SuperScalar Processors

CS 1541
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 adds to power and cycle time

- 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

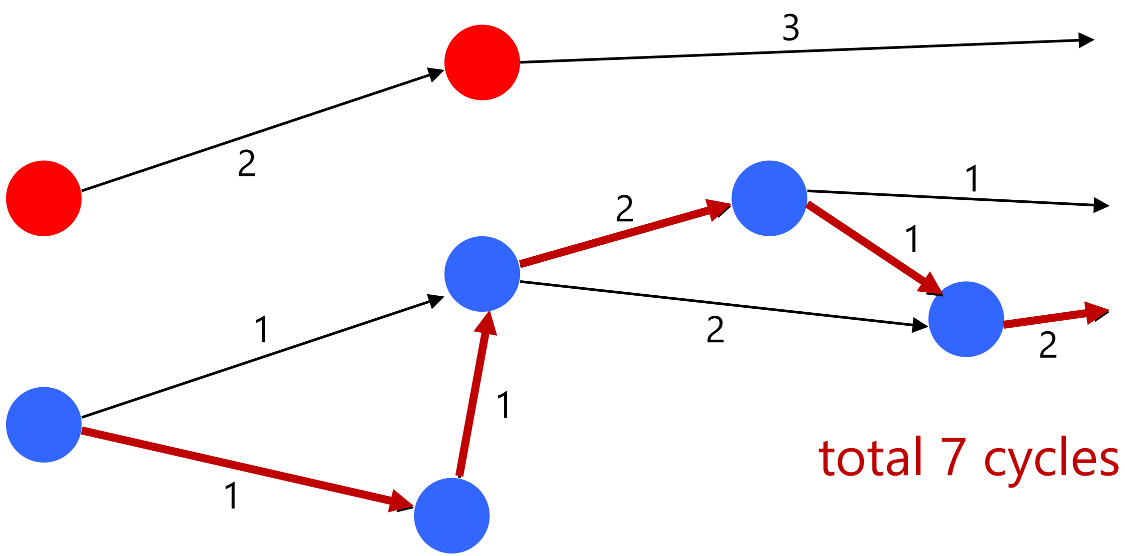
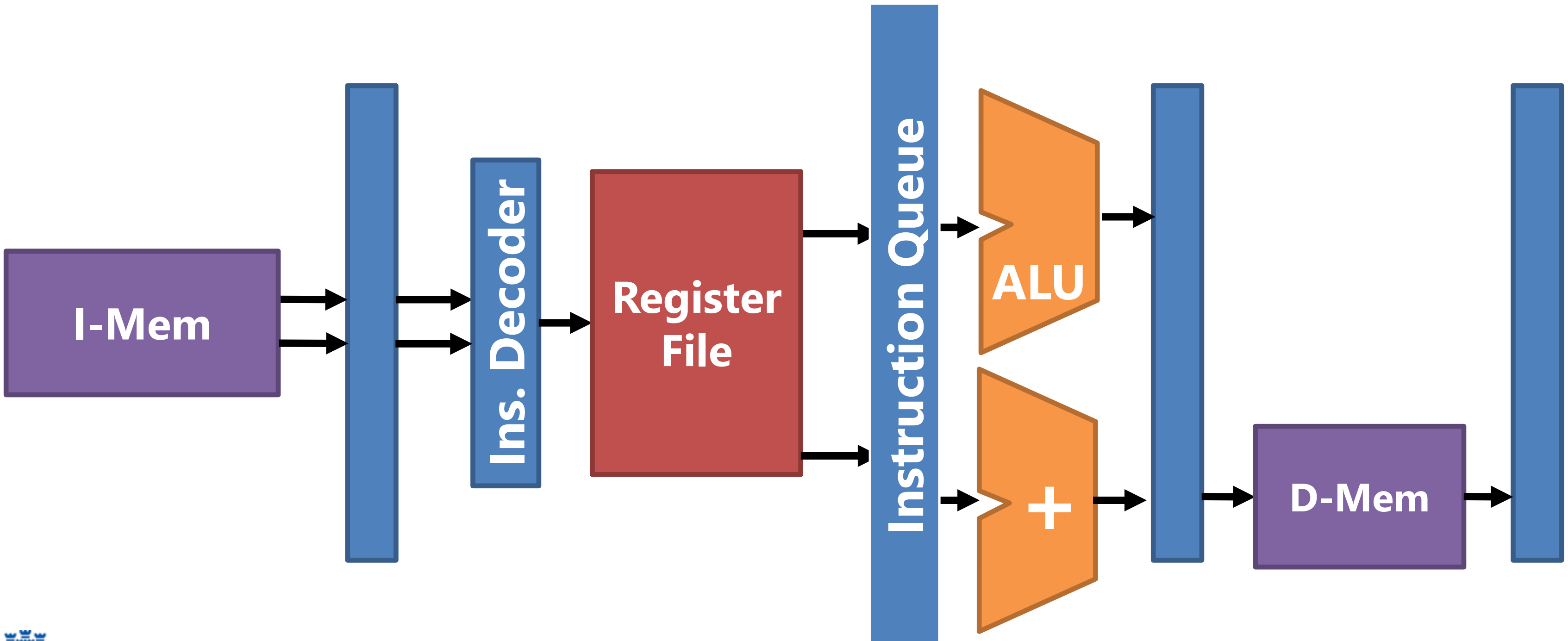


illustration courtesy of Dr. Melhem

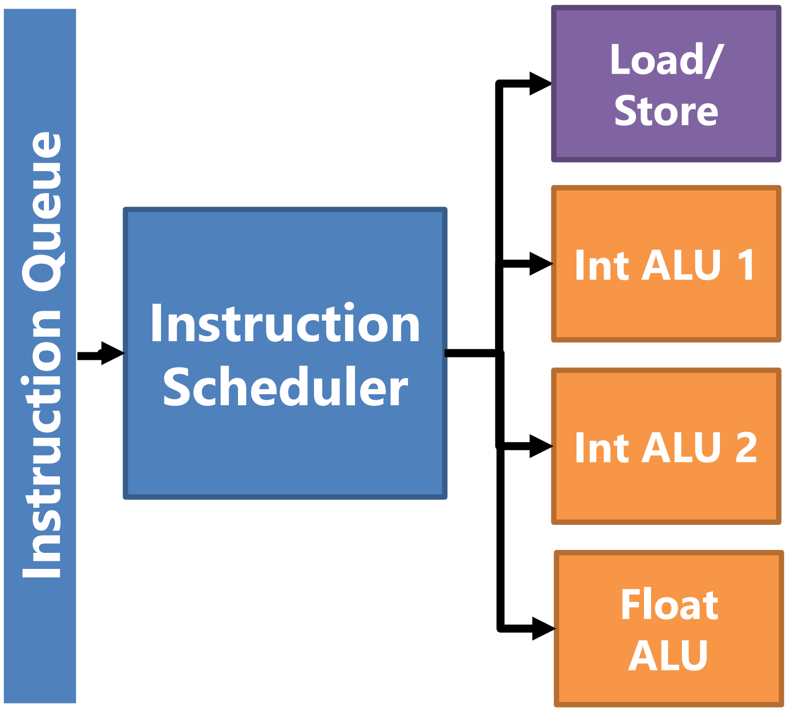
Instruction Queue

- 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***
  - Instructions **queue in-order** but can **execute out-of-order**



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***



- **ID**: decode insts**, dispatch** to i-queue
  - This happens **in-order** of the program
- **EX**: **issue** ready insts to an EX unit
  - Insts become ready when values of operands are available
  - This can happen **out-of-order**
- Queue grows / shrinks dynamically
  - Grows when data hazards stall issue
  - Shrinks when control hazards flush instructions from queue

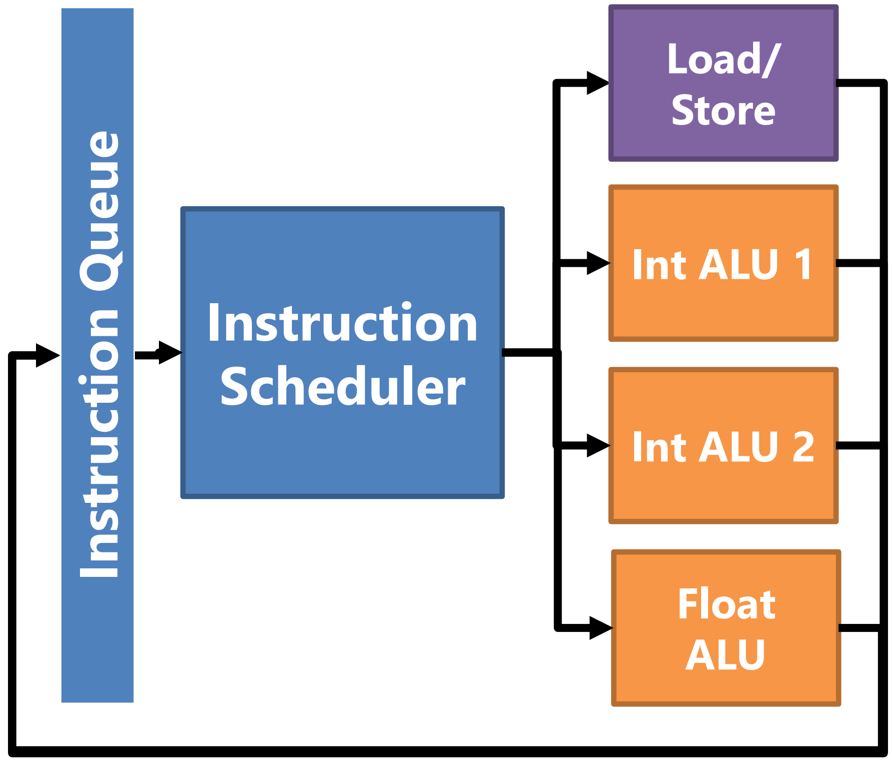
University of Pittsburgh

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
```



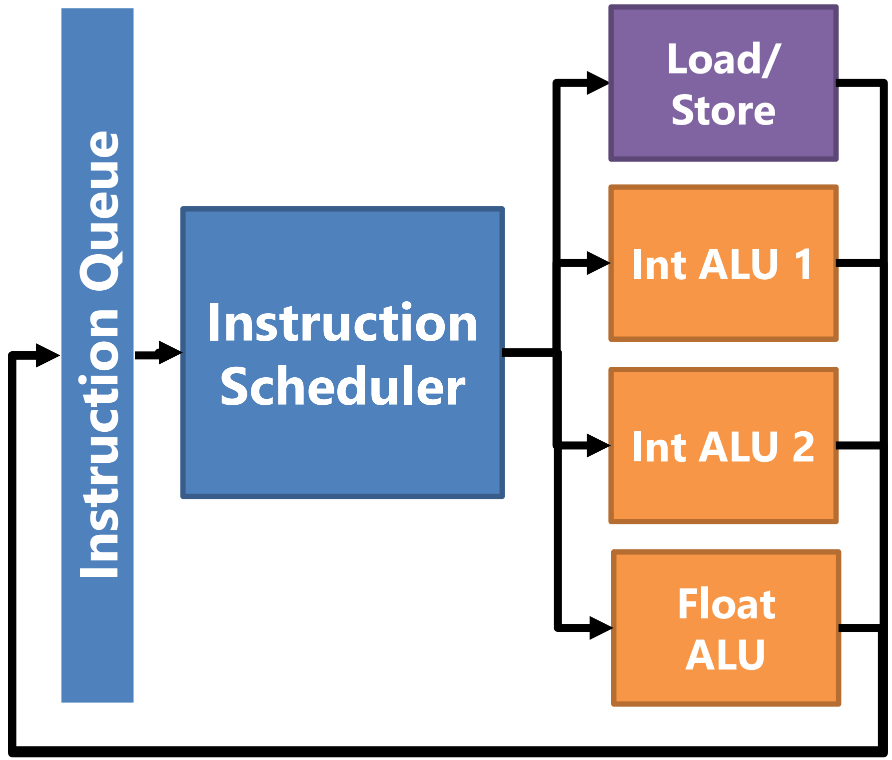
[Issued]

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
```



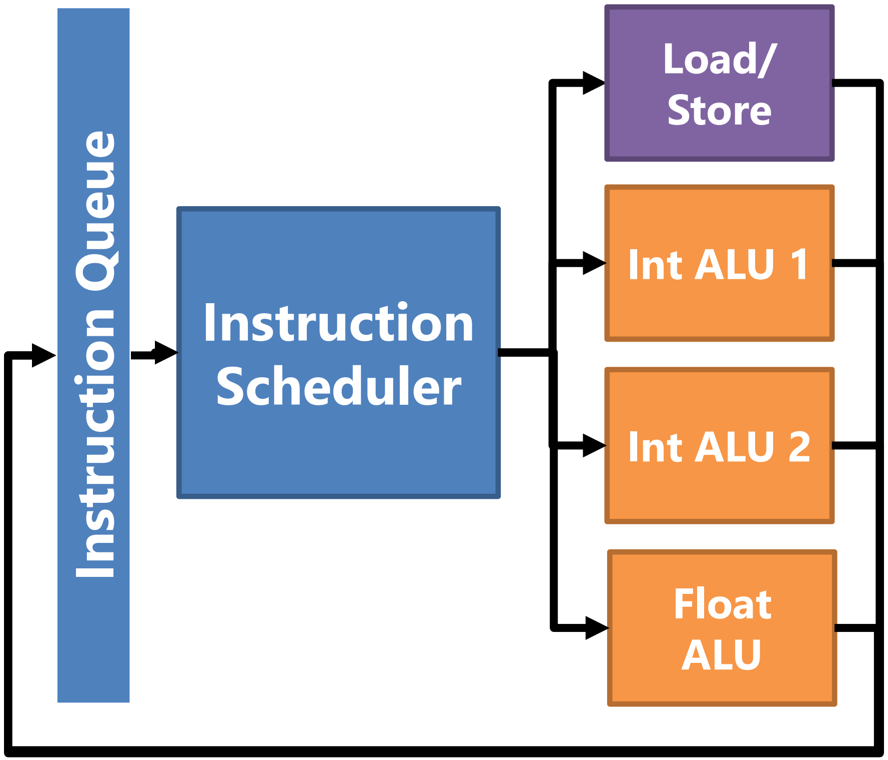
[Issued]

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
```

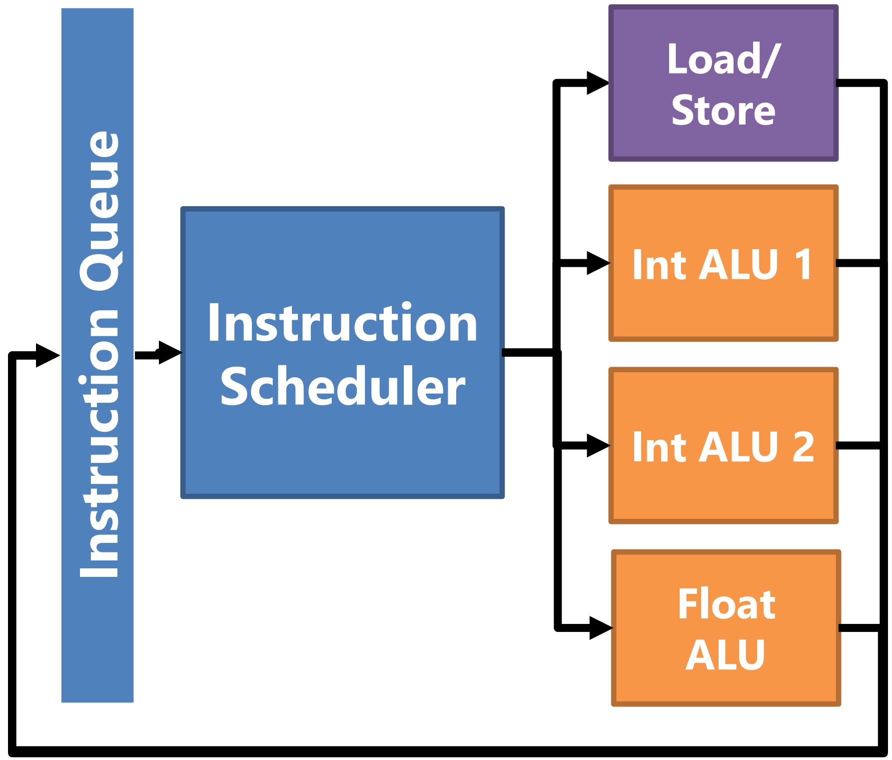
Scheduling the Instruction Queue

- 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

[Dispatched]

`li t0, 1`

`add t2, t0, t1`



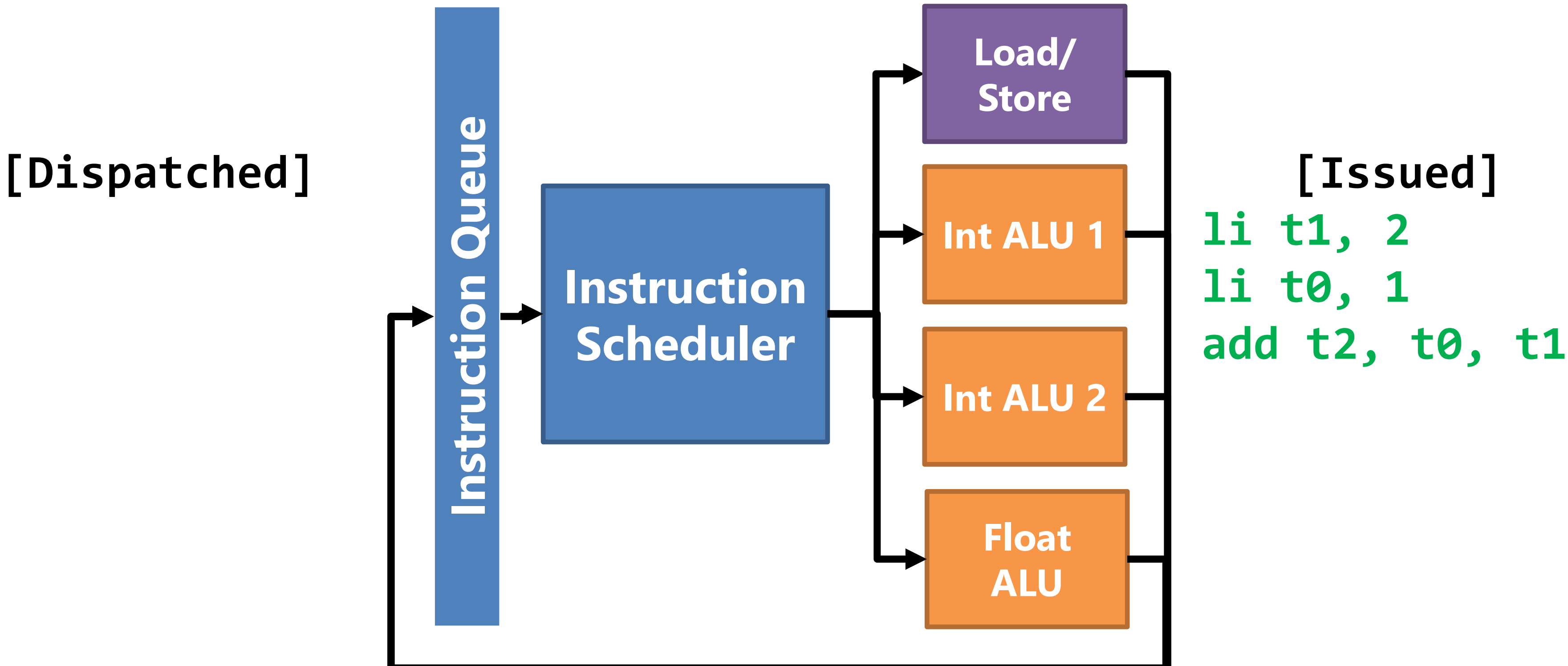
[Issued]

`li t1, 2`

`li t0, 1`

Scheduling the Instruction Queue

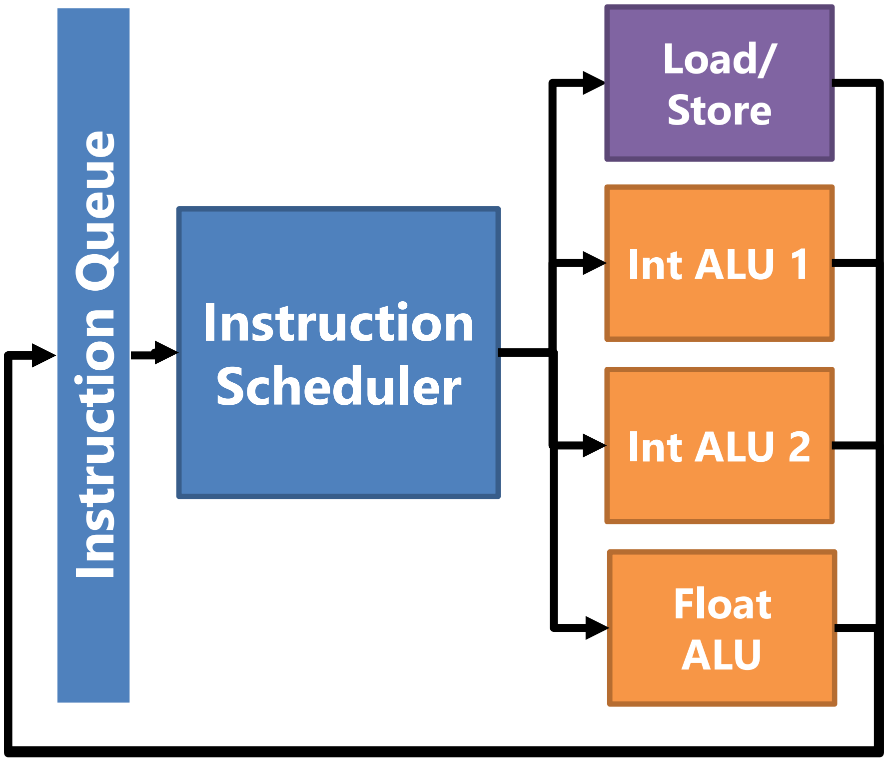
- And we are done!



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
```

University of Pittsburgh

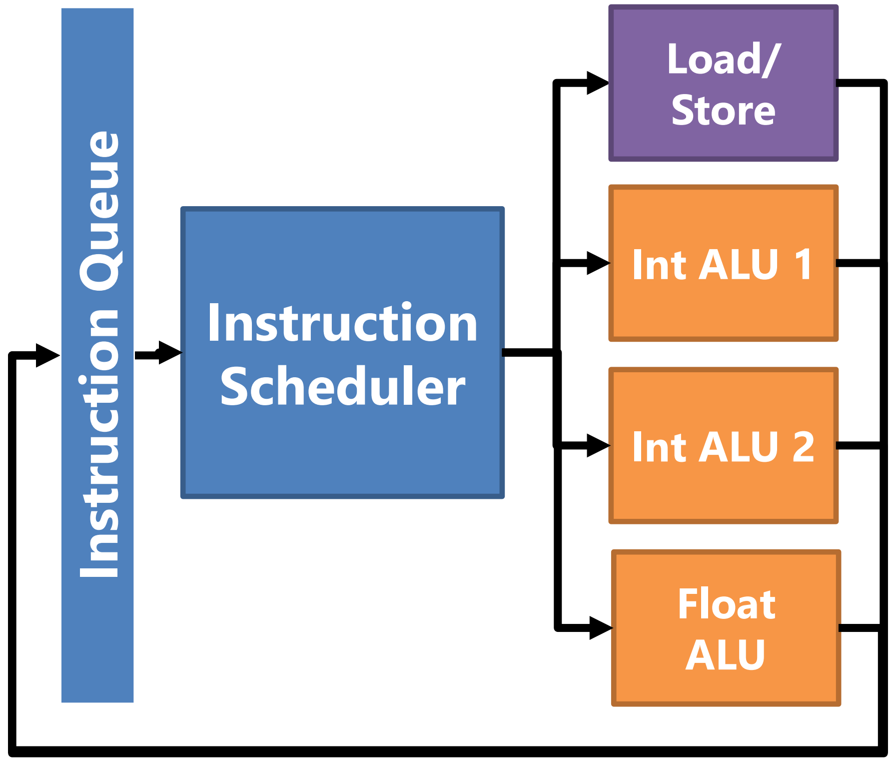
What if we had a WAW dependency?

- 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
```

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]

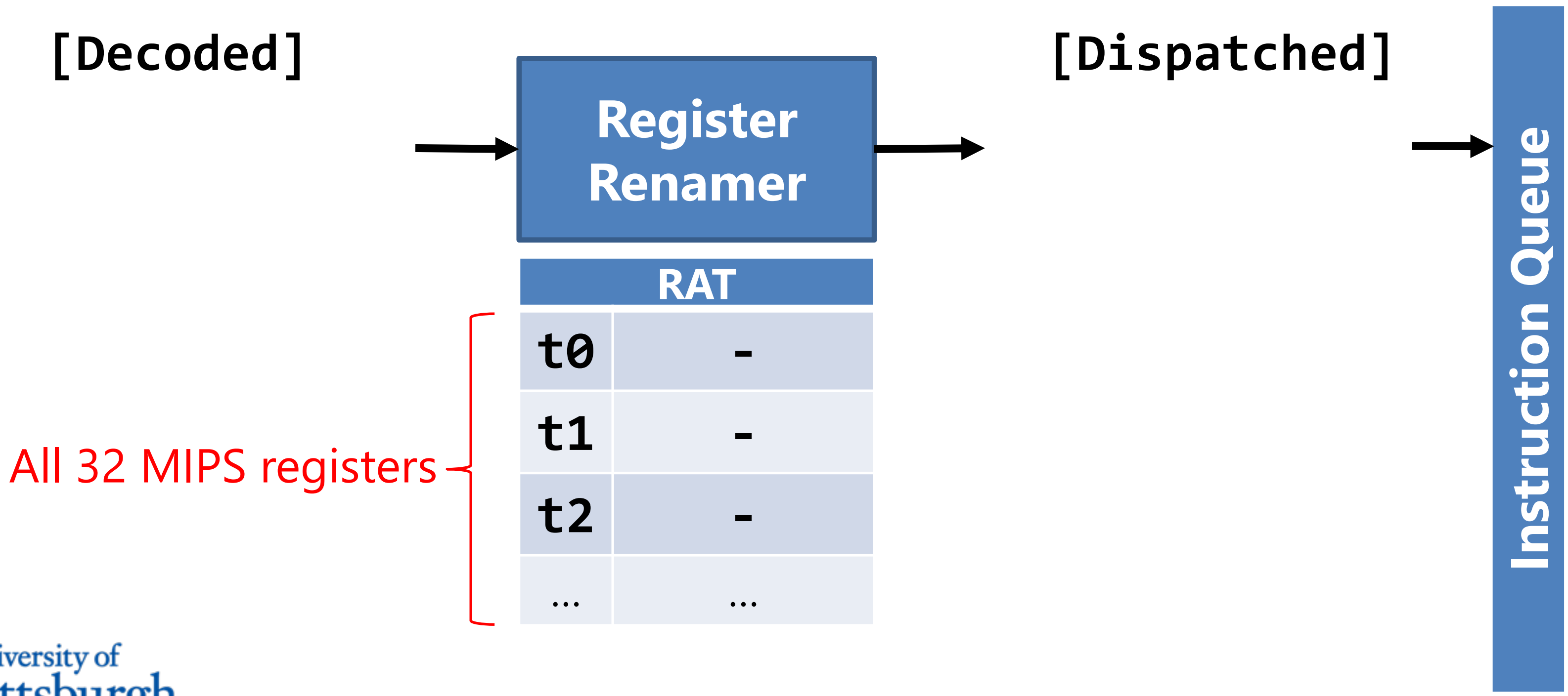
```
li p0, 1
li p1, 2
add p2, p1, p1
```

No WAW!

Instruction Queue

Register Renamer and the RAT

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



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`

Instruction Queue

All 32 MIPS registers

RAT	
t0	p0
t1	-
t2	-
…	…

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`**
   - Remember the single assignment rule?
   - A new value always gets a new register

[Decoded]

```
li t0, 1
li t0, 2
```

→ **Register Renamer** →

[Dispatched]

```
li p0, 1
li p1, 2
```

→ **Instruction Queue**

RAT

t0	p1
t1	-
t2	-
…	…

All 32 MIPS registers

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
```

→ **Instruction Queue**

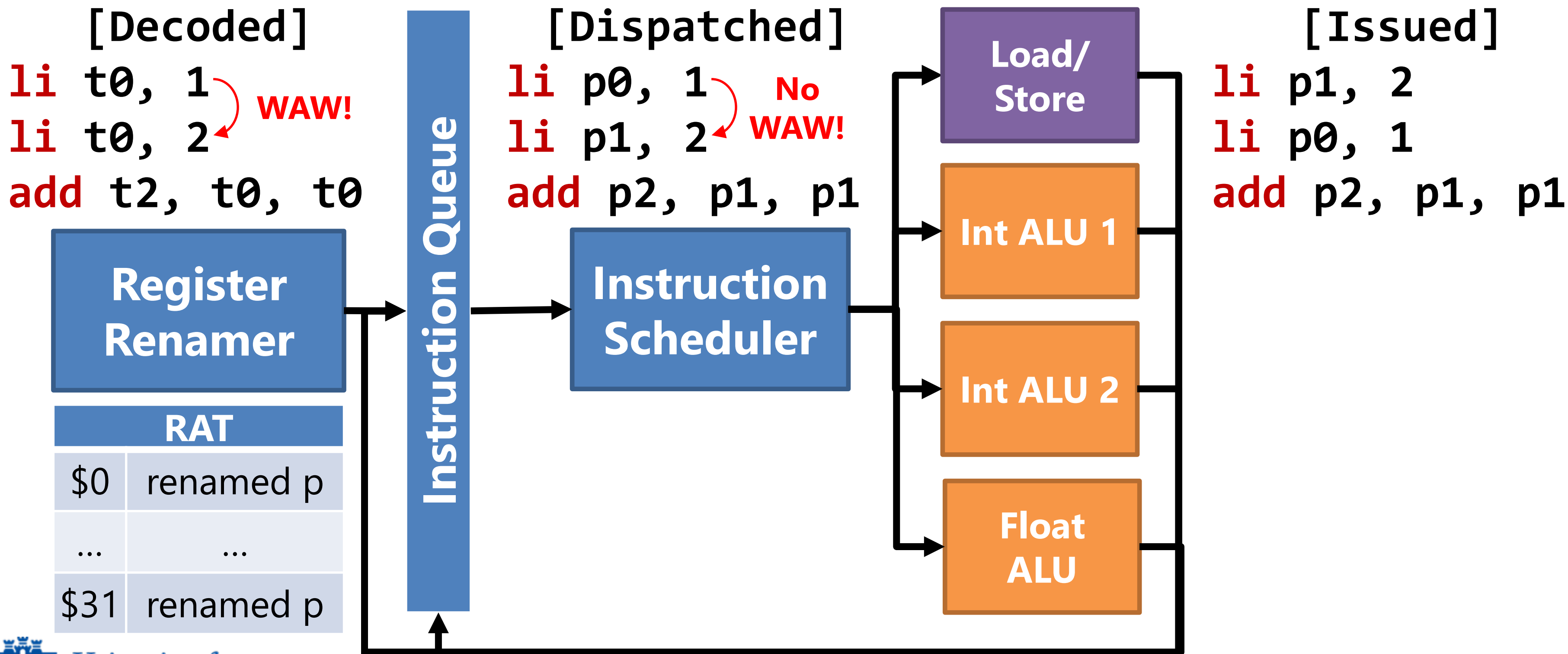
RAT

t0	p1
t1	-
t2	p2
…	…

All 32 MIPS registers

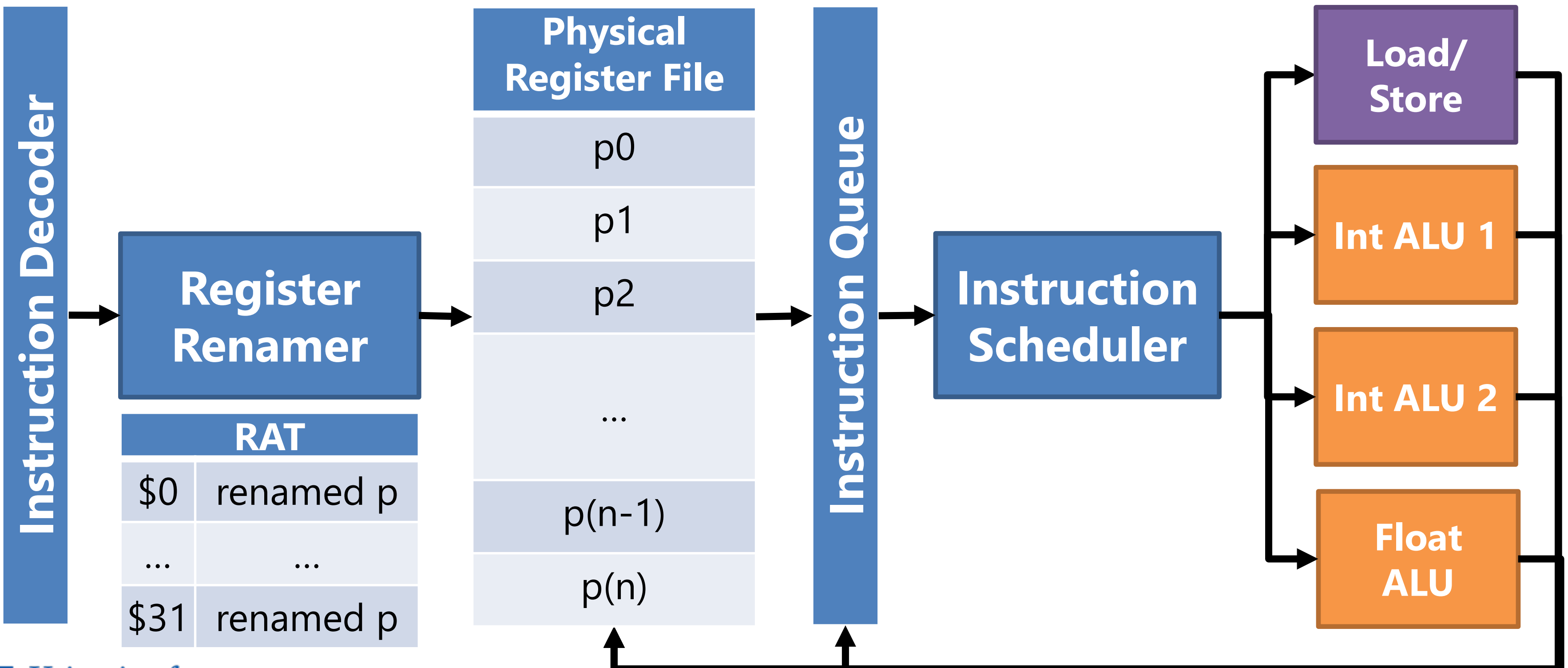
Register Rename Removes all WAW/WAR Deps

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



All Computation Done using Physical Registers

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



Benefit of increasing width of superscalar processors

- Can wider and wider processors give us limitless speedups?

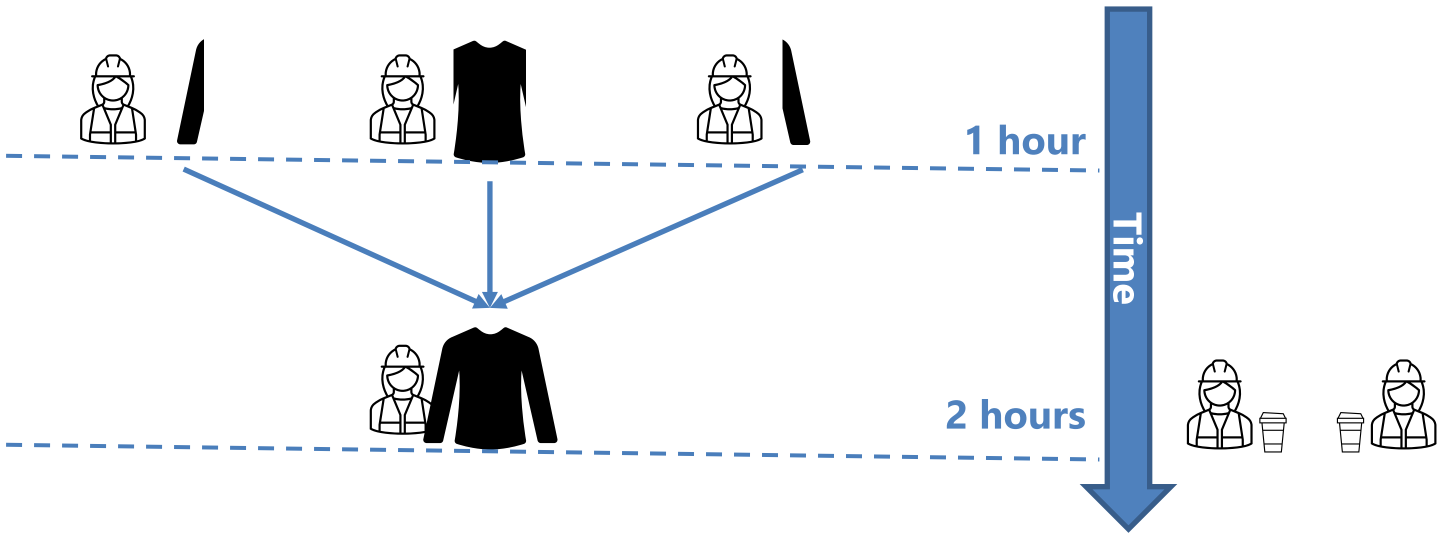
- There is a **fundamental limit** to achievable IPC
  - Amount of **ILP (Instruction Level Parallelism)** in code
  - This applies to both superscalar and VLIW processors

- What limits ILP?
  - True **RAW** dependencies (WAR and WAW can be removed)
  - Control dependencies?
    - Branches can be predicted so not a fundamental limit

- ILP is a **property of the program**, not the processor
  - How much ILP is there in programs?

Remember this slide from the introduction?

- 3 workers = fastest but low utilization:



- During the 2nd hour, 2 workers were slacking over coffee.
- IPC = 4 instructions / 2 cycles = 2, when processor width = 3 (utilization = 66%)
- Utilization depends on the ILP present in the data dependence graph.

ILP present in different programs

- 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

Cost of increasing width of superscalar processors

- Achieving the theoretical limit would be awesome
  - In reality, superscalars are typically no more than 10-wide

- Increasing width of superscalar processors has costs:
  - To leverage ILP, a large instruction window is needed, meaning:
    - Large instruction queue size
    - Large physical register file size
  - Upsizing above structures negatively impacts **cycle time**
    - Time to search and schedule instruction queue
    - Time to access register file (increased size and number of ports)
  - It also negatively impacts **power consumption**

University of Pittsburgh

Load / Store Queue

How about data dependencies through memory?

- RAW, WAR, and WAW dependencies happen through memory as well
  - Clearly, the below code has no data dependencies through registers

```
sw   $t0, 4($s0)    // stores to 0xdeadbeef
lw   $t1, 8($s1)    // loads from 0xdeadbeef
```

  - But there is a RAW dependency through the location 0xdeadbeef

- Question: how does processor enforce RAW dependencies?
- Question: how does processor deal with WAR and WAW dependencies?

- Answer: through **memory renaming** using a **load / store queue**
  - Just like registers, a **new queue entry** created for every **store** instruction
    → All **WAR** and **WAW** memory dependencies are **removed**
  - Loads fetch data from most recent queue entry with same address
    → All **RAW** memory dependencies are **enforced**

Every store gets a new store queue entry

- Loads / stores are inserted into load / store queue as well instruction queue
  - Age denotes age of memory operation (incremented at every mem op)
  - Address and value of mem op is unknown until mem op is complete

[Decoded]

```
sw $t0, 0($s0)
sw $t1, 0($s1)
lw $t2, 0($s2)
```

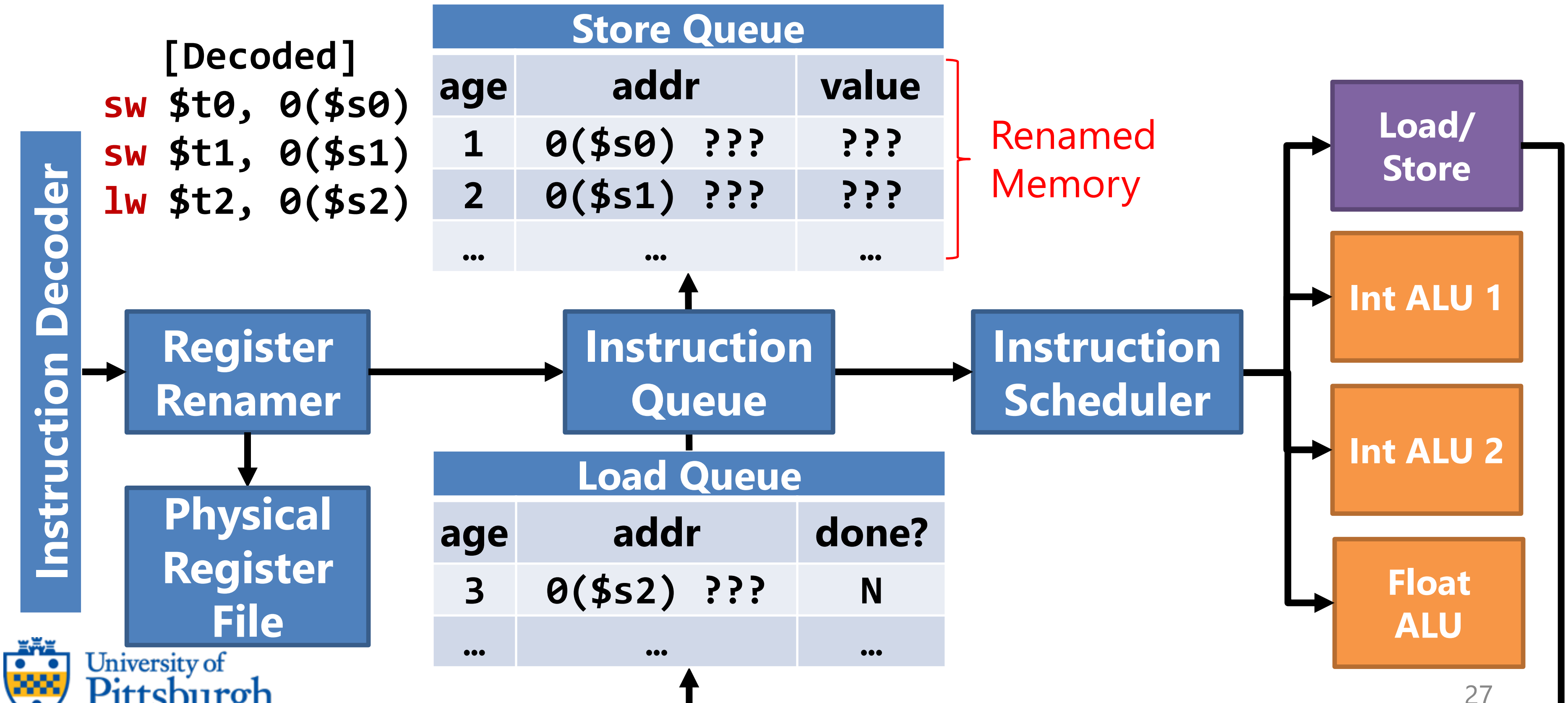
Store Queue

age	addr	value
1	0($s0) ???	???
2	0($s1) ???	???
…	…	…

Renamed Memory

Load Queue

age	addr	done?
3	0($s2) ???	N
…	…	…



Scenario 1: WAW reordering of two stores

- Let's say **sw** `$t1, 0($s1)` becomes ready first in the i-queue and executes
  - `0($s1)` contains value `0xdeadbeef` and `$t1` contains value **1**

[Decoded]
```
sw $t0, 0($s0)
sw $t1, 0($s1)
lw $t2, 0($s2)
```

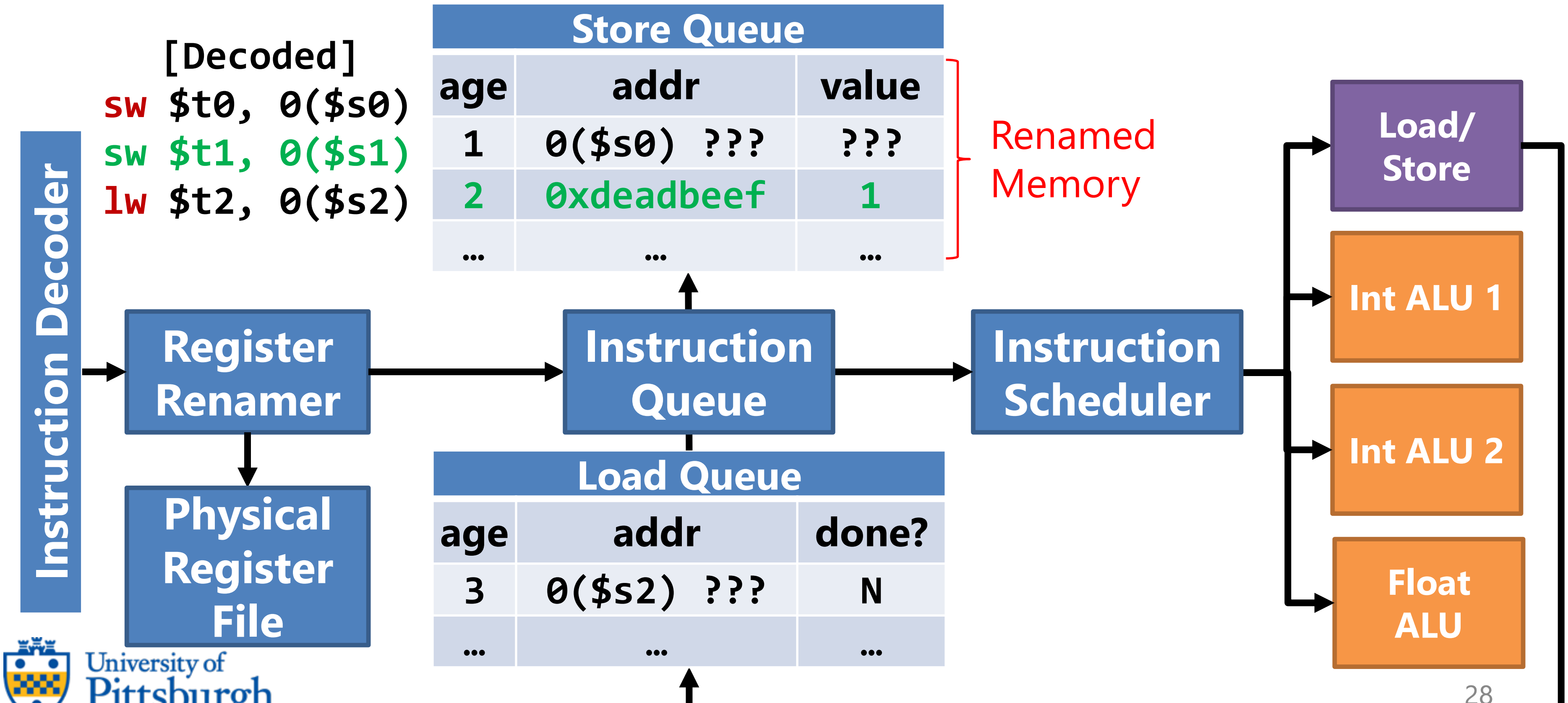
Store Queue

age	addr	value
1	0($s0) ???	???
2	0xdeadbeef	1
…	…	…

Renamed Memory

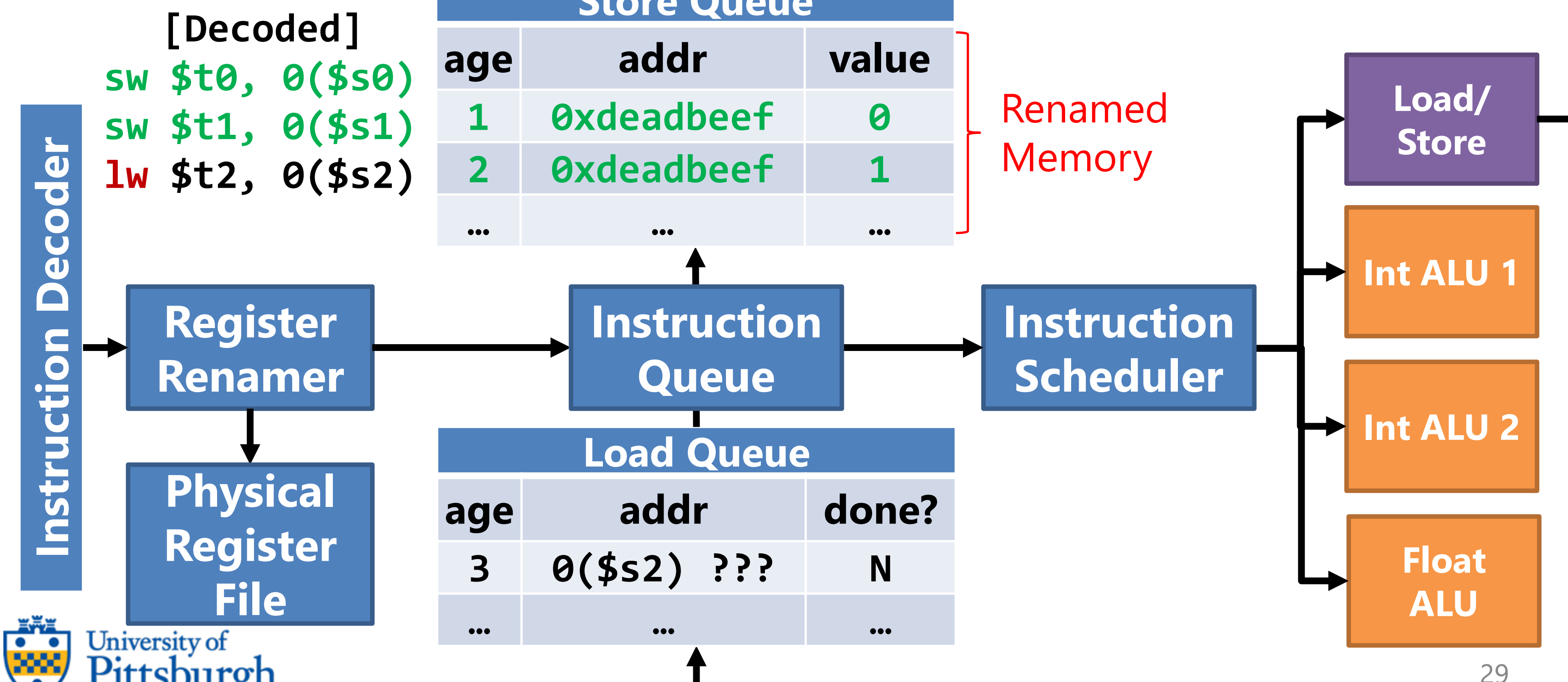
Load Queue

age	addr	done?
3	0($s2) ???	N
…	…	…



Scenario 1: WAW reordering of two stores

- Next, `sw $t0, 0($s0)` becomes ready in the i-queue and executes
  - `0($s1)` is also contains **`0xdeadbeef`** and **`$t0`** contains **0**
  - So, we have effectively reordered execution of a WAW dependency



Store Queue

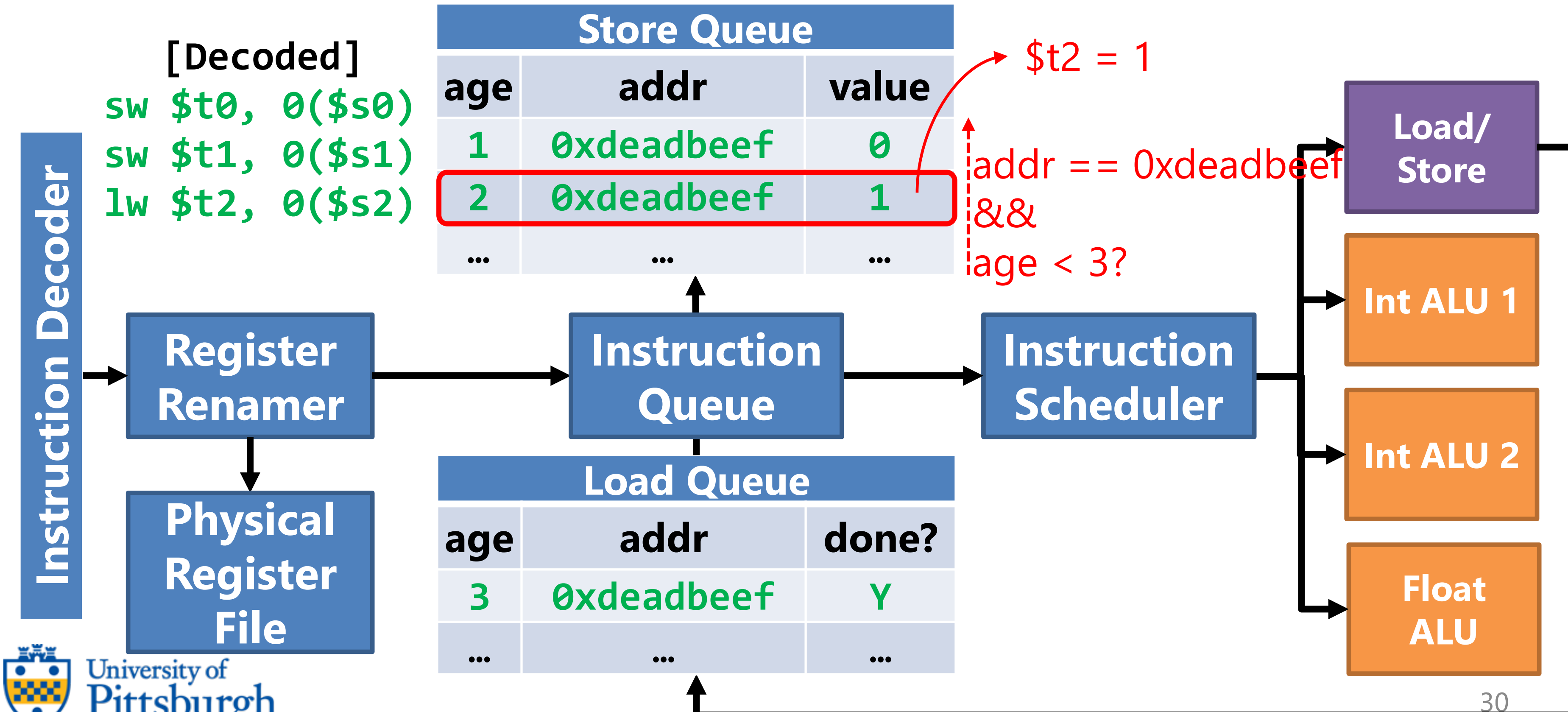
age	addr	value
1	0xdeadbeef	0
2	0xdeadbeef	1
…	…	…

Load Queue

age	addr	done?
3	0($s2) ???	N
…	…	…

Scenario 1: WAW reordering of two stores

- Finally, `lw $t2, 0($s2)` becomes ready in the i-queue and executes
  - `0($s2)` also resolves to **`0xdeadbeef`** meaning a RAW dependence
  - Load Unit searches Store Queue for most recent store matching address



University of Pittsburgh

Scenario 2: Flush due to RAW violation

- In this scenario, `lw $t2, 0($s2)` becomes ready first and executes

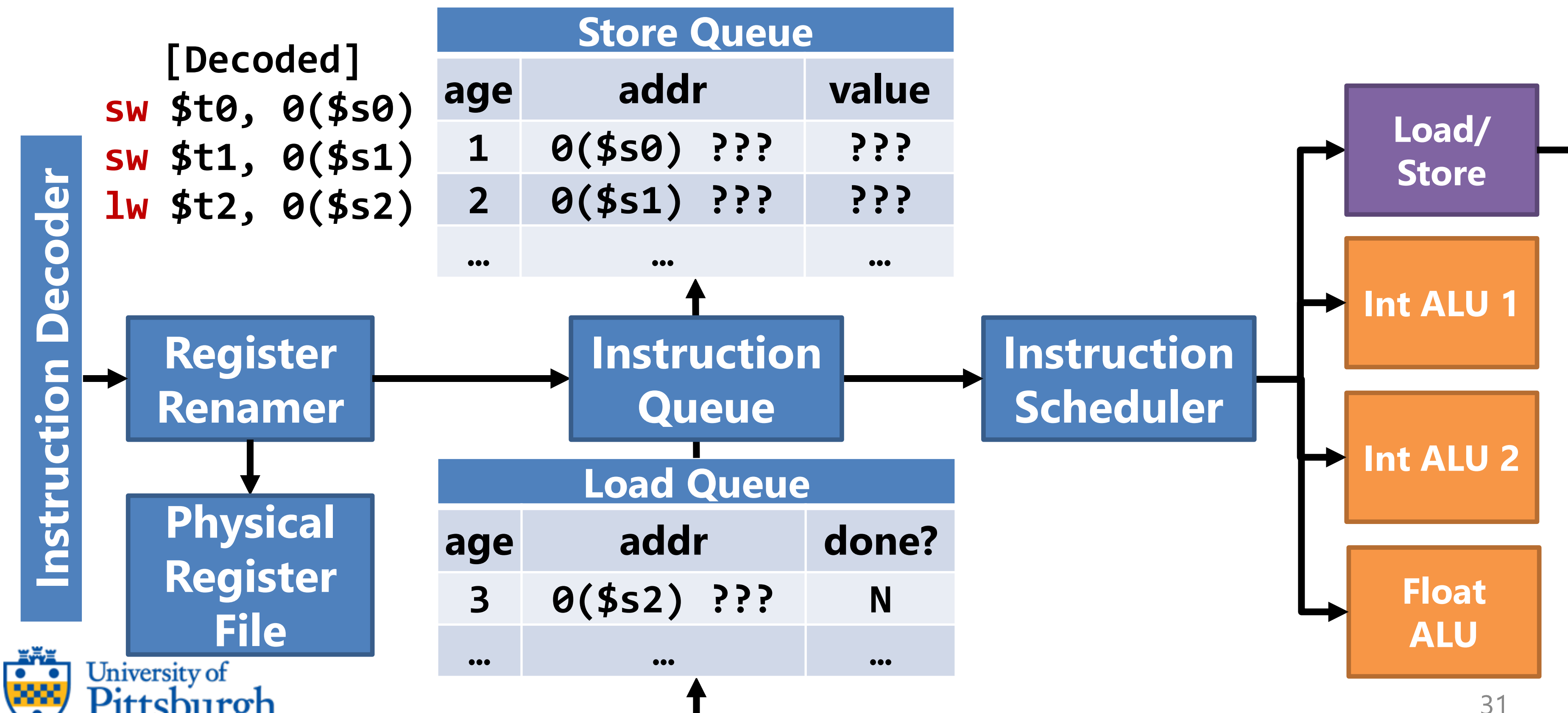
```
[Decoded]
sw $t0, 0($s0)
sw $t1, 0($s1)
lw $t2, 0($s2)
```

Store Queue

age	addr	value
1	0($s0) ???	???
2	0($s1) ???	???
…	…	…

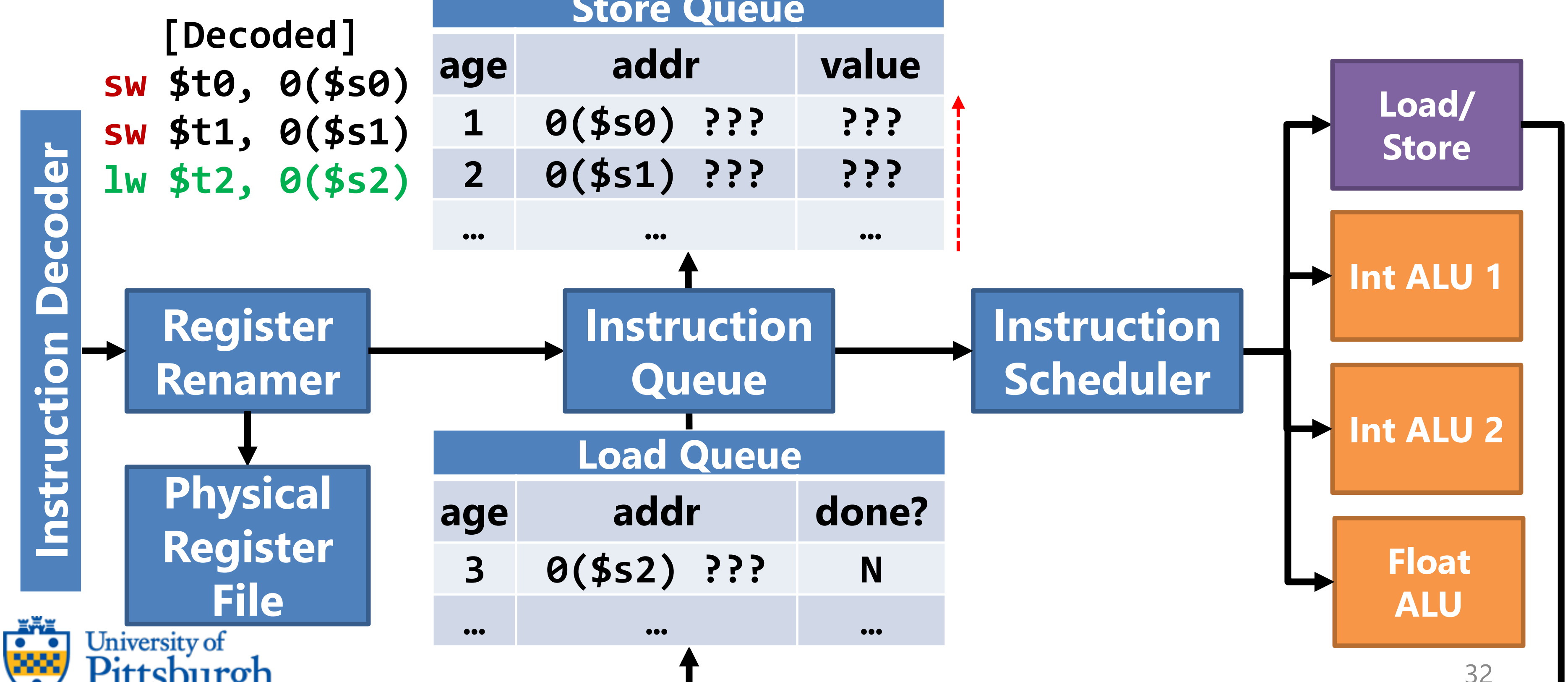
Load Queue

age	addr	done?
3	0($s2) ???	N
…	…	…



Scenario 2: Flush due to RAW violation

- In this scenario, `lw $t2, 0($s2)` becomes ready first and executes
  - Load Unit searches store queue but does not find matching entry



Store Queue

age	addr	value
1	0($s0) ???	???
2	0($s1) ???	???
…	…	…

Load Queue

age	addr	done?
3	0($s2) ???	N
…	…	…

Scenario 2: Flush due to RAW violation

- In this scenario, **`lw $t2, 0($s2)`** becomes ready first and executes
  - Load Unit searches store queue but does not find matching entry
  - So, it simply fetches value for **`$t2`** from memory

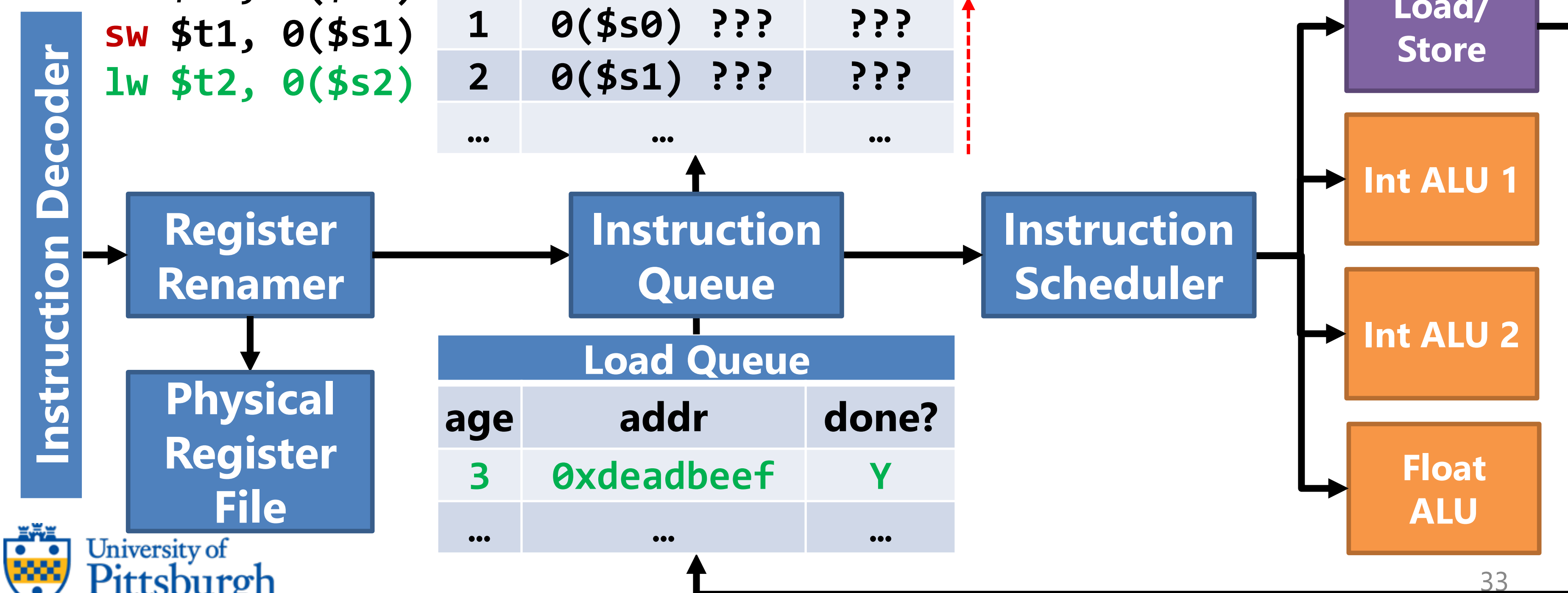
```
[Decoded]
sw $t0, 0($s0)
sw $t1, 0($s1)
lw $t2, 0($s2)
```

Store Queue

age	addr	value
1	0($s0) ???	???
2	0($s1) ???	???
…	…	…

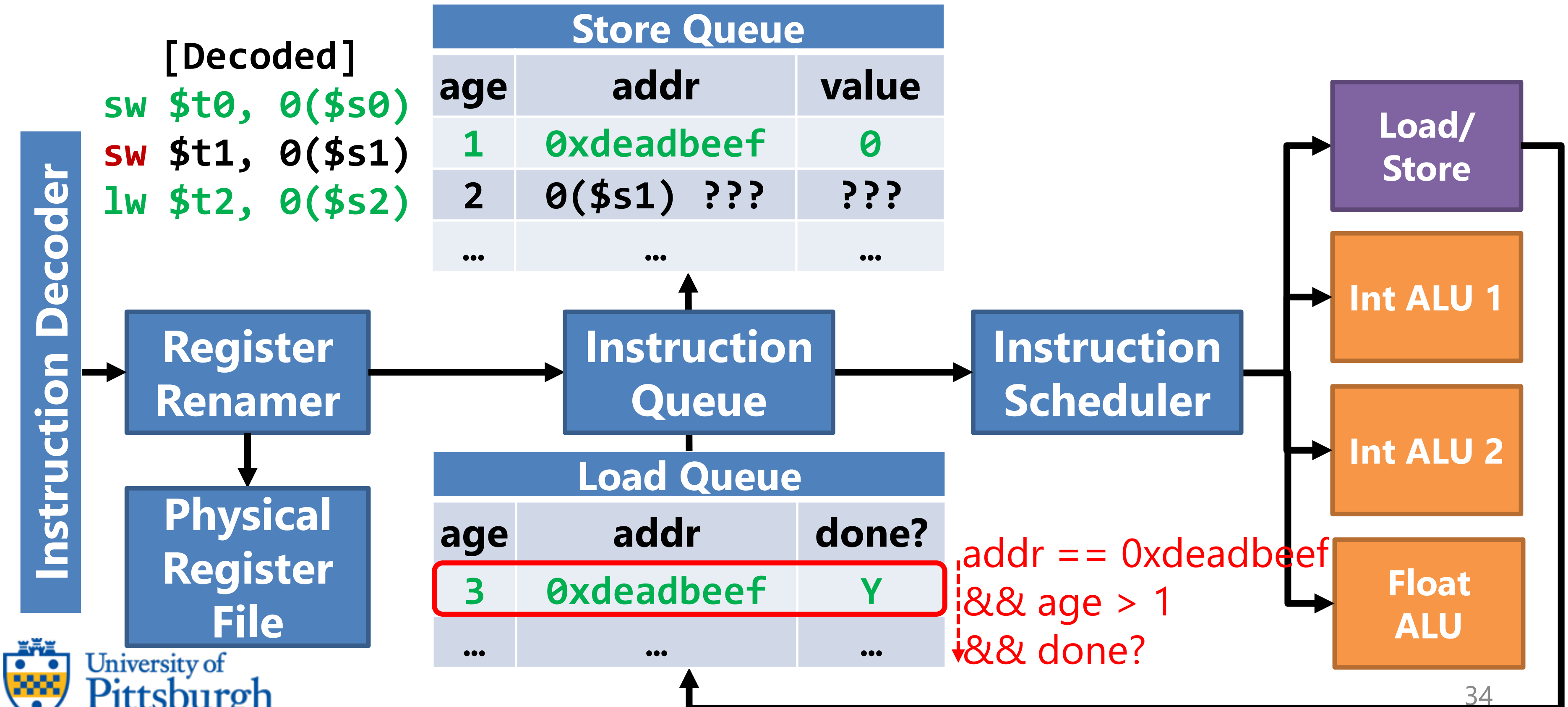
Load Queue

age	addr	done?
3	0xdeadbeef	Y
…	…	…



Scenario 2: Flush due to RAW violation

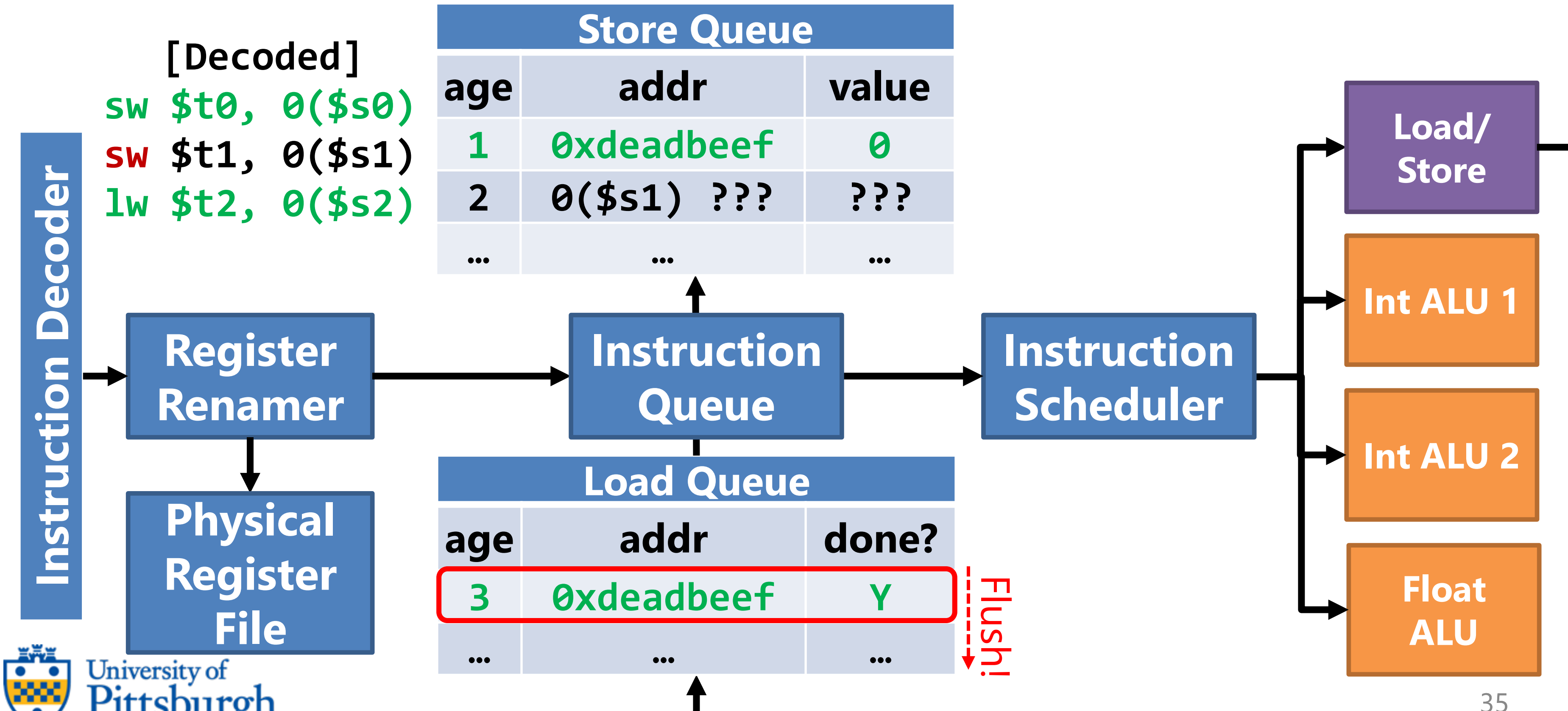
- Next, `sw $t0, 0($s0)` becomes ready in the i-queue and executes
  - Store Unit searches Load Queue to see if there were RAW violations
  - And, yes, there is a Load that performed earlier than it should have!



University of Pittsburgh

Scenario 2: Flush due to RAW violation

- Flush **lw $t2, 0($s2)** and all instructions that follow it in i-queue
  - All following execution has been polluted by incorrect value of **$t2**



Precise 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 OS **exception handler** is invoked

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 OS exception handler.
  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**

Rules for Precise Exceptions

1. **All instructions before** the exception must have executed

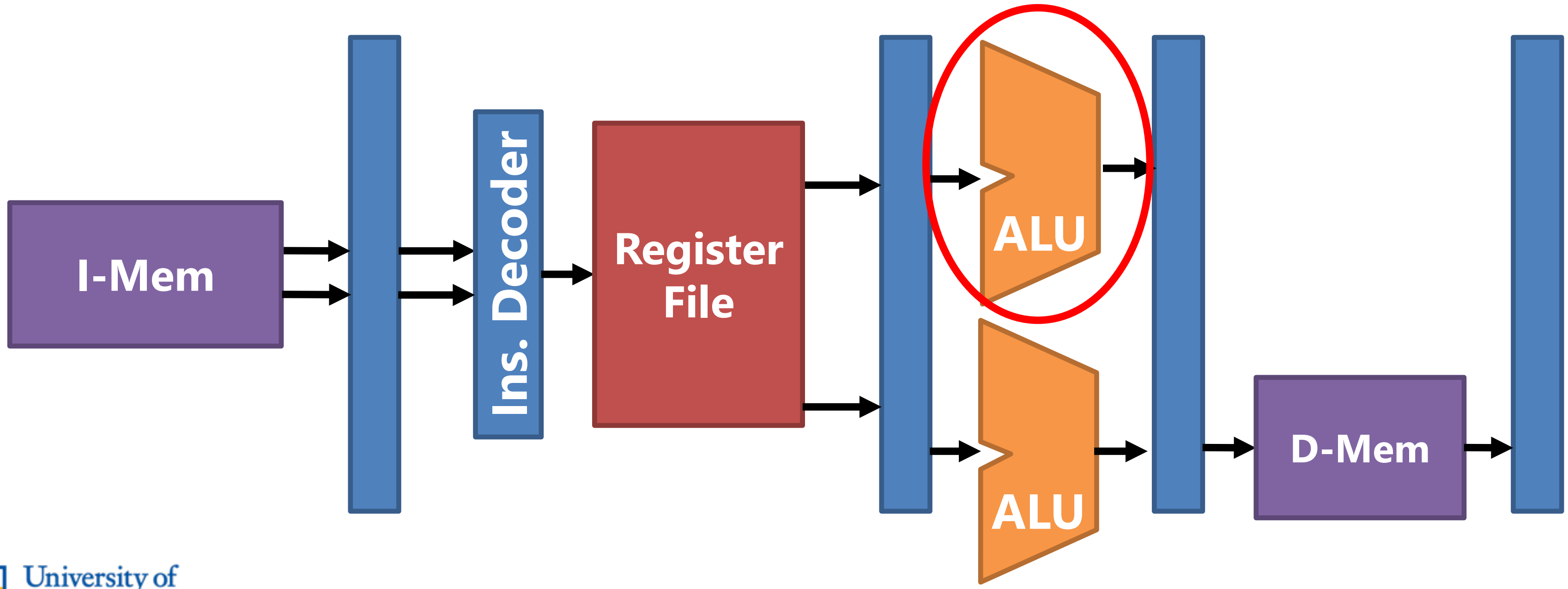
2. **No instructions after** the exception must execute

- **Architectural state**: the state visible to the ISA (i.e. software)
  - State in architectural registers (For MIPS: t0, t1, t2, ...)
  - State in data memory

- Architectural state at point of exception must reflect above rules

Precise Exceptions in In-order Processors is Easy

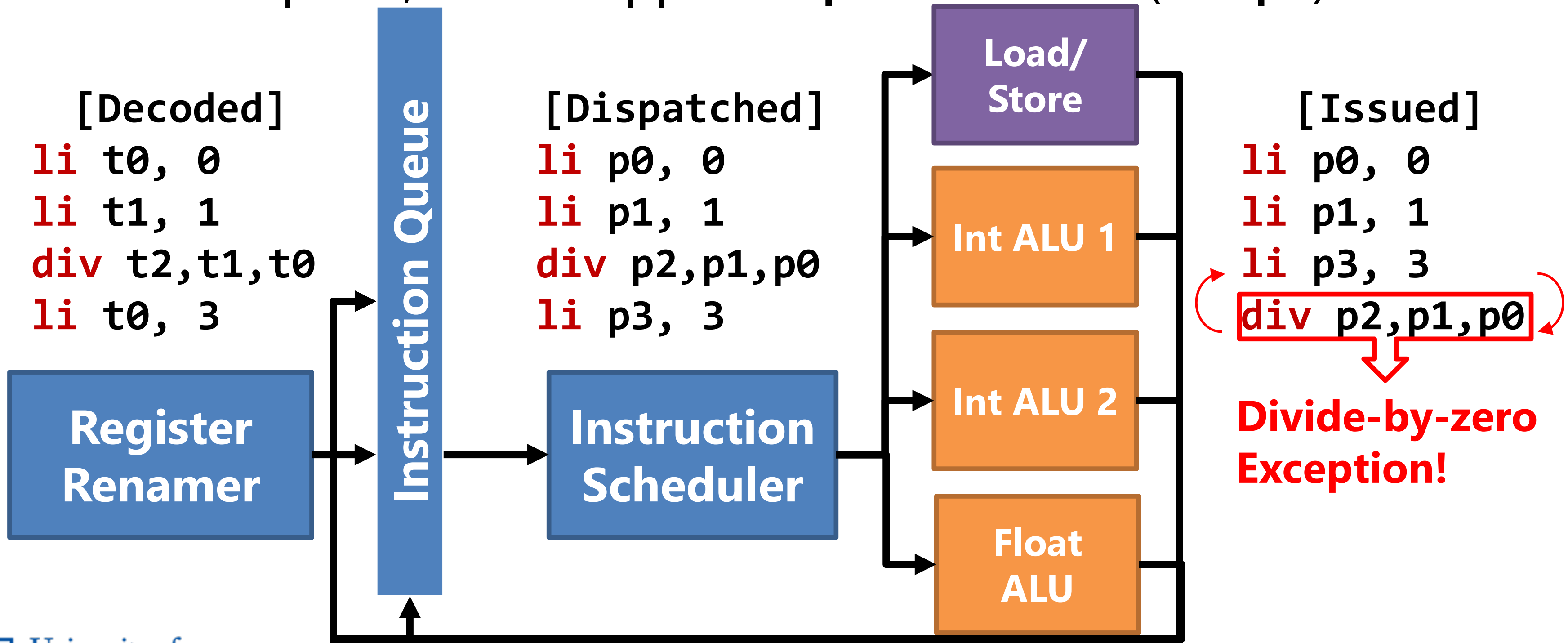
- Exceptions are typically detected at the EX stage
  - Stage where all arithmetic happens as well as address calculations
- On exception, flush EX and all previous stages (ID and IF)
  - Since in-order, guarantees instructions following EX do not writeback
  - Only state leading up to instruction at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



University of Pittsburgh

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`)



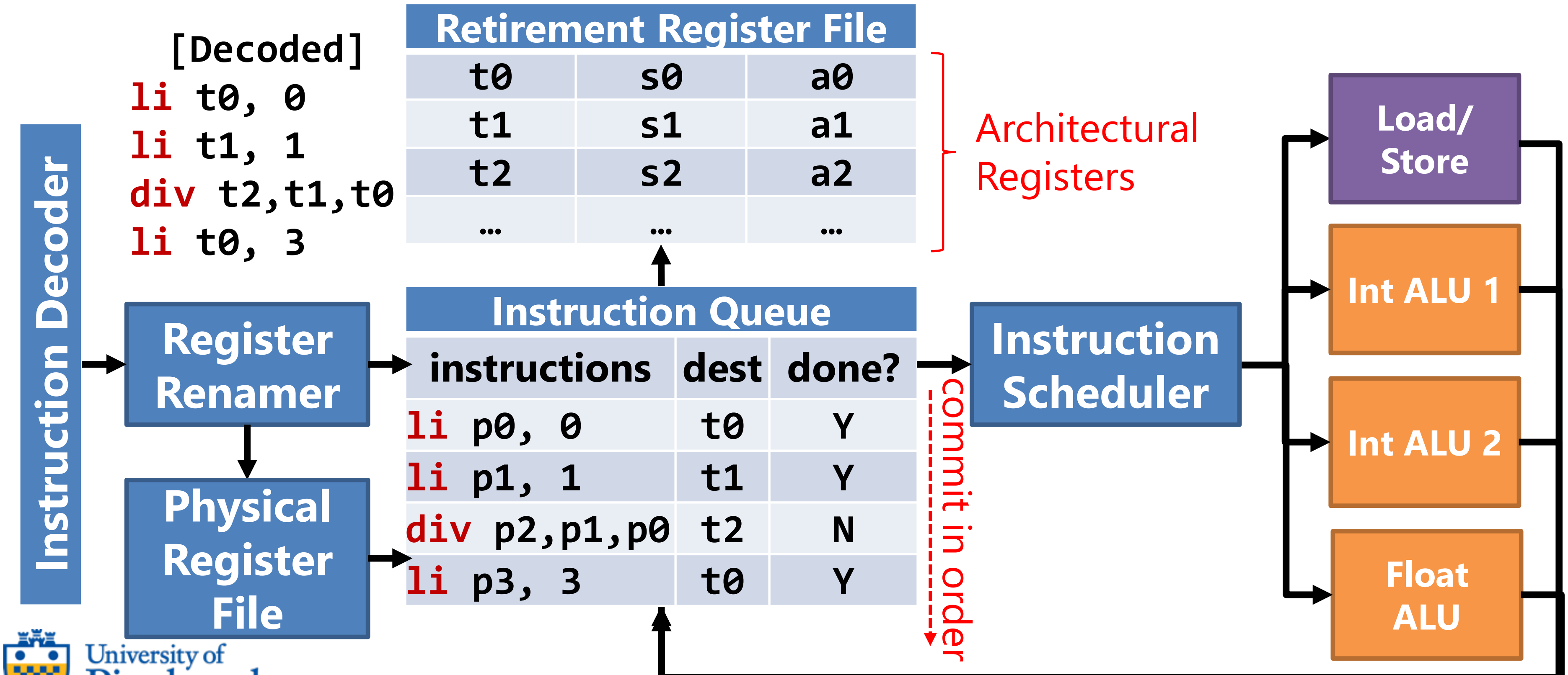
Precise Exceptions in Out-of-order Processors is Hard

- 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**
  - 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**
    - Physical registers are copied to architectural registers
    - Store queue entries are written to memory

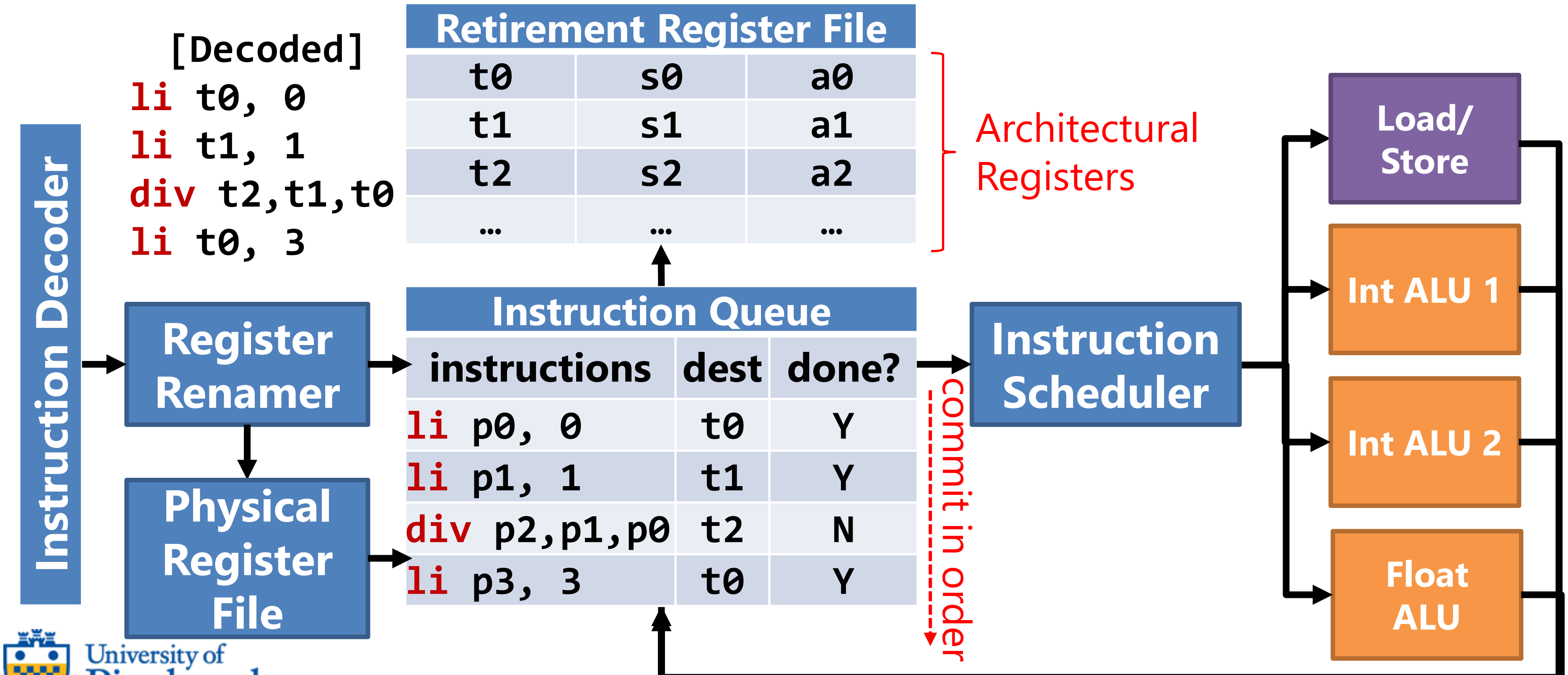
In-order Commit

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



In-order Commit Example: Cycle 1

- At this point, all **li** instructions have completed but not the **div**
- `li p0, 0` and `li p1, 1` can commit on the next cycle
  - But not `li p3, 3` since we have in-order commit!




```
[Decoded]
li t0, 0
li t1, 1
div t2,t1,t0
li t0, 3
```

Retirement Register File

t0	s0	a0
t1	s1	a1
t2	s2	a2
…	…	…

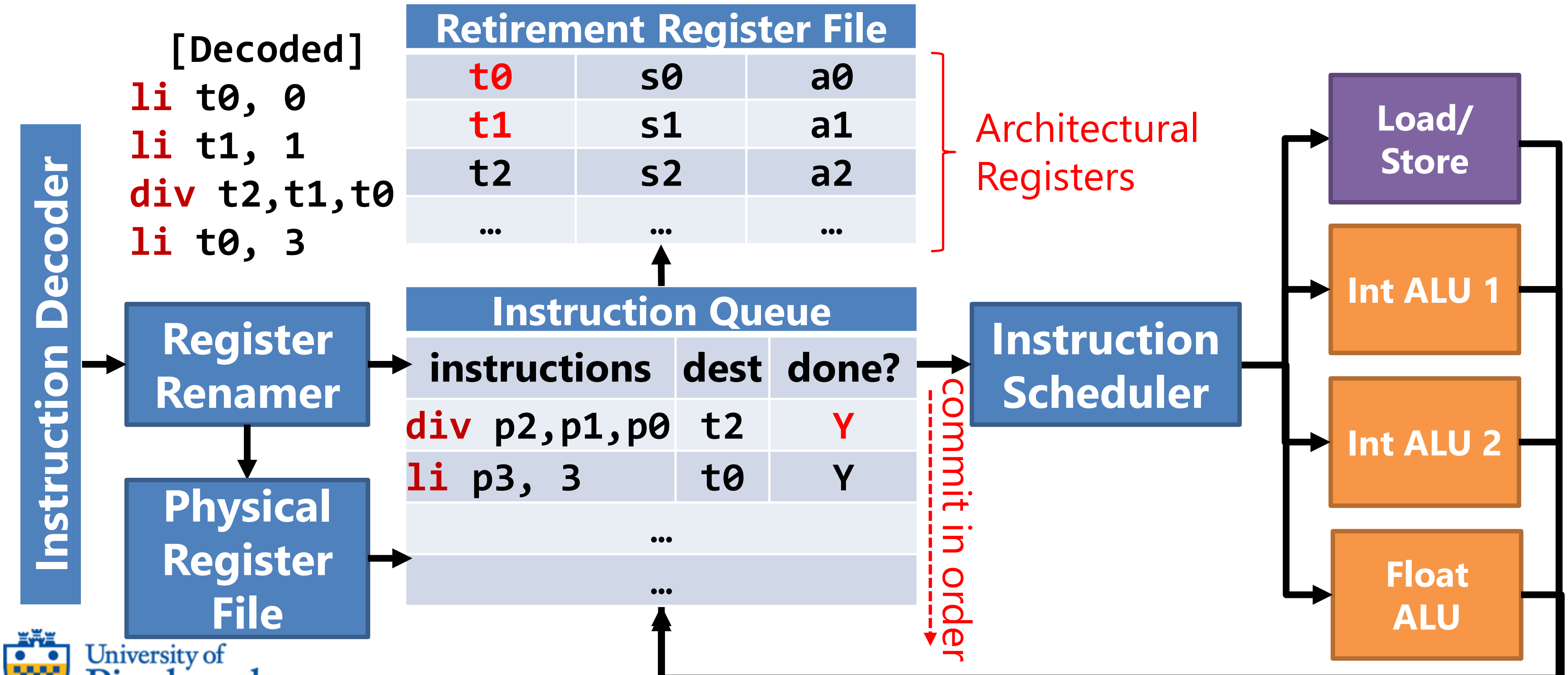
Instruction Queue

instructions	dest	done?
li p0, 0	t0	Y
li p1, 1	t1	Y
div p2,p1,p0	t2	N
li p3, 3	t0	Y

University of Pittsburgh

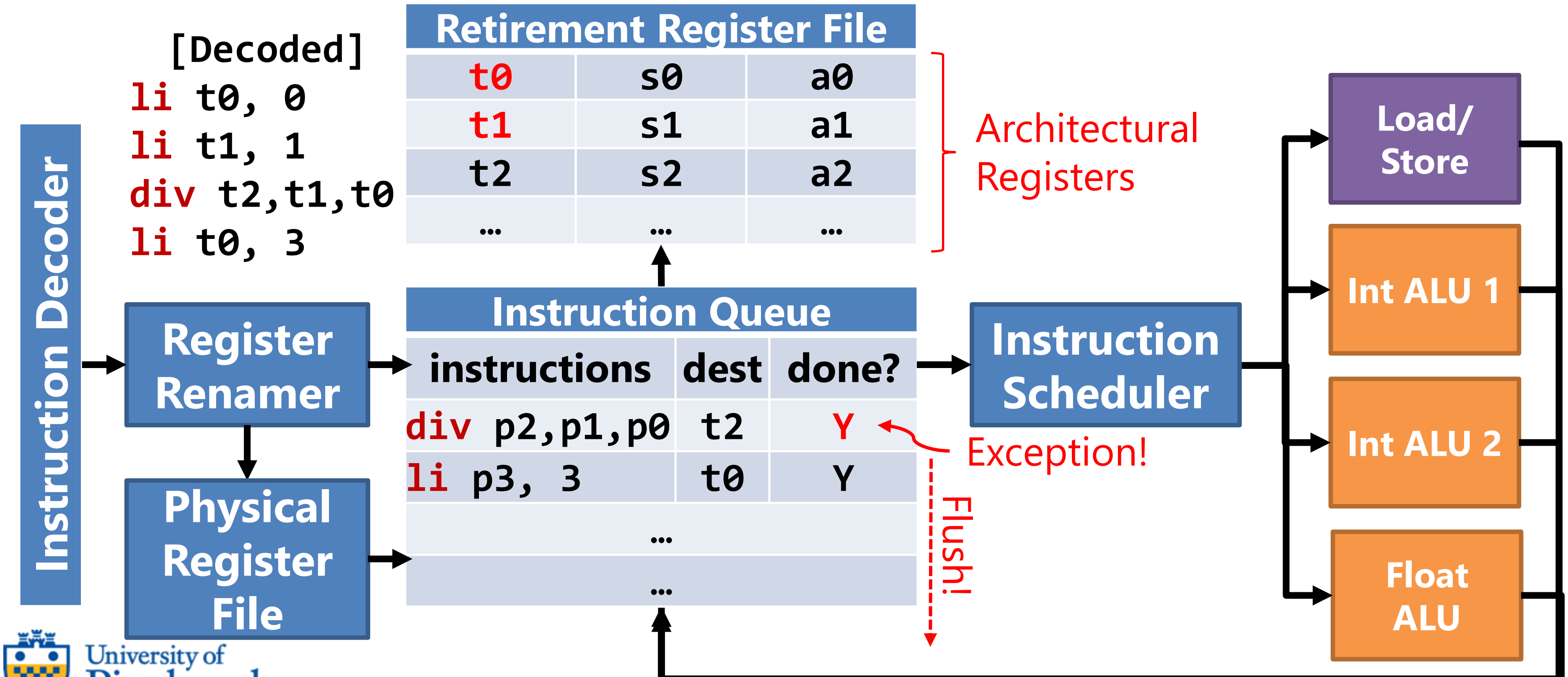
In-order Commit Example : Cycle 2

- `li p0, 0` and `li p1, 1` have committed updating `t0` and `t1`
- `div p2,p1,p0` has completed execution and is finally ready to commit
  - On completion, `div` has set an "exception bit" in i-queue (not shown here)



In-order Commit Example : Cycle 3

- An exception is raised for **`div`** **`p2,p1,p0`** when it tries to commit
  - Instructions following **`div`** are flushed, without modifying RRF
- **Retirement Register File** contains a **precise** architectural state



In-order Commit of Stores

- Values in Store Queue are committed **in-order**
  - Second store can't commit until address for `sw $t0, 0($s0)` is resolved
  - **Data memory** contains a **precise** architectural state

```
[Decoded]
sw $t0, 0($s0)
sw $t1, 0($s1)
lw $t2, 0($s2)
```

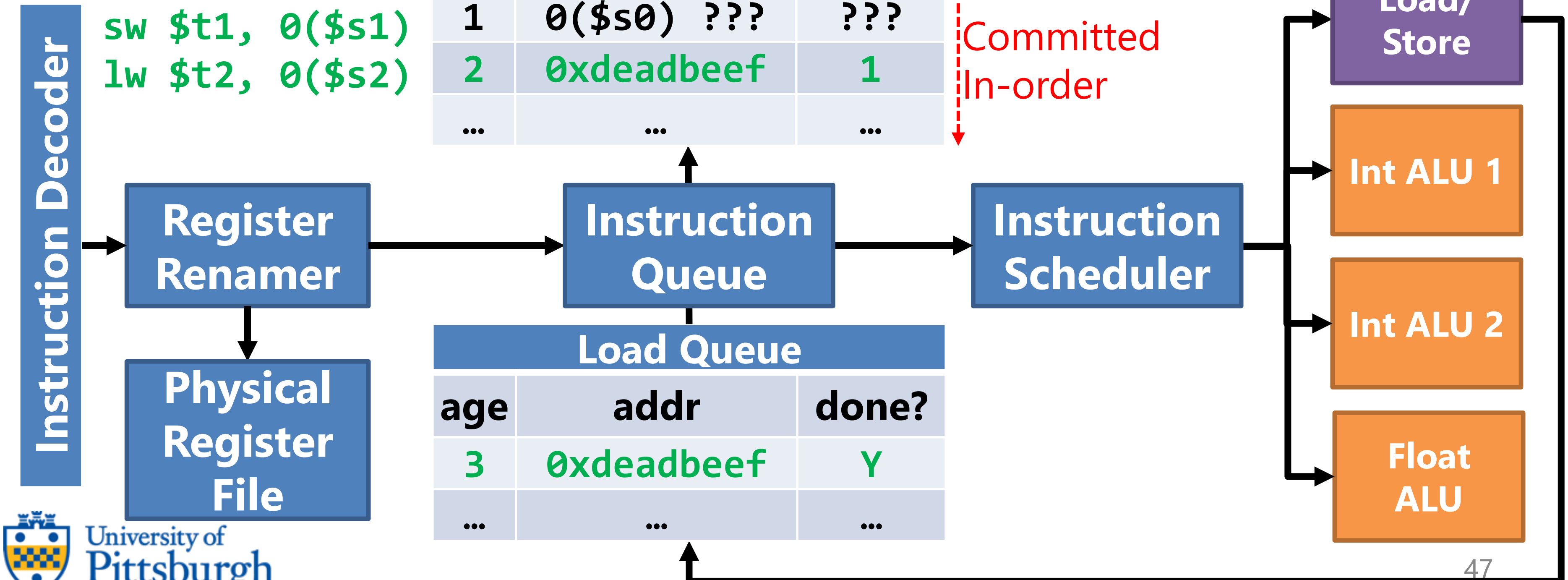
Store Queue

age	addr	value
1	0($s0) ???	???
2	0xdeadbeef	1
…	…	…

Committed In-order

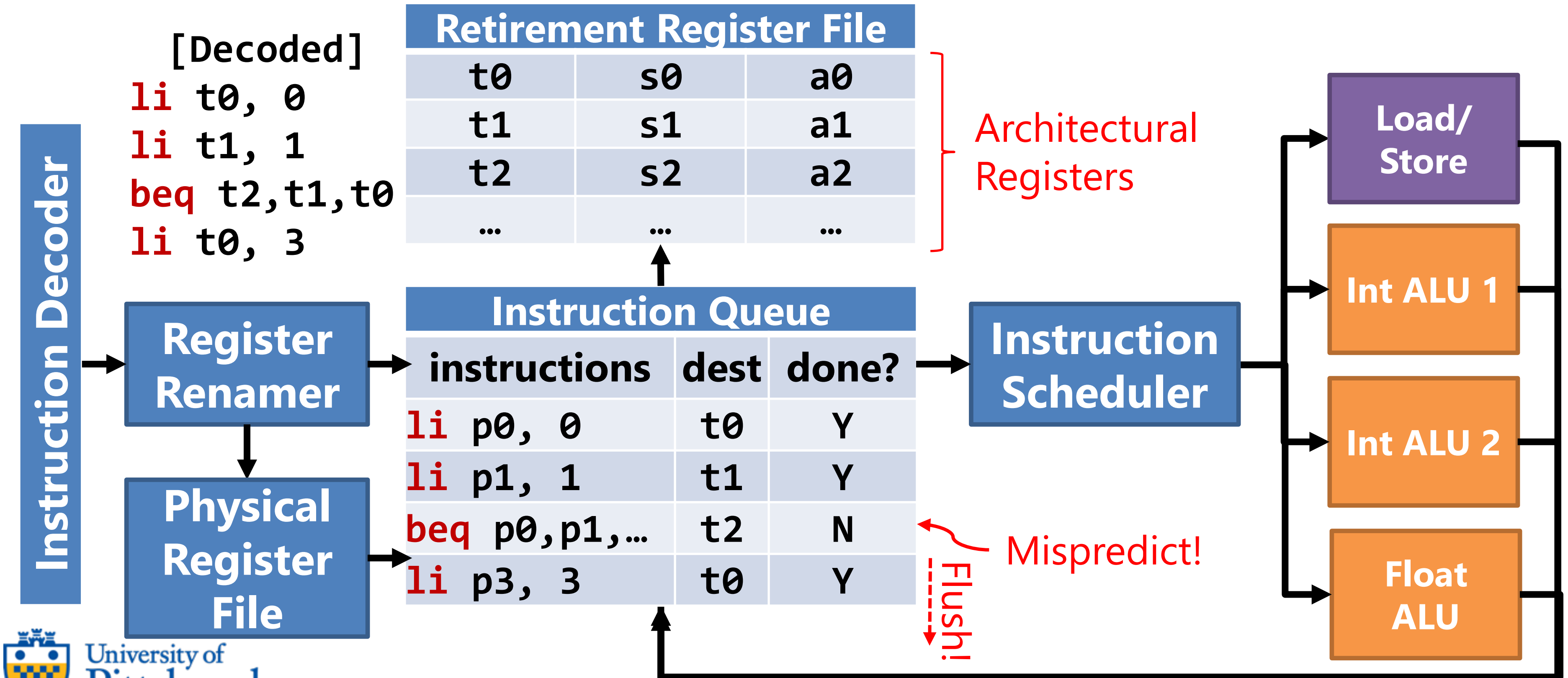
Load Queue

age	addr	done?
3	0xdeadbeef	Y
…	…	…



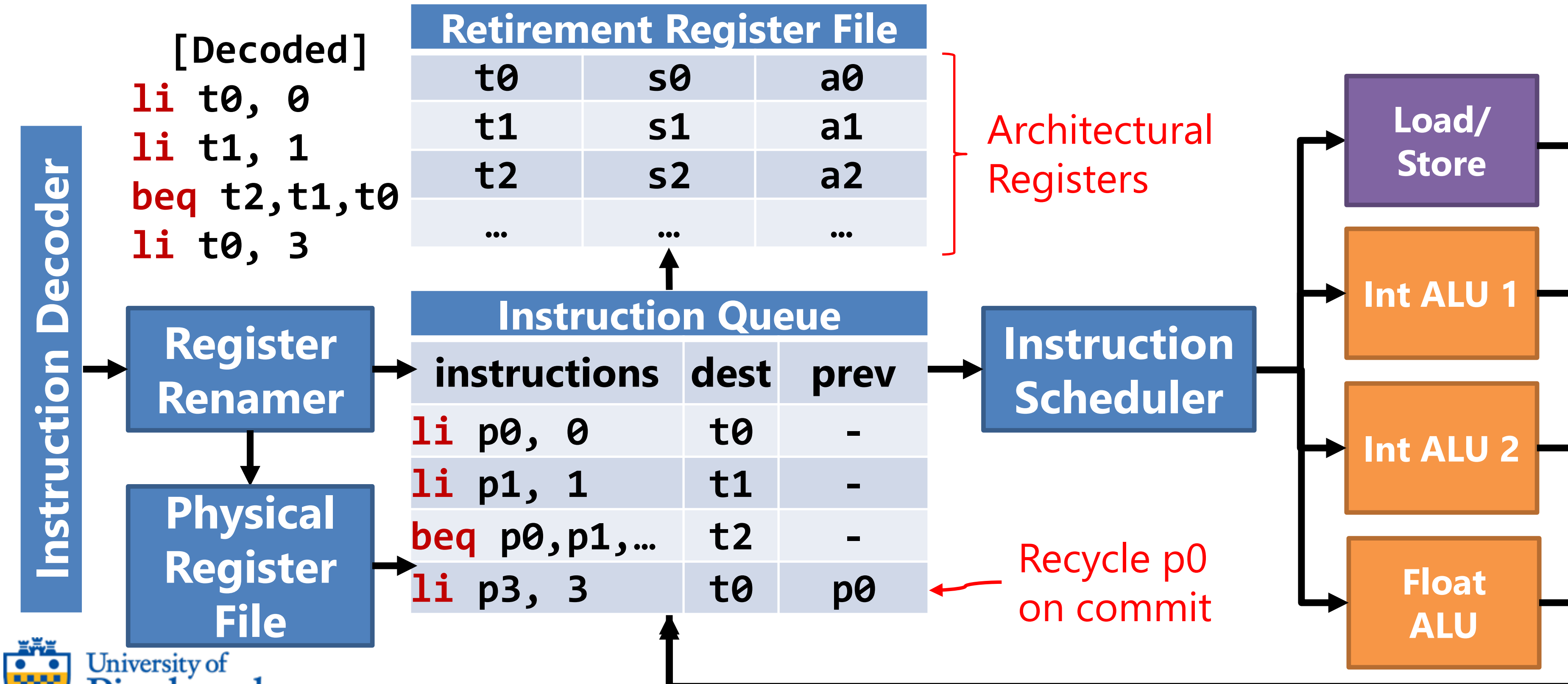
In-order Commit also solves branch misprediction

- What if processor finds out it **mispredicted** a branch?
  - Just **flush** instructions below it after the branch executes!
  - Also **restore** an **RAT snapshot** that was taken at point of branch.



In-order Commit also solves physical register recycling

- When can the processor recycle physical registers?
  - The **prev** column records previous physical register mapped to **dest**.
  - When `li p3, 3` commits, `p0` previously mapped to `t0` can be recycled



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**

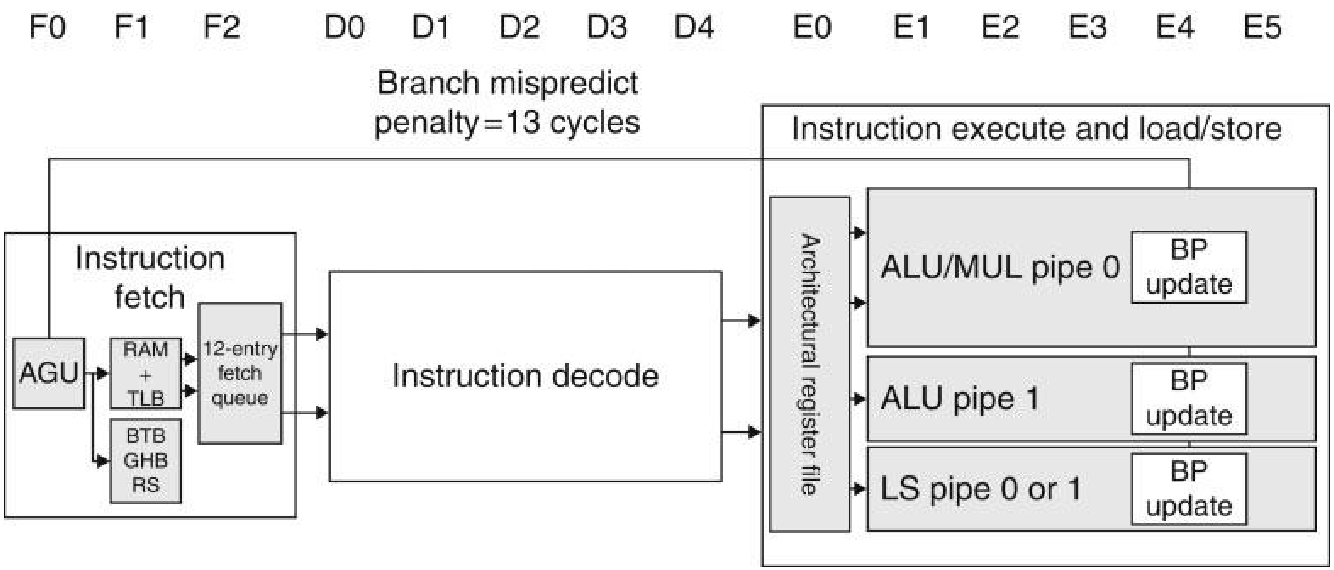
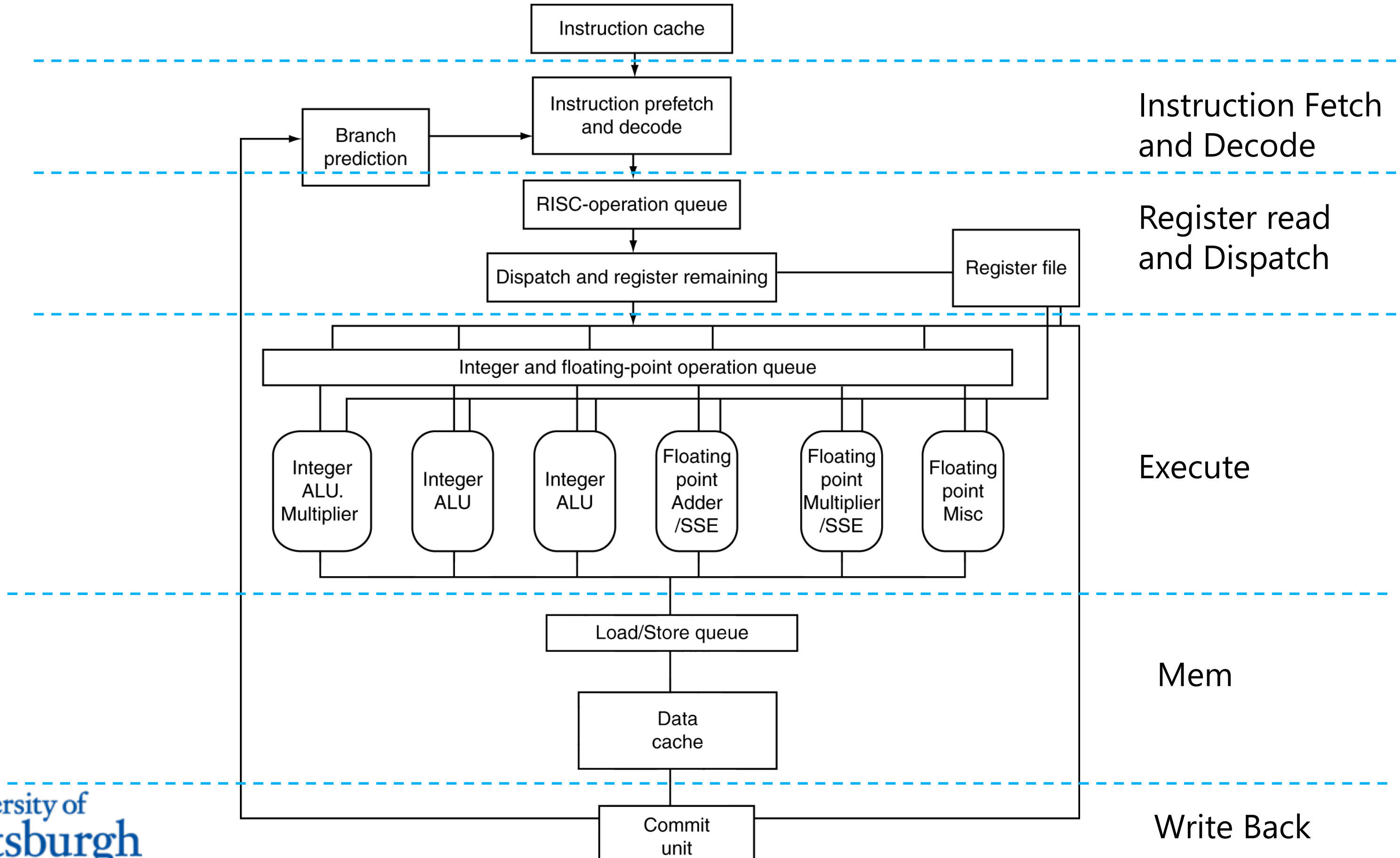


FIGURE 4.75 The A8 pipeline. The first three stages fetch instructions into a 12-entry instruction fetch buffer. The *Address Generation Unit* (AGU) uses a ***Branch Target Buffer*** **(BTB)**, ***Global History Buffer*** **(GHB)**, and a ***Return Stack*** **(RS)** to predict branches to try to keep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



University of Pittsburgh

The Intel Core i7 architecture

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

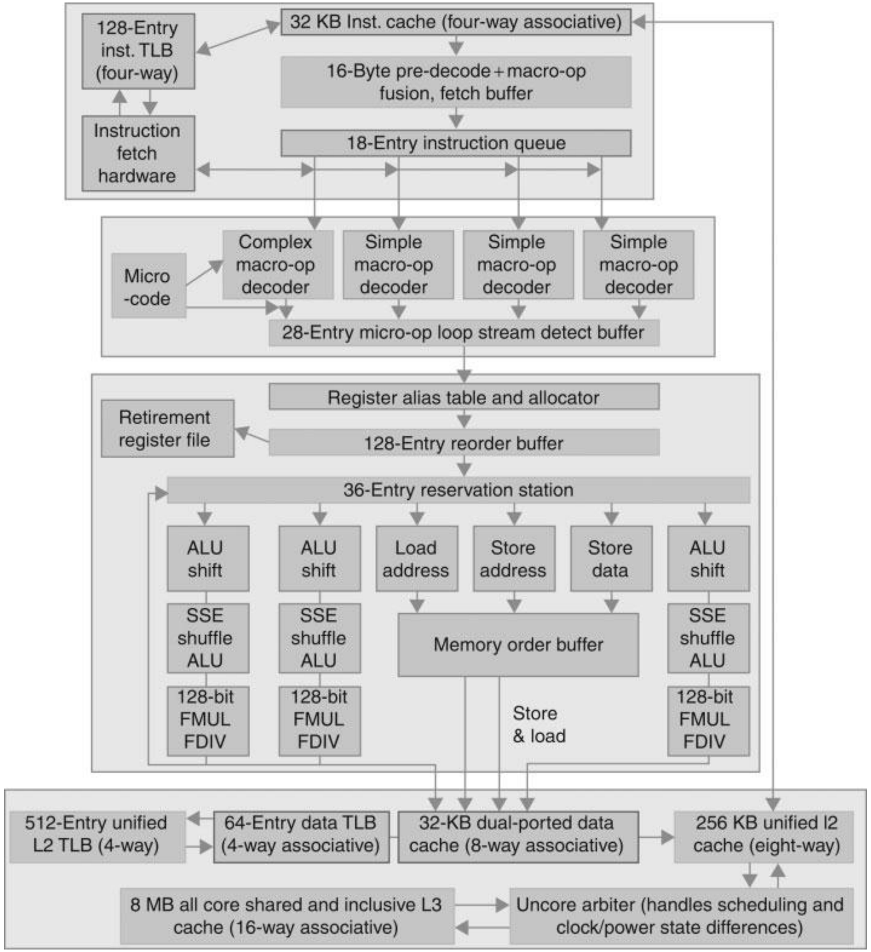


FIGURE 4.77 The Core i7 pipeline with memory components. The total pipeline depth is 14 stages, with **branch mispredictions costing 17 clock cycles**. This design can buffer 48 loads and 32 stores. It is a 4-wide processor but has 6 execution units of different types to reduce structural hazards.

University of Pittsburgh

Intel Core i7 Performance

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

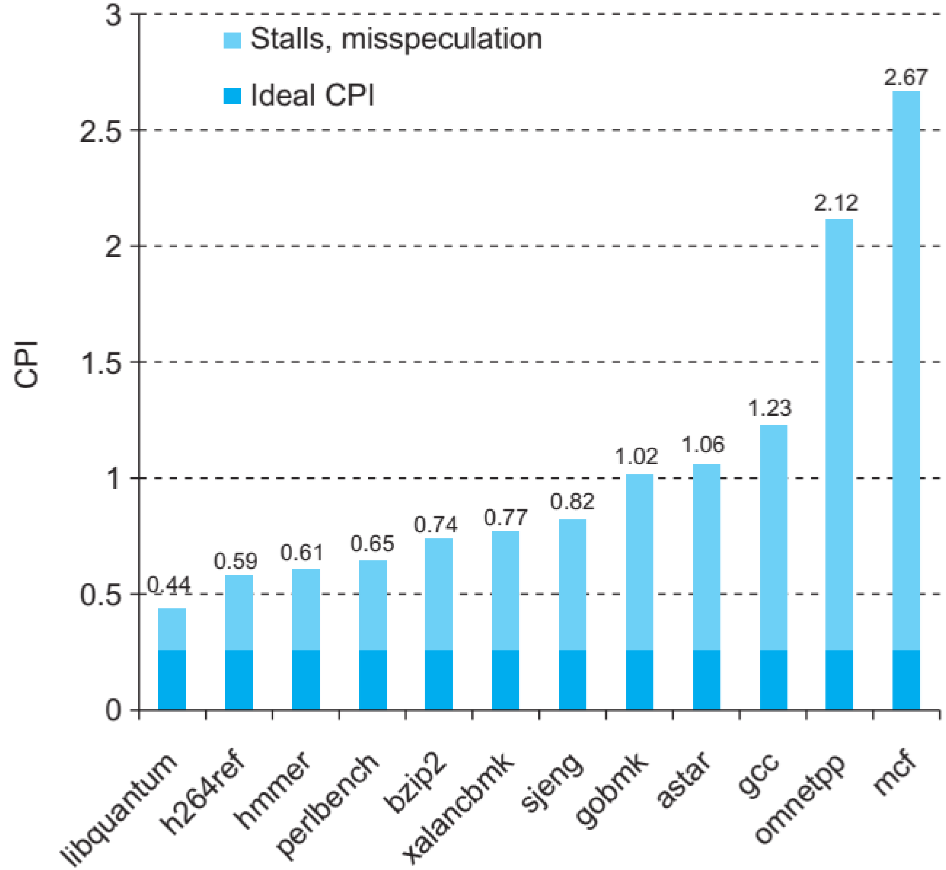


FIGURE 4.78 CPI of Intel Core i7 920 running SPEC2006 integer benchmarks.

Intel Core i7 Impact of Branch Misprediction

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

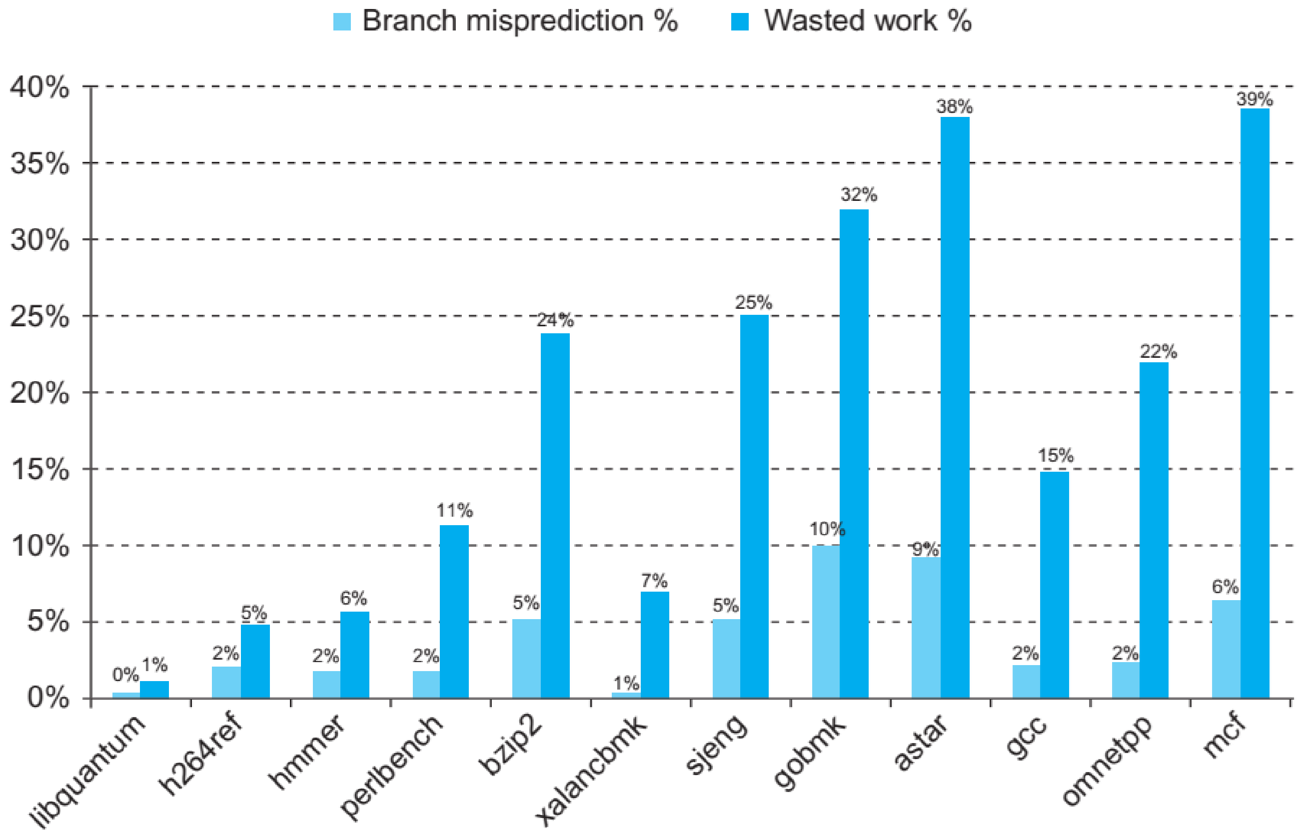


FIGURE 4.79 Percentage of branch mispredictions and wasted work due to unfruitful speculation of Intel Core i7 920 running SPEC2006 integer benchmarks.

Recap:
VLIWs vs SuperScalars

Ability to deal with hazards

- Hazards prevent the full exploitation of ILP (Instruction Level Parallelism)
- Which processor type has the advantage on each of these aspects?

	VLIW	Out-of-order SuperScalar
Dealing with data hazards involving registers		
Dealing with data hazards involving memory locations		
Dealing with control hazards		
Large instruction window for scheduling around hazards		

Ability to operate energy efficiently

- We learned that performance and power are two sides of the same coin.
- Which processor requires these power-hungry control structures?

	VLIW	**Out-of-order SuperScalar**
Big Register File		
Register Alias Table		
Instruction Queue		
Data Forwarding Wires		

University of Pittsburgh