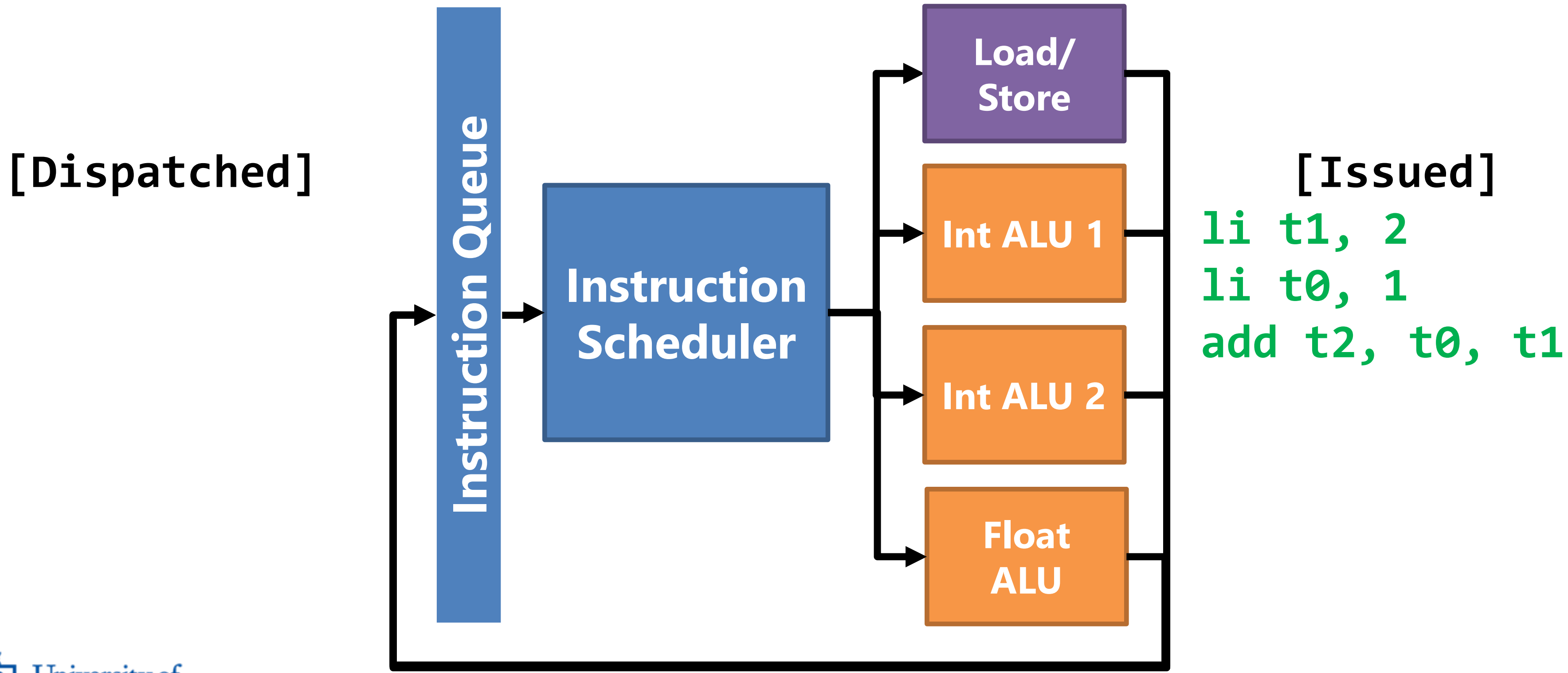
Scheduling the Instruction Queue

- 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

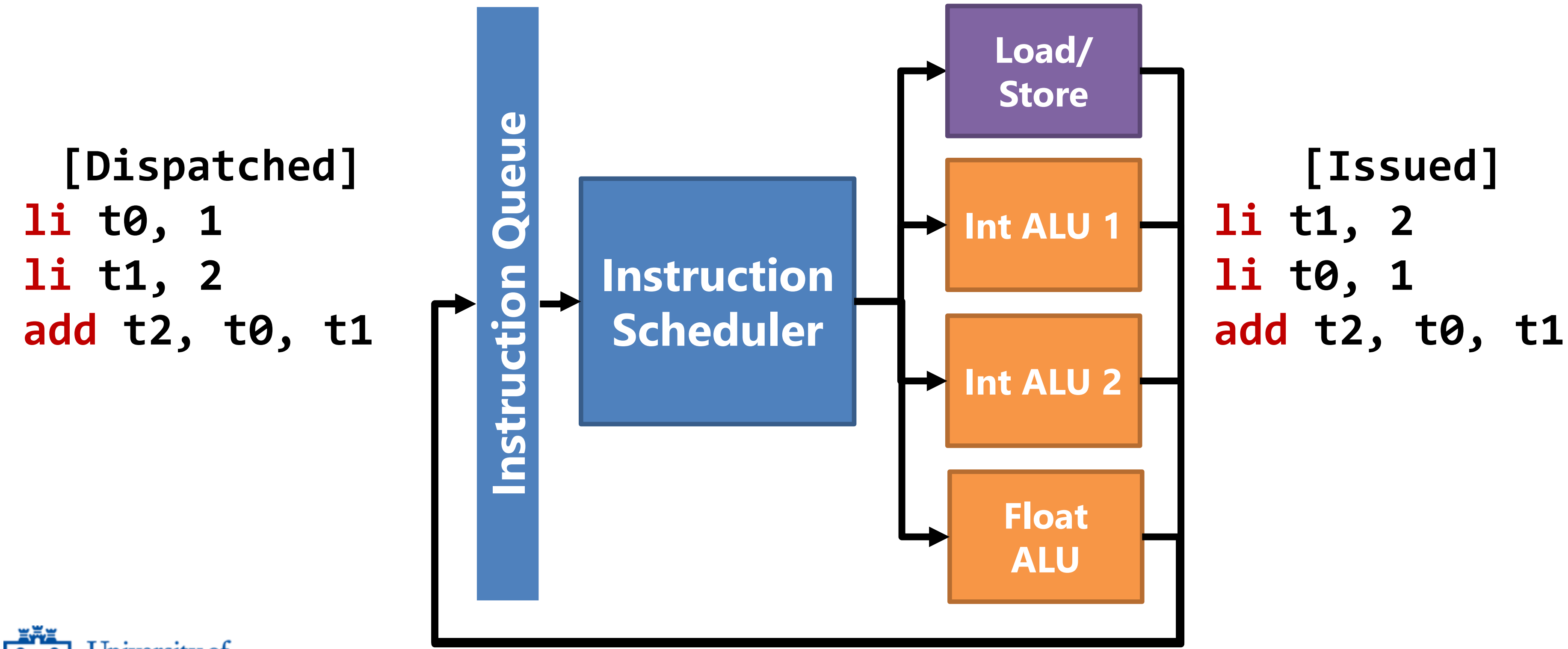
- And we are done!



University of Pittsburgh

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



University of Pittsburgh

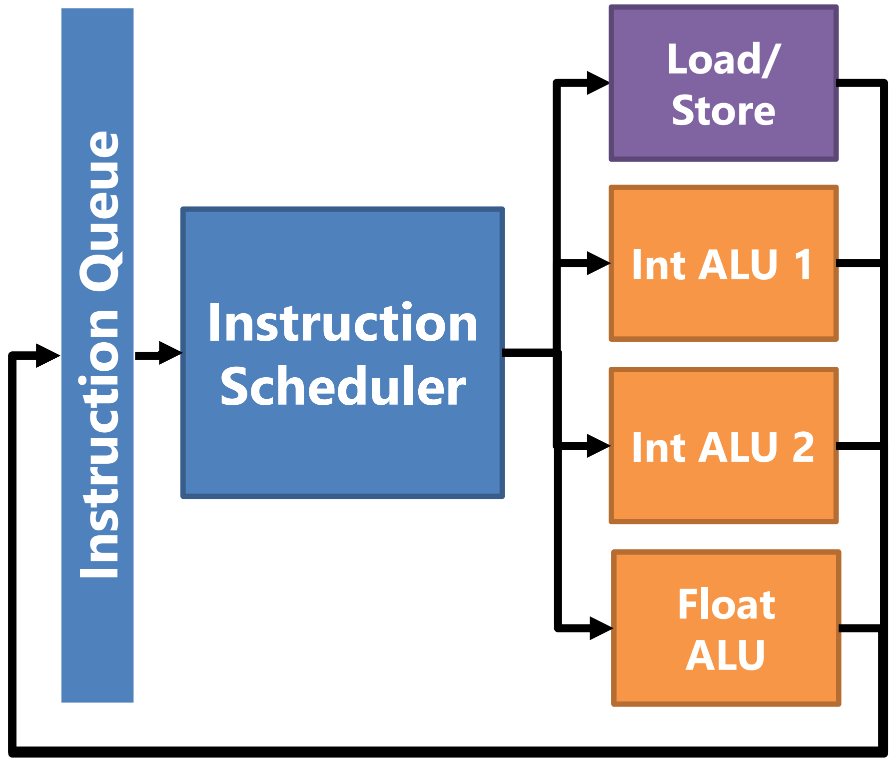
What if we had a WAW dependency?

- 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
```

University of Pittsburgh

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 and the RAT

- As soon as decode, **Register Renamer** renames all registers
  - Done with the help of the **Register Alias Table (RAT)**
  - **RAT** is current mapping between **architectural** and **physical** registers
    - Architectural registers: Registers in ISA used in programs (t0, t1, t2, …)
    - Physical registers: Renamed registers used in processor (p0, p1, p2, …)

[Decoded]

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

WAW!

Register Renamer

RAT	
t0	p1
t1	-
t2	p2
…	…

All 32 MIPS registers

[Dispatched]

```
li p0, 1
li p1, 2
add p2, p1, p1
```

No WAW!

Register Renamer and the RAT

- So how does the RAT work?
- Initially, no assignments have been done, so mapping is empty.

All 32 MIPS registers

RAT	
t0	-
t1	-
t2	-
…	…

Register Renamer and the RAT

1. `li t0, 1` is decoded, destination **`t0`** is renamed to **`p0`**

```
[Decoded]
li t0, 1
```

Register Renamer

```
[Dispatched]
li p0, 1
```

All 32 MIPS registers

RAT	
t0	p0
t1	-
t2	-
…	…

University of Pittsburgh

Register Renamer and the RAT

1. `li t0, 1` is decoded, destination **`t0`** is renamed to **`p0`**
2. `li t0, 2` is decoded, destination **`t0`** is renamed to **`p1`**

```
[Decoded]
li t0, 1
li t0, 2
```

Register Renamer

```
[Dispatched]
li p0, 1
li p1, 2
```

RAT	
t0	p1
t1	-
t2	-
…	…

All 32 MIPS registers

University of Pittsburgh

Register Renamer and the RAT

1. **`li t0, 1`** is decoded, destination **`t0`** is renamed to **`p0`**
2. **`li t0, 2`** is decoded, destination **`t0`** is renamed to **`p1`**
3. **`add t2, t0, t0`** is decoded:
    - Two **`t0`** input registers use current mapping **`p1`**
    - Destination register **`t2`** is renamed to **`p2`**

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

Register Renamer

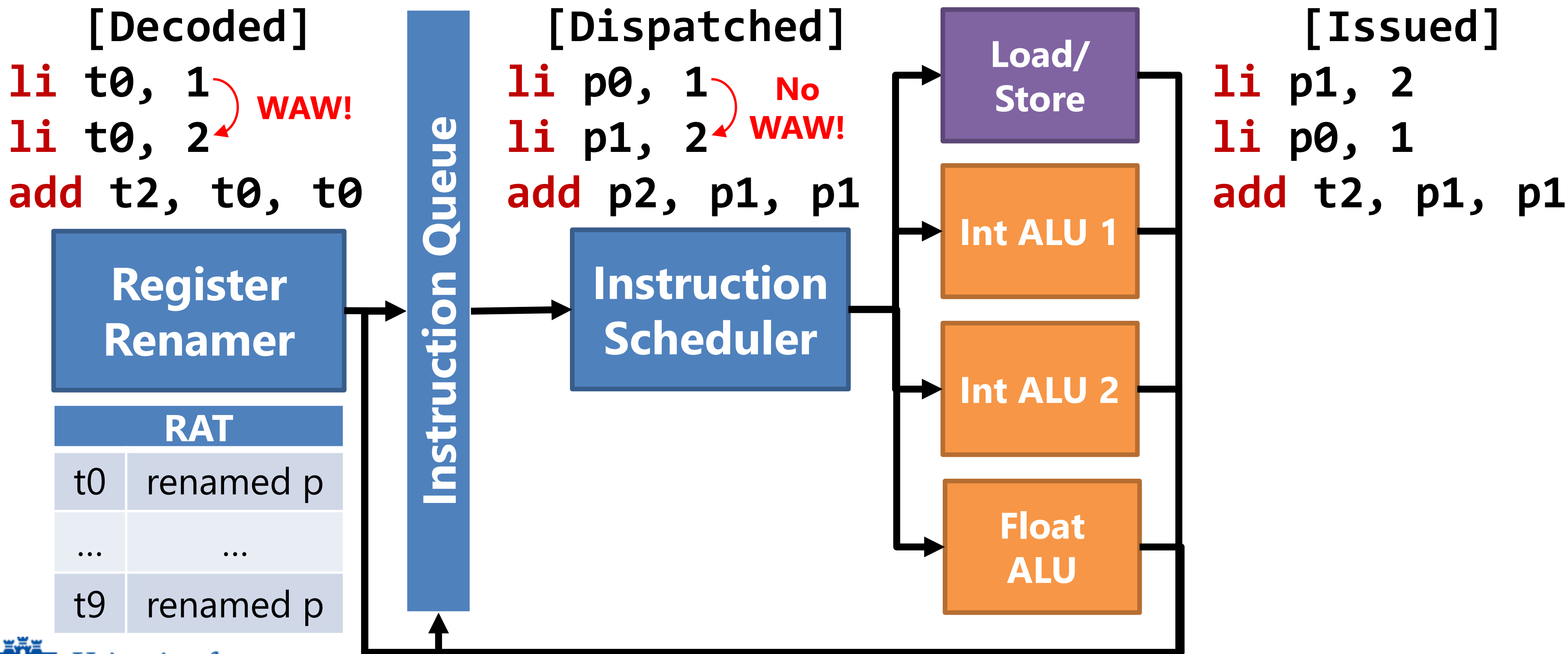
```
[Dispatched]
li p0, 1
li p1, 2
add p2, p1, p1
```

RAT	
t0	p1
t1	-
t2	p2
…	…

All 32 MIPS registers

Register Rename Removes all WAW/WAR Deps

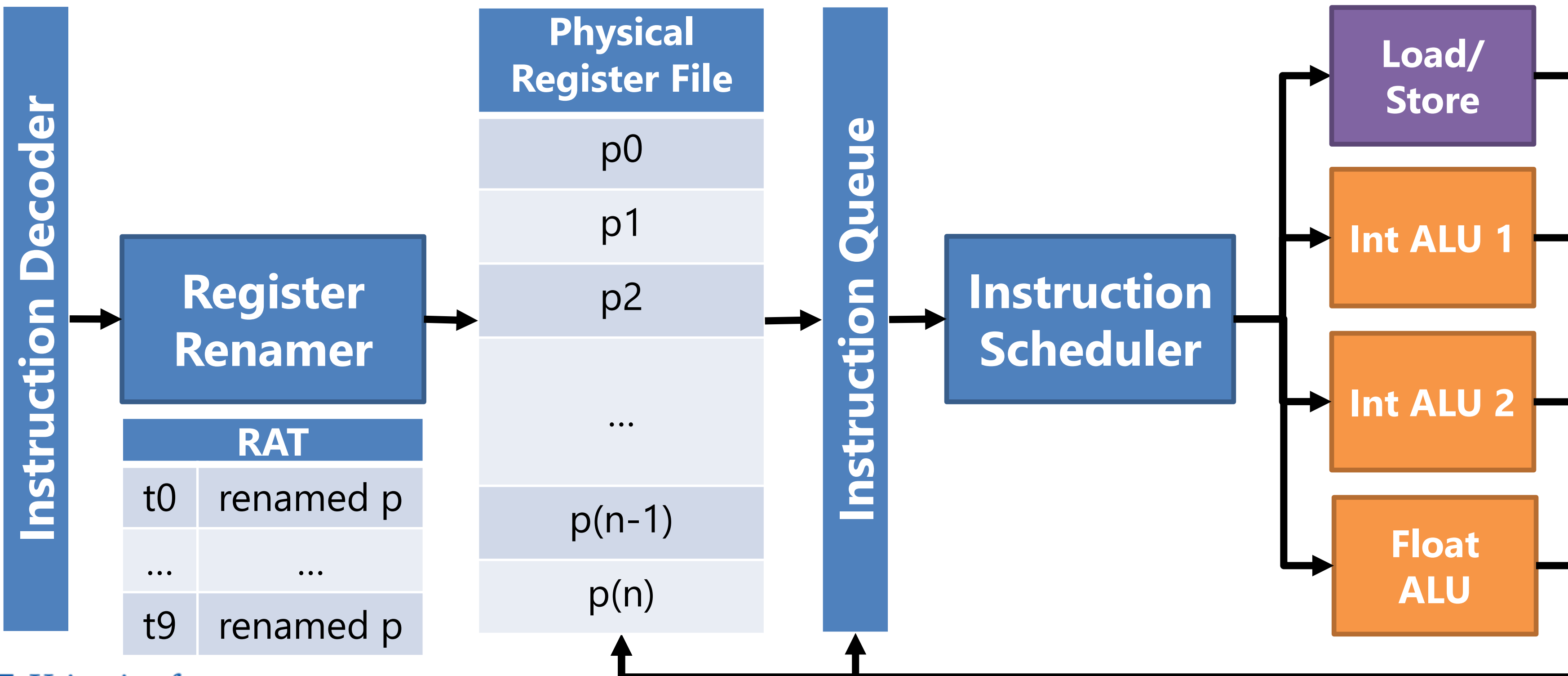
- By the time instructions are dispatched to i-queue
  - All architectural registers have been renamed to physical registers
  - All WAR and WAR dependencies have been removed



University of Pittsburgh

All Computation Done using Physical Registers

- Now ID stage (dispatch) reads registers for physical register file
- All data forwarding also done based on physical registers



University of Pittsburgh

Limits on IPC

- We already discussed limits on pipelining.
- Time to discuss limits on IPC improvements through wide-issue!

- There is a **fundamental limit** to achievable IPC
  - Amount of **ILP (Instruction Level Parallelism)** in code
  - Remember the data dependence graph?
  - For instructions to be executed in the same cycle:
    - They must be data independent (no RAW true dependencies)
  - How about control dependencies?
    - Not a fundamental limit → can elide using branch prediction

- ILP is a **property of the program**, not the processor
  - This limit applies to both VLIW and superscalar processors

Limits on IPC

- After renaming, theoretical limit of IPC is 35 ~ 4003!

Benchmark	IPC No Renaming	IPC Register Renaming	IPC Memory Renaming	IPC r29 Removed	IPC 10K Window
compress95	3.12	26.25	73.88	226.33	18.89
cc1	3.61	39.79	41.63	239.96	86.45
go	2.50	49.15	53.77	141.46	70.71
ijpeg	2.41	55.47	93.60	94.11	52.94
li	3.56	19.60	19.61	81.45	27.70
m88ksim	2.76	19.93	62.06	363.26	20.50
perl	3.47	82.01	127.57	153.05	128.84
vortex	4.57	26.26	26.27	271.97	92.04
applu	2.82	106.65	2037.61	2076.06	78.67
apsi	3.6	54.89	183.44	1224.86	79.56
fpppp	3.33	103.62	774.13	1837.96	134.62
hydro2d	3.09	144.80	147.67	242.08	52.14
mgrid	3.34	1876.11	3933.03	4003.44	286.48
su2cor	3.22	38.21	34.81	55.56	47.60
swim	3.10	112.08	112.08	275.21	89.15
tomcatv	3.61	32.85	61.47	119.67	58.91
turb3d	3.42	370.98	482.24	3652.46	0
wave5	3.25	29.28	35.71	35.71	0

Matthew Postiff et al. "The Limits of Instruction Level Parallelism in SPEC95 Applications". ACM SIGARCH Computer Architecture News, 1999

Limits on IPC

- Achieving the theoretical limit would be awesome
  - But in real life, processors are no more than 10-wide

- Practical limits on IPC
  - Scheduling window size (amount of ILP you can extract)
    - Instruction queue size
    - Physical register file size
  - Number and types of execution units (i.e. width of processor)
  - Increasing complexity and its impact on cycle time
    - Increase in ports required in register file and memory
    - Time to search i-queue and update ready operands

Exceptions

Exceptions Review

- **Exception**: an event which causes the CPU to stop the normal flow of execution and go somewhere else (the exception handler)

- There are mainly two causes of exceptions:
    - **Software exceptions** (or **traps**): Triggered by a program instruction
        - Trap instruction: typically used to call OS routines (system calls)
        - Page fault: instruction accessed a page not mapped to memory
        - Divide-by-0: instruction performed a divide-by-0 arithmetic
        - Arithmetic overflow: instruction overflowed MAX_INT of register
    - **Hardware exceptions** (or **interrupts**): Triggered by hardware event
        - User has typed on the keyboard
        - A network packet has arrived
        - A file block read has completed

- In all cases, the exception handler must be invoked

University of Pittsburgh

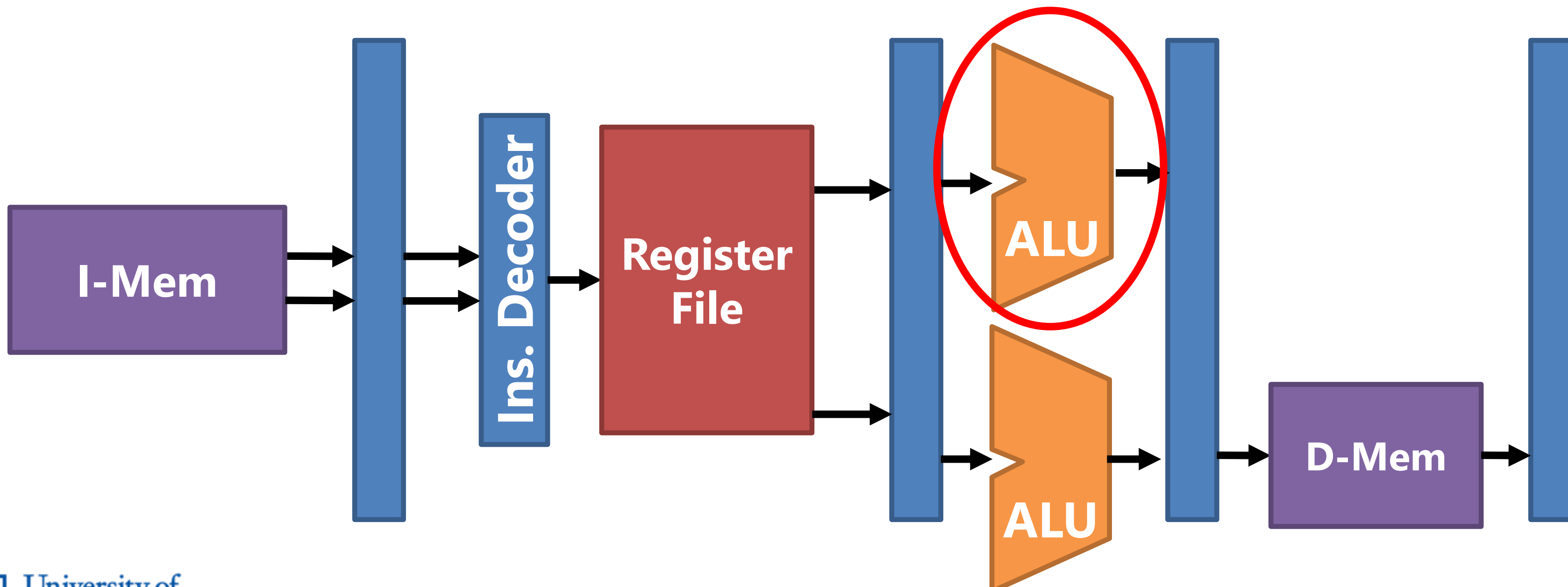
Handling exceptions

- What happens when an exception is triggered:
  1. Processor stops execution of user program.
  2. Processor stores information about exception (cause, PC).
  3. Processor jumps to the exception handler (code inside OS).
  4. Handler **creates backup** of program **register values** in memory.
  5. Handler inspects exception info and handles it accordingly.
     - While overwriting some of the registers that were backed up.
  6. Handler **restores** program **register values** from memory.
  7. Processor resumes execution of user program.

- Processor must provide precise register values at point of exception
  - Otherwise, when processor resumes, program will malfunction
  - Guaranteeing this is called a **precise exception**

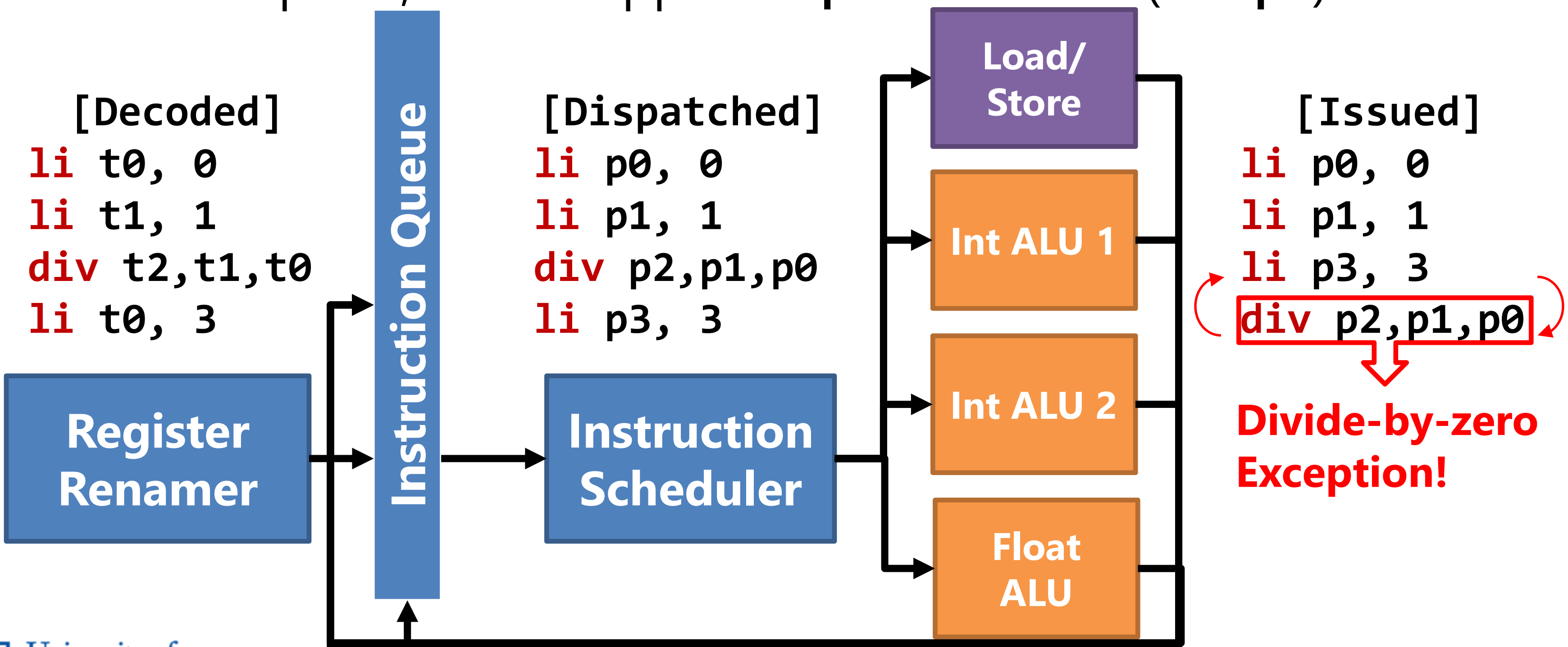
Precise Exceptions in In-order Processors is Easy

- Exceptions are typically triggered at the EX stage
  - Stage where all arithmetic happens as well as address calculations
- On exception, flush EX and all previous stages (ID and IF)
  - Since in-order, guarantees instructions following EX do not writeback
  - Only state leading up to instruction at EX will be written to reg / mem



Precise Exceptions in Out-of-order Processors is Hard

- Suppose `div t2,t1,t0` and `li t0, 3` issue out-of-order as below
  - `div p2,p1,p0` triggers a divide-by-zero exception (`p0 = 0`)
  - But at point of exception, `t0` appears to be `3` due to `li p3, 3`!
    - At that point, `t0` is mapped to `p3` in the RAT (not `p0`)



University of Pittsburgh

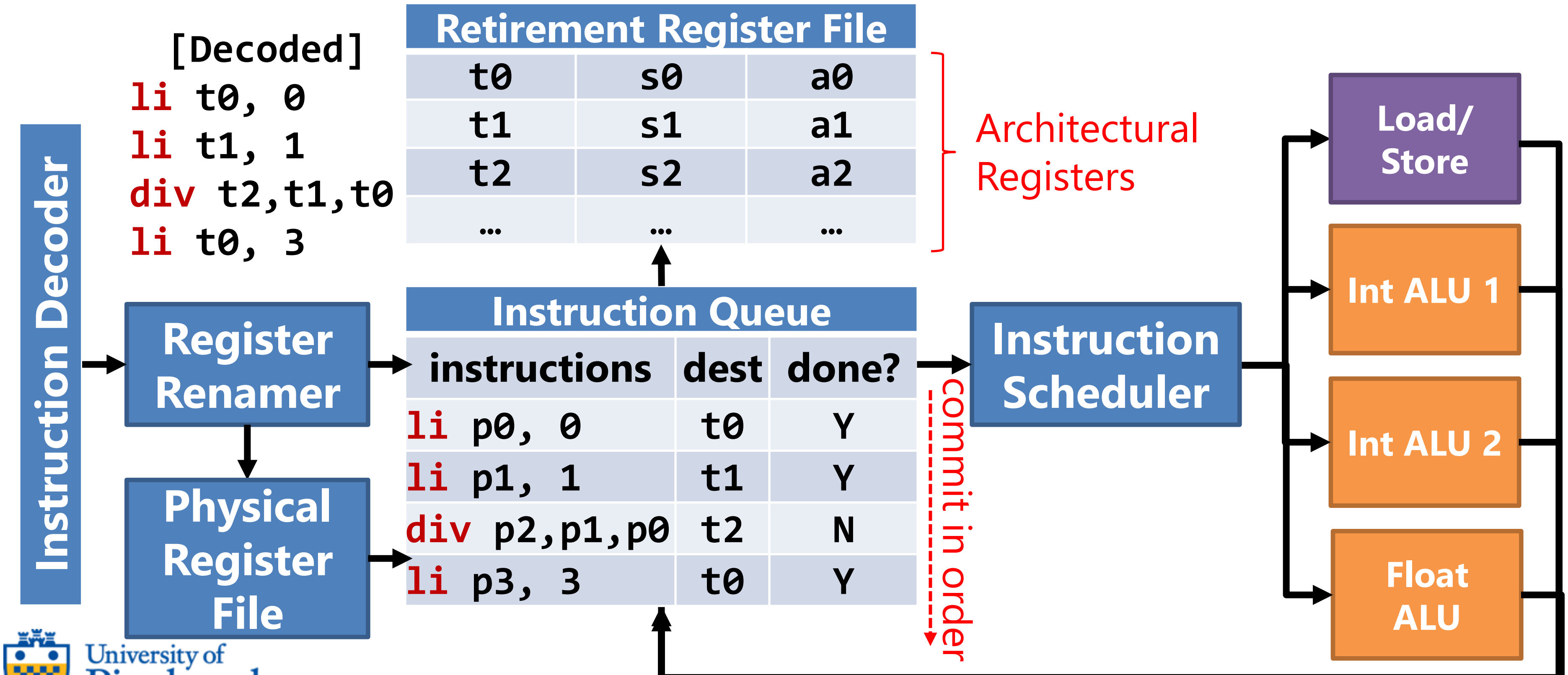
Precise Exceptions in Out-of-order Processors is Hard

- This is the challenge with out-of-order processors
  - Instructions execute and complete out-of-order
  - For precise exceptions, instructions must appear to complete in-order

- Solution: update architectural state in-order
  - Architectural state is the state visible to the ISA
    - State in 32 MIPS registers (t0, t1, t2, …)
    - State in data memory
  - When instructions execute, have them only update "internal" state
    - Physical registers
    - Store queue (MEM queues up stores instead of performing them)
  - Internal state is transferred to visible state during in-order **commit**
    - Only completed instructions can commit
    - Physical registers are copied to architectural registers
    - Store queue entries are written to memory

In-order Commit

- Decoded instructions are **stored** to i-queue **in-order**
- Instructions **execute out-of-order** (updating done? field)
- Done instructions **commit** from i-queue **in-order** to Retirement Register File



Real Life Superscalars

The ARM Cortex-A8 architecture

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

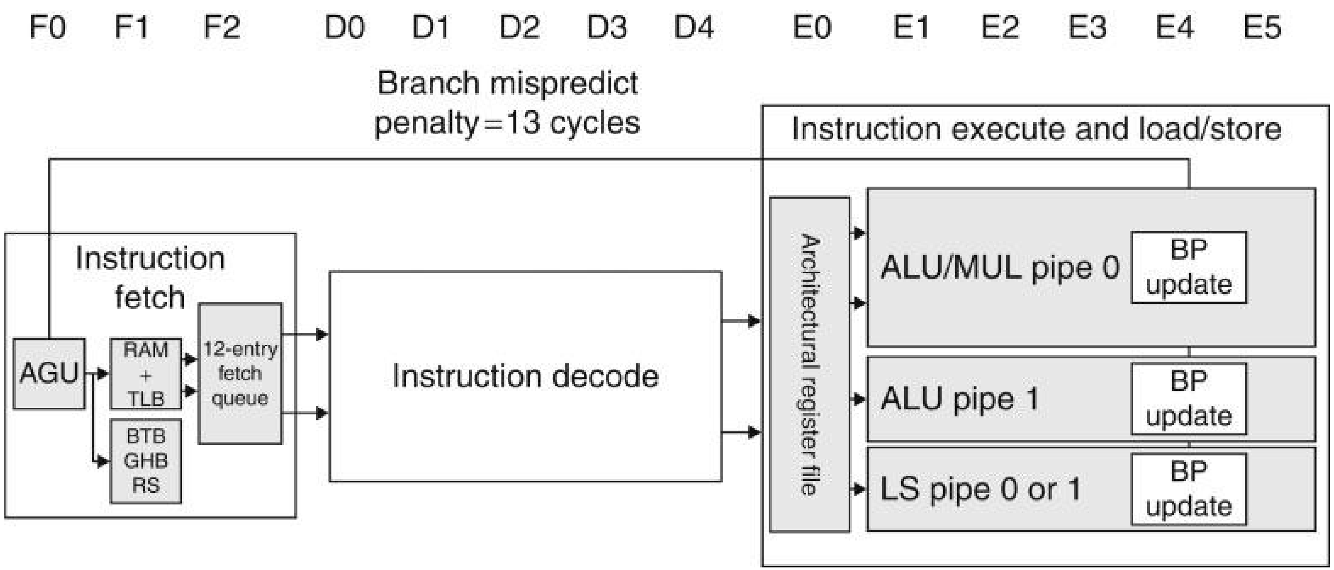
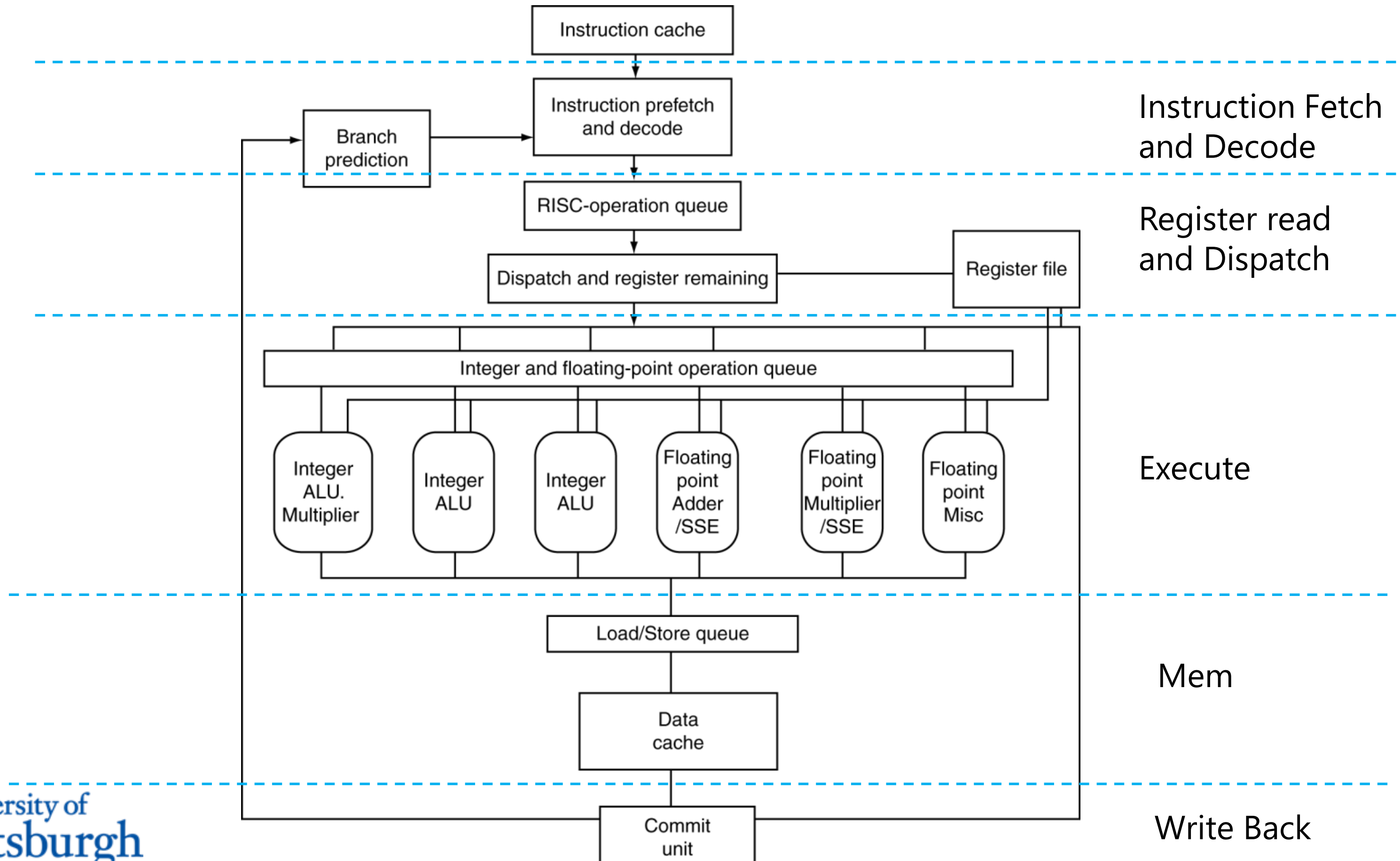


FIGURE 4.75 The A8 pipeline. The first three stages fetch instructions into a 12-entry instruction fetch buffer. The *Address Generation Unit* (AGU) uses a *Branch Target Buffer* (BTB), *Global History Buffer* (GHB), and a *Return Stack* (RS) to predict branches to try to keep the fetch queue full. Instruction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



University of Pittsburgh

The Intel Core i7 architecture

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

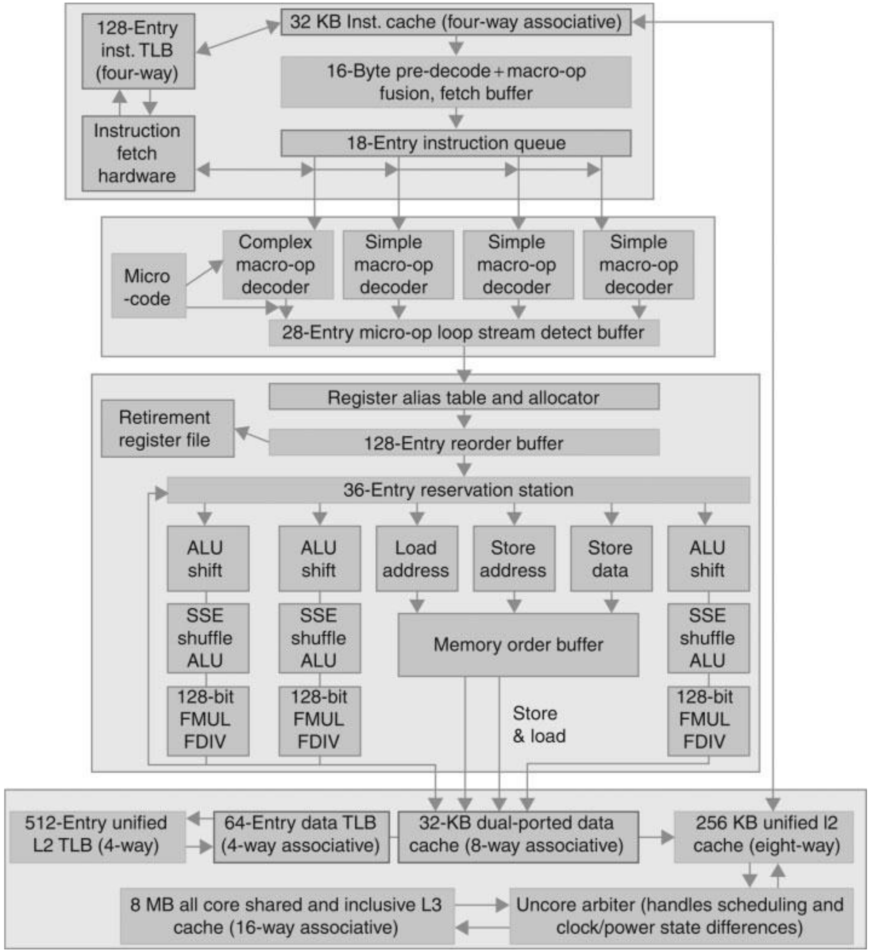


FIGURE 4.77 The Core i7 pipeline with memory components. The total pipeline depth is 14 stages, with branch mispredictions costing 17 clock cycles. This design can buffer 48 loads and 32 stores. The six independent units can begin execution of a ready RISC operation each clock cycle.

Intel Core i7 Performance

- Ideal CPI = 0.25 since this is a 4-wide processor

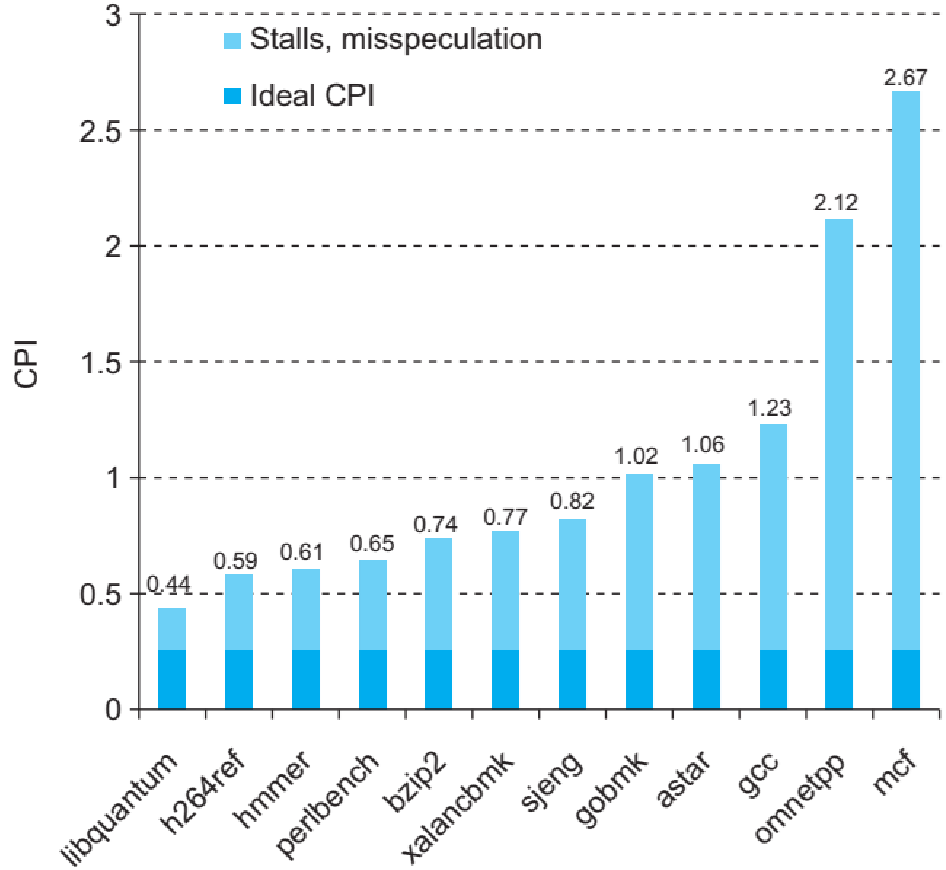


FIGURE 4.78 **CPI of Intel Core i7 920 running SPEC2006 integer benchmarks.**

University of Pittsburgh

Intel Core i7 Impact of Branch Misprediction

- Due to deep pipeline, tiny misprediction can have outsized impact

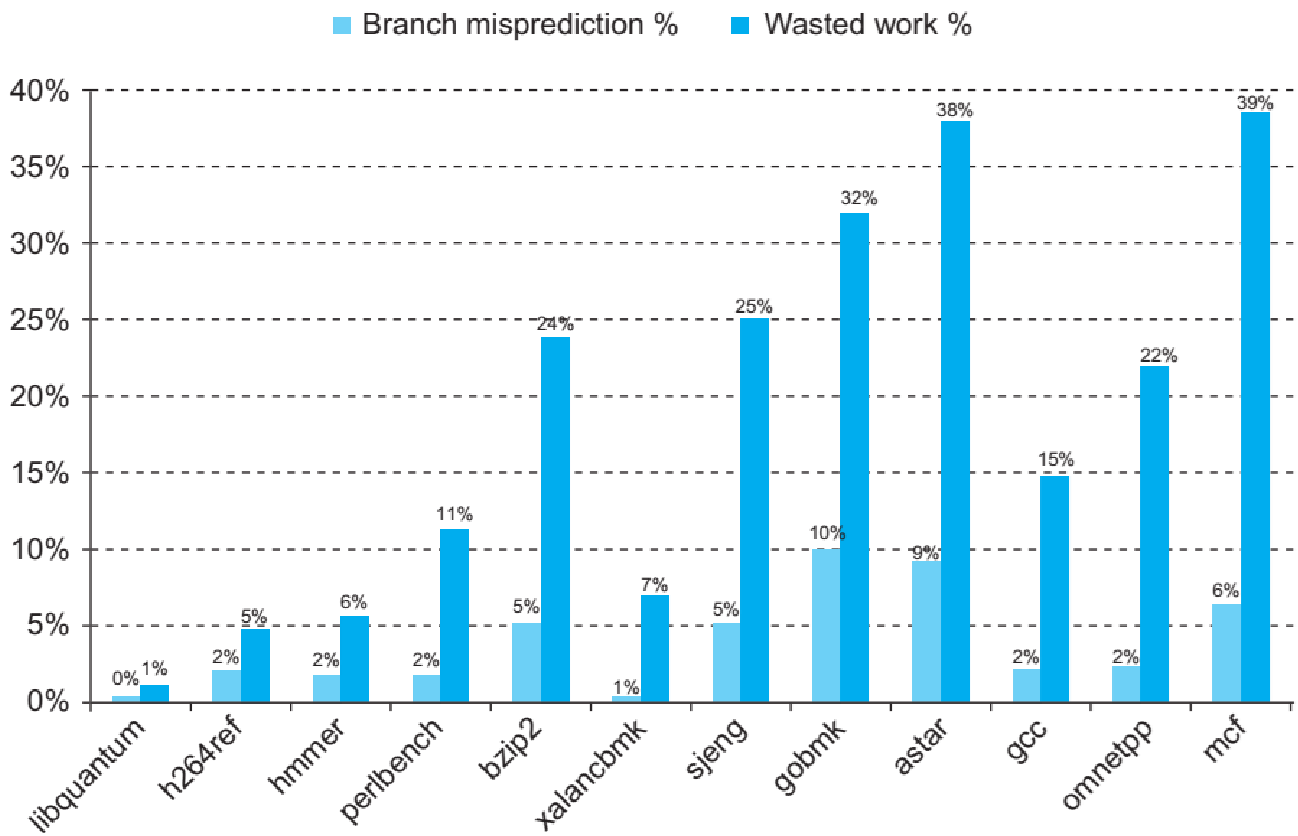


FIGURE 4.79 Percentage of branch mispredictions and wasted work due to unfruitful speculation of Intel Core i7 920 running SPEC2006 integer benchmarks.

University of Pittsburgh