Out of Order Processing

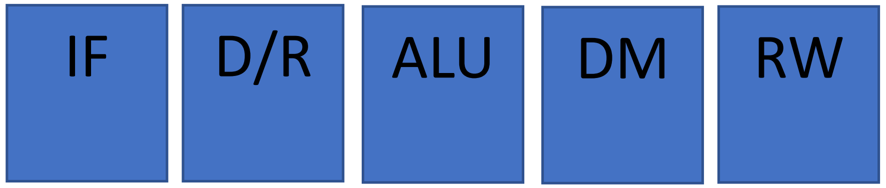
Manu Awasthi

July 3rd 2018

Computer Architecture Summer School 2018

Slide deck acknowledgements : Rajeev Balasubramonian (University of Utah), Computer Architecture: A Quantitative Approach; John Hennessy, David A. Patterson (Book)

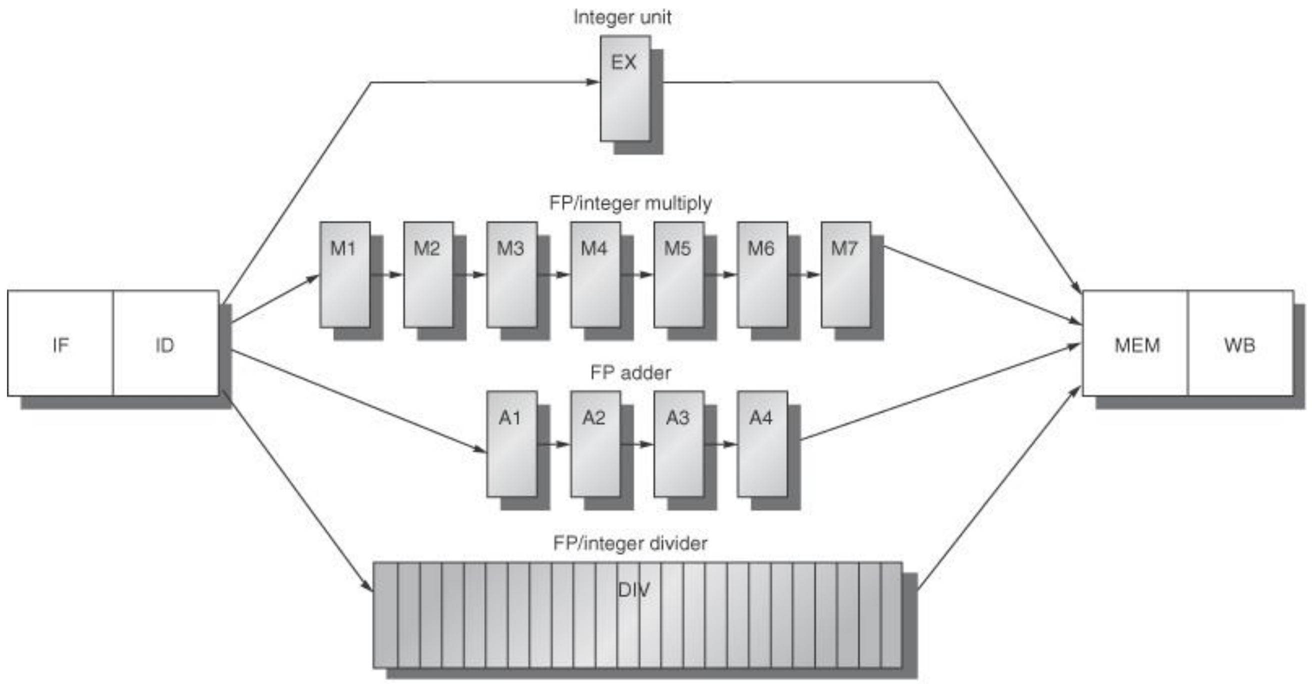
A Simple Pipeline



- Assumptions?
  - All stages take a the exact same amount of time
- Advantages?
  - Not all instructions same amount of time; simpler ones can finish faster
- Disadvantages
  - Later instructions can finish earlier

Multi Cycle Instructions

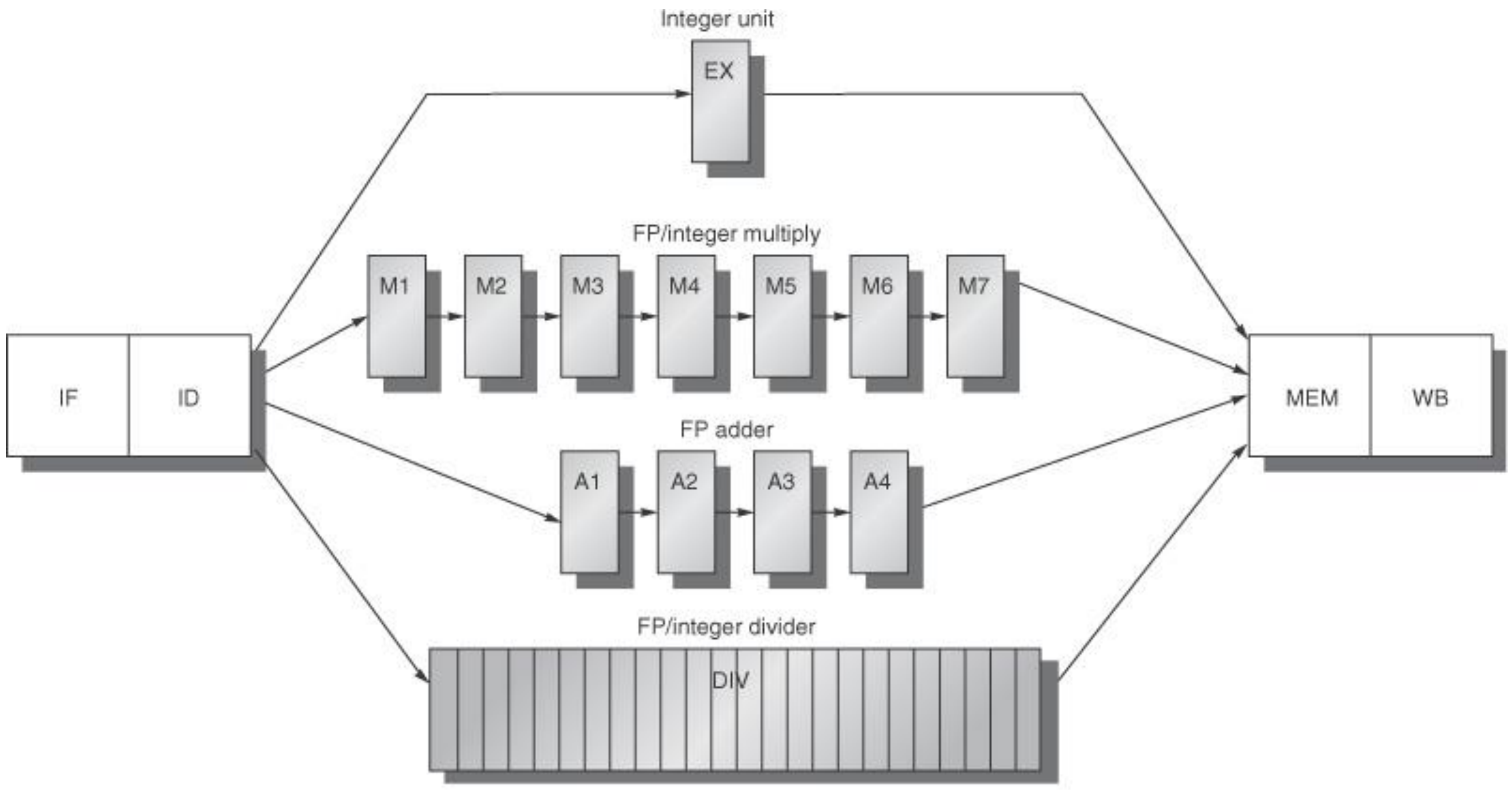
- Multiple parallel pipelines – each pipeline can have a different number of stages
- Instructions can now complete **out of order** – must make sure that writes to a register happen in the correct order



Effects of Multicycle Instructions

- Potentially multiple writes to the register file
- Frequent RAW hazards
- WAW hazards also possible
- WAR/RAR?
- Imprecise exception support

Recap Multicycle Instructions



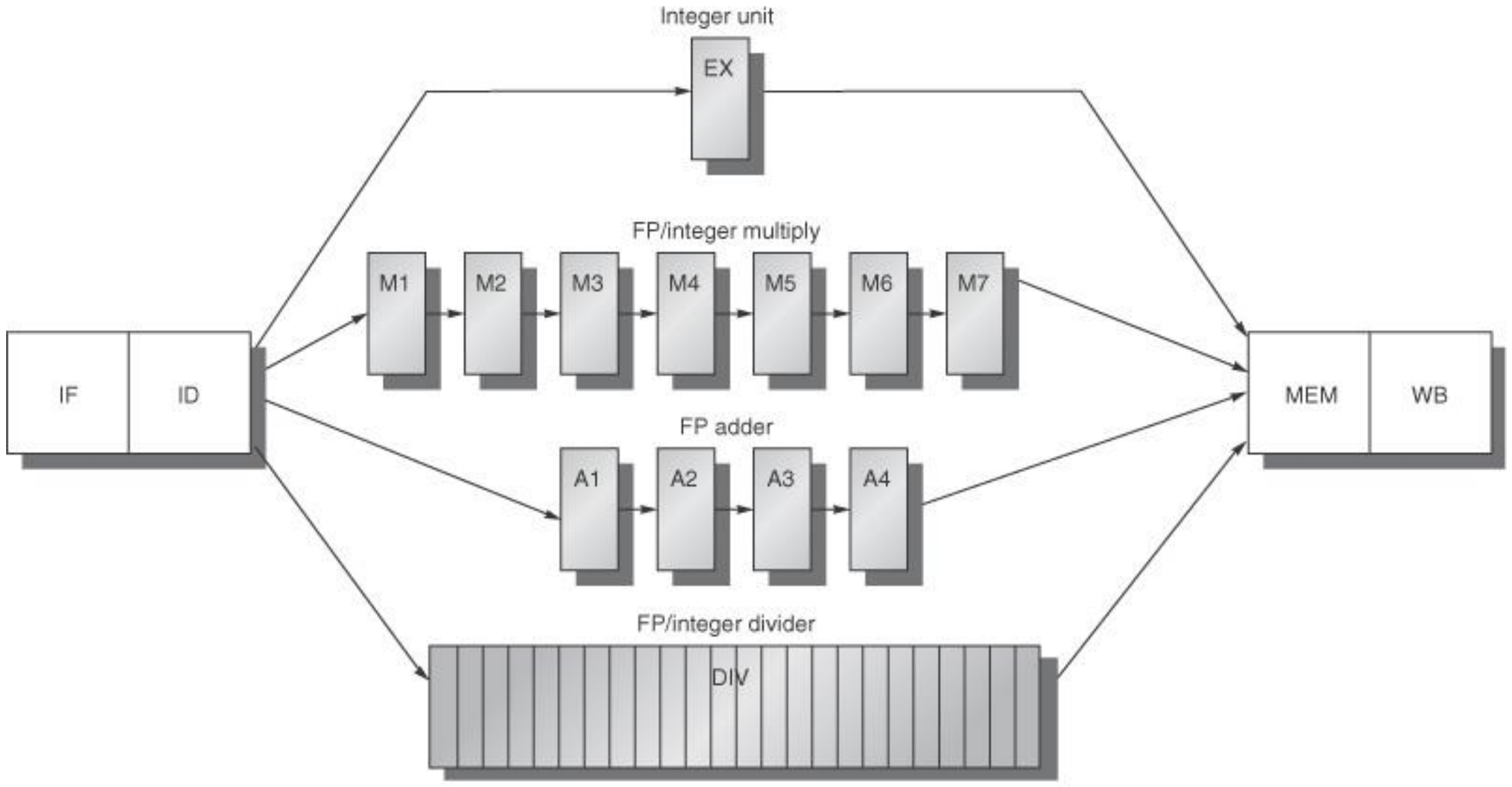
Dealing With Effects of Multicycle Instructions

- Multiple Writes to the register file (structural hazard)
  - Increase number of ports
  - Stall one of the writers during its decode (ID) stage
- WAW hazards
  - Detect hazards during ID; stall later instruction
- Imprecise Exceptions
  - Buffer results for instructions completing early (ROB) – need to make sure to return to the same state where it was left

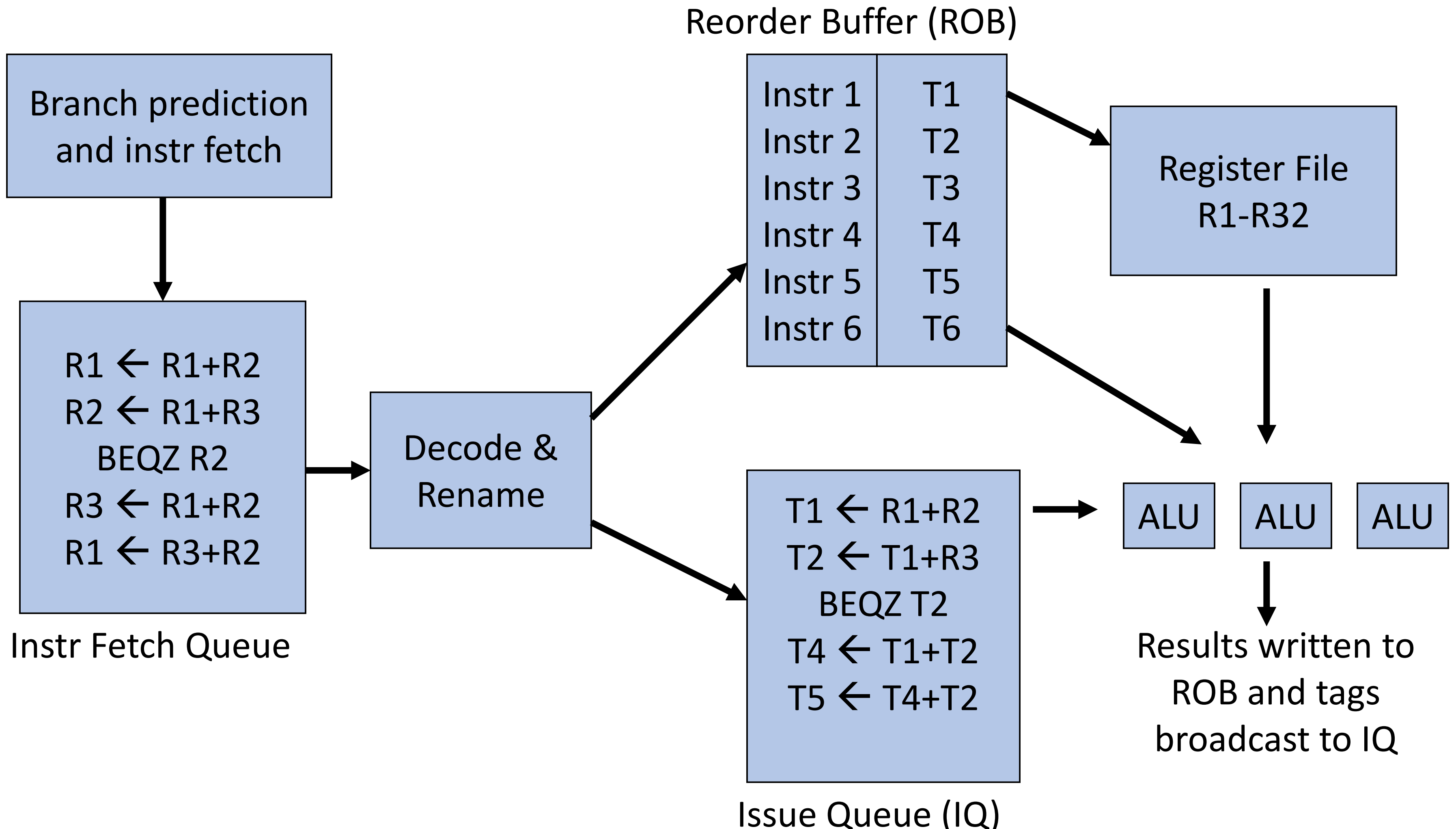
Exceptions

- “Exceptional circumstances” where normal instruction execution order is changed
- Causes
  - I/O device request
  - OS service from user program
  - Integer/FP arithmetic overflow
  - Misaligned memory access
  - Page fault
  - Hardware/power failure
- Interrupt/Fault/Exception used interchangeably

Dealing with Exceptions



An Out-of-Order Processor Implementation



Design Details

- Instructions enter the pipeline in order

- No need for branch delay slots if prediction happens in time

- Instructions leave the pipeline in order – all instructions that enter also get placed in the ROB – the process of an instruction leaving the ROB (in order) is called commit – an instruction commits only if it and all instructions before it have completed successfully (without an exception)

- A result is written into the register file only when the instruction commits – until then, the result is saved in a temporary register in the ROB

Design Details

- Instructions get renamed and placed in the issue queue – some operands are available (T1-T6; R1-R32), while others are being produced by instructions in flight (T1-T6)
- As instructions finish, they write results into the ROB (T1-T6) and broadcast the operand tag (T1-T6) to the issue queue – instructions now know if their operands are ready
- When a ready instruction issues, it reads its operands from T1-T6 and R1-R32 and executes (out-of-order execution)
- Can you have WAW or WAR hazards? By using more names (T1-T6), name dependences can be avoided

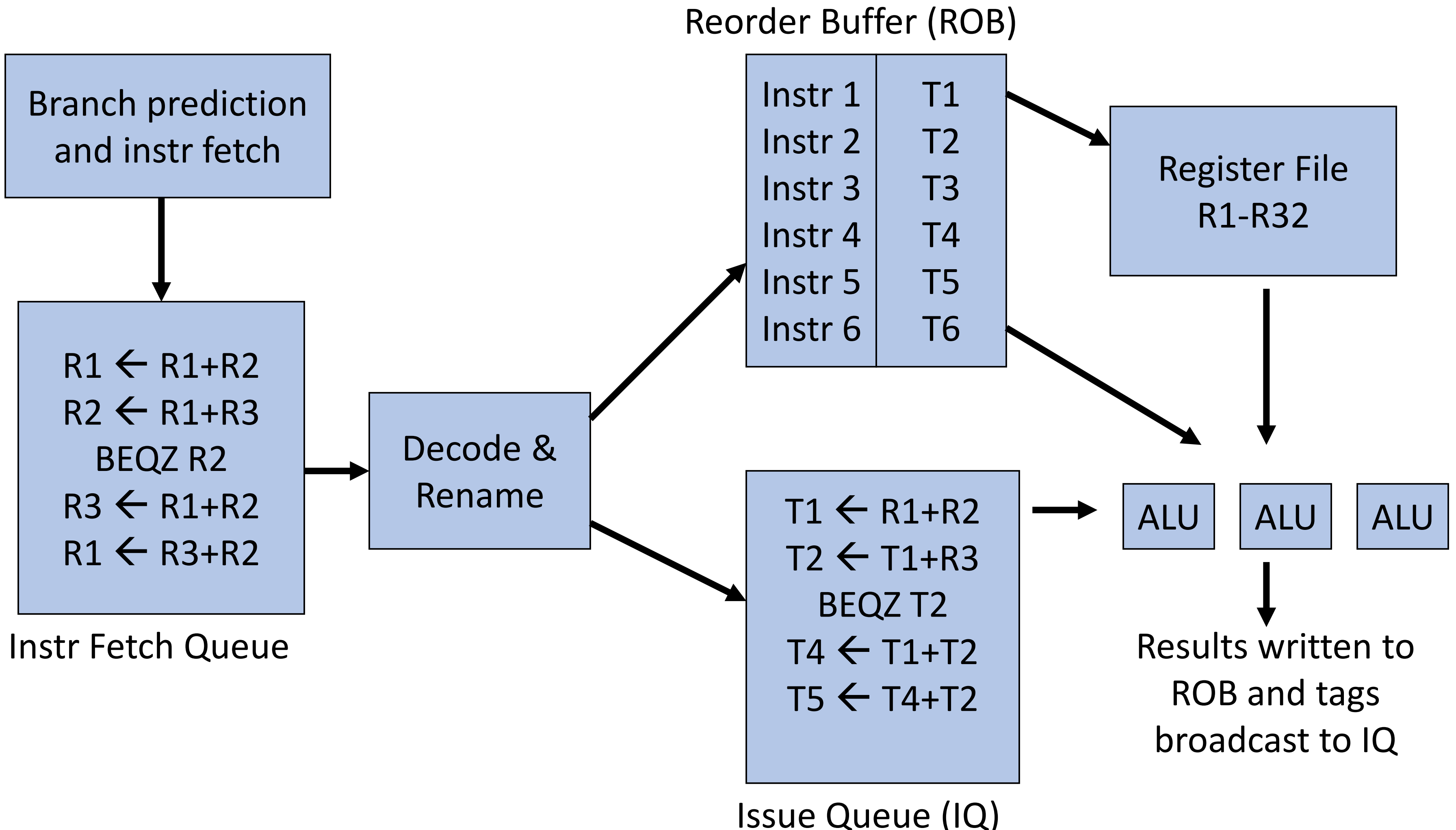
Design Details

- If instr-3 raises an exception, wait until it reaches the top of the ROB – at this point, R1-R32 contain results for all instructions up to instr-3 – save registers, save PC of instr-3, and service the exception

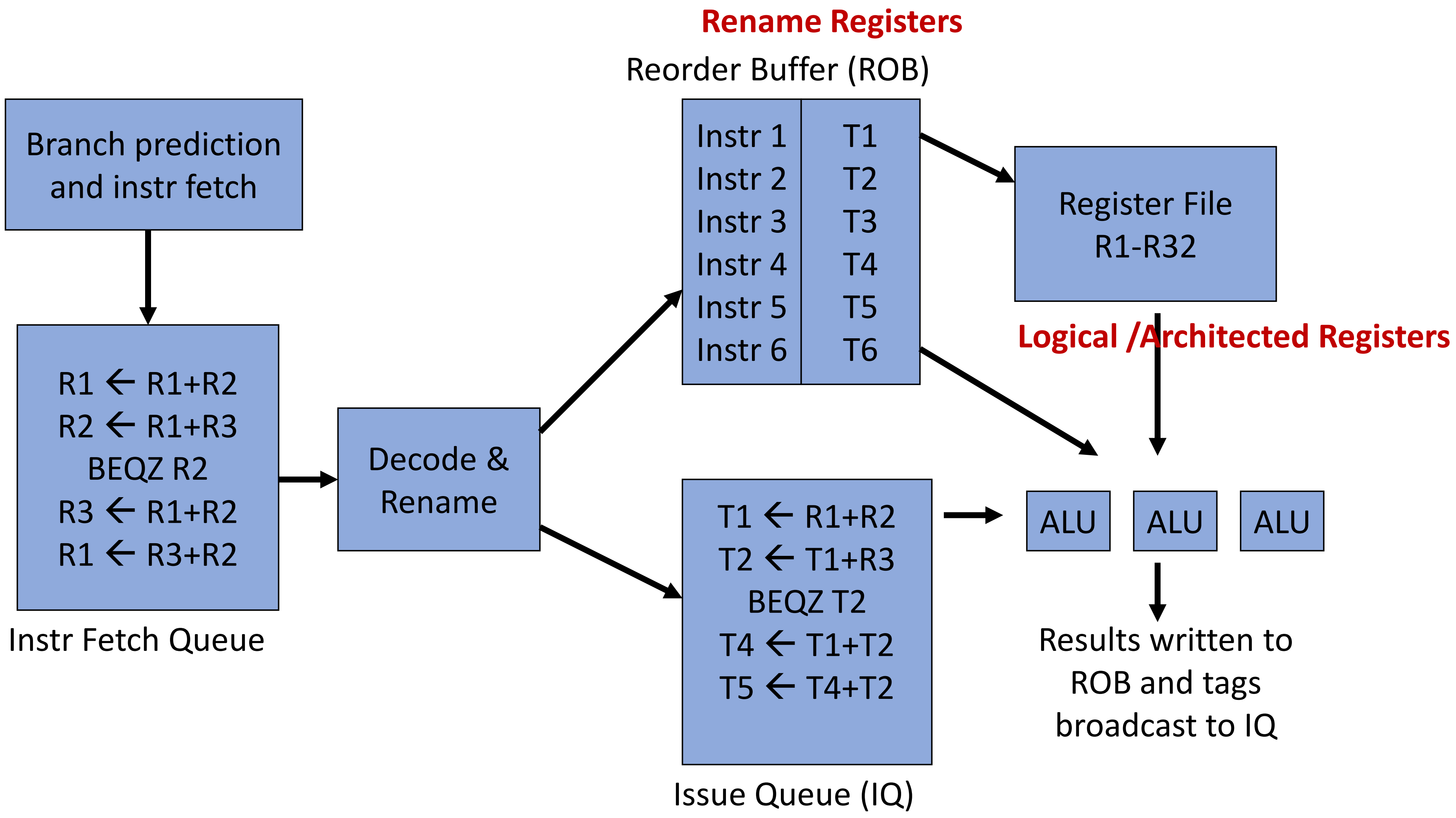
- If branch is a mispredict, flush all instructions after the branch and start on the correct path – mispredicted instructions will not have updated registers (the branch cannot commit until it has completed and the flush happens as soon as the branch completes)

- Potential problems: ?

Recap – OoO, Register Renaming

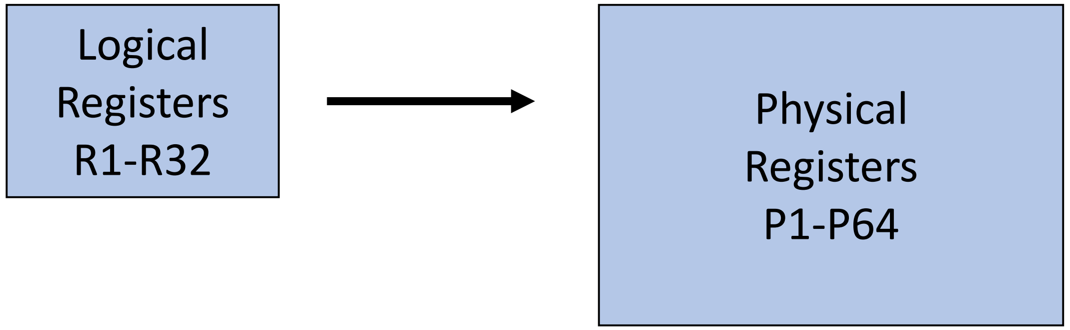


Out of Order Implementation



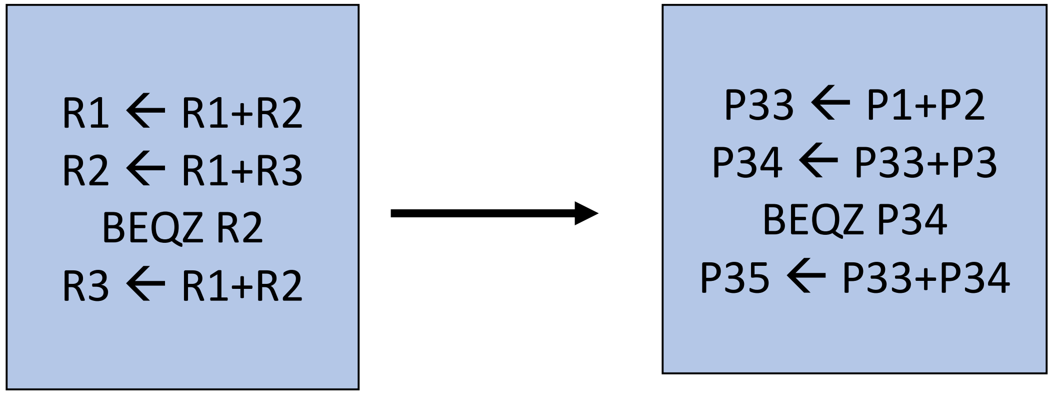
Managing Register Names

Temporary values are stored in the register file and not the ROB



At the start, R1-R32 can be found in P1-P32

Instructions stop entering the pipeline when P64 is assigned



```
R1 ← R1+R2
R2 ← R1+R3
BEQZ R2
R3 ← R1+R2
```

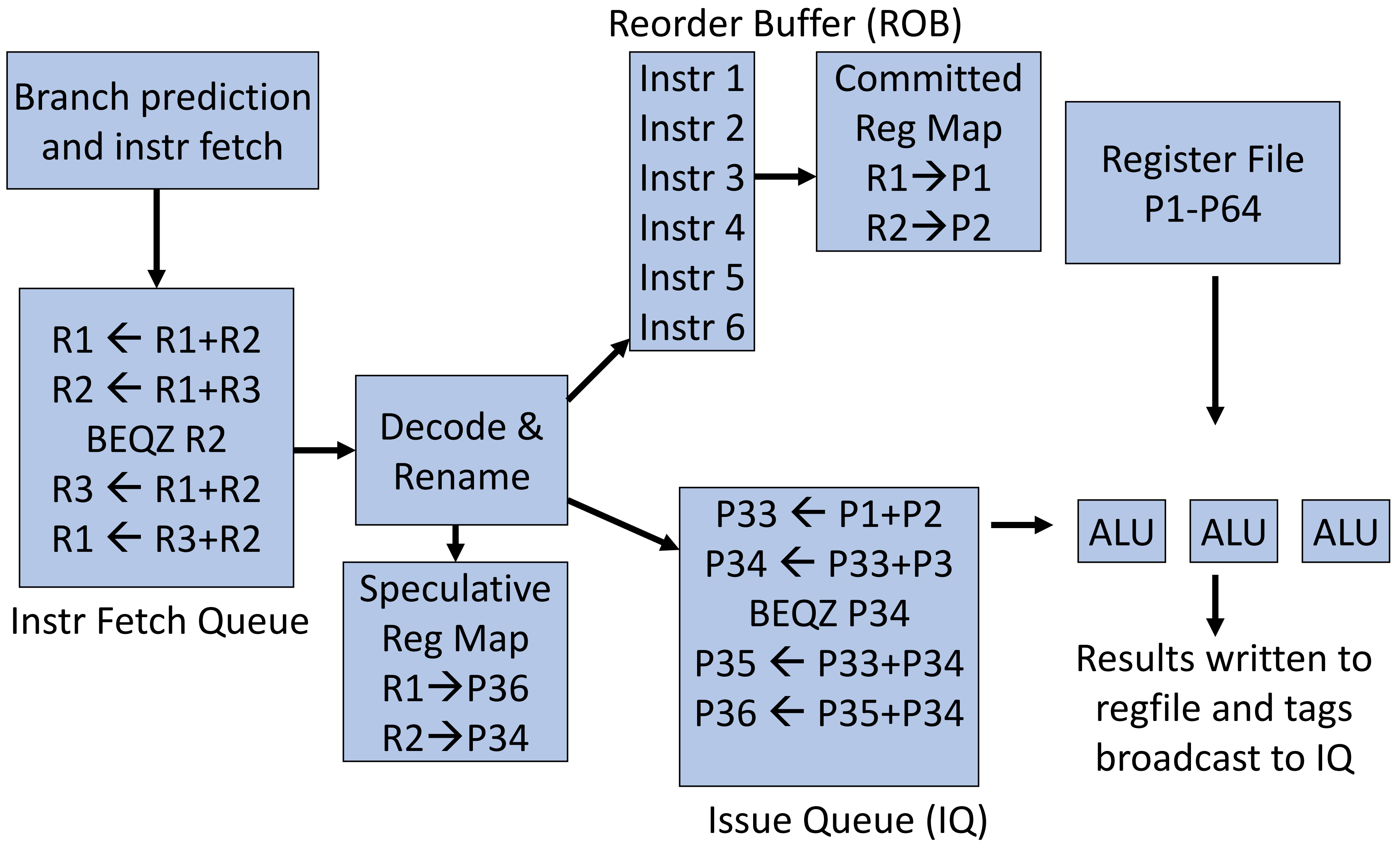
```
P33 ← P1+P2
P34 ← P33+P3
BEQZ P34
P35 ← P33+P34
```

What happens on commit?

The Commit Process

- On commit, no copy is required
- The register map table is updated – the “committed” value of R1 is now in P33 and not P1 – on an exception, P33 is copied to memory and not P1
- An instruction in the issue queue need not modify its input operand when the producer commits
- When instruction-1 commits, we no longer have any use for P1 – it is put in a free pool and a new instruction can now enter the pipeline $\rightarrow$ for every instr that commits, a new instr can enter the pipeline $\rightarrow$ number of in-flight instrs is a constant = number of extra (rename) registers

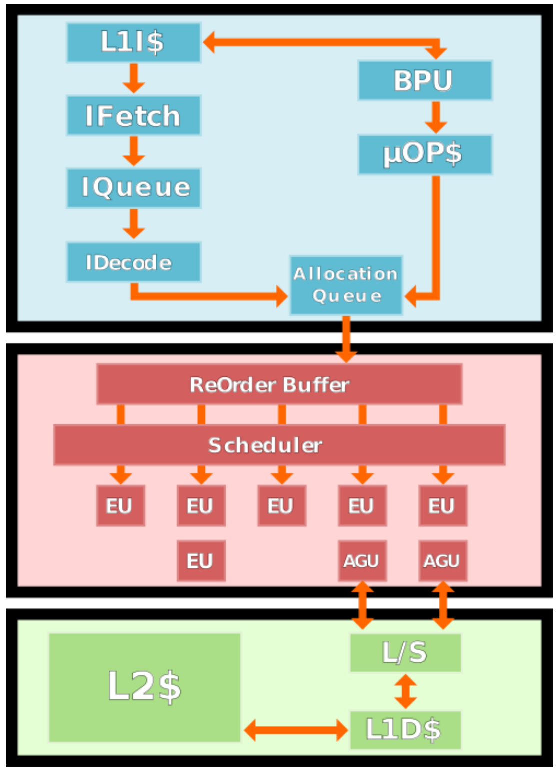
The Alpha 21264 Out-of-Order Implementation

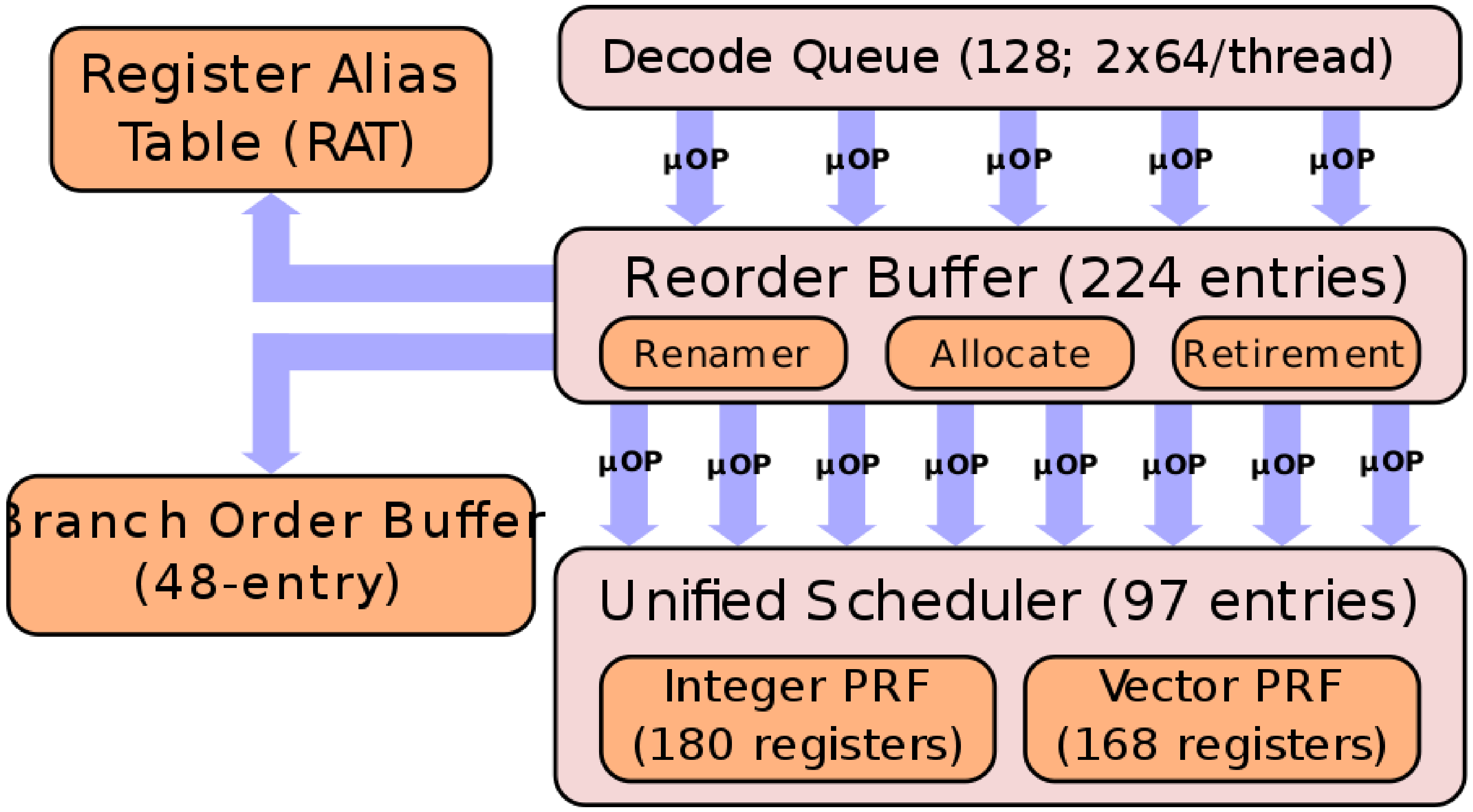


More Details

- When does the decode stage stall? When we either run out of registers, or ROB entries, or issue queue entries
- What type of instructions are present in the ROB?
- When does the ROB stall?
- **Issue width**: the number of instructions handled by each stage in a cycle. High issue width ➔ high peak ILP
- **Window size**: the number of in-flight instructions in the pipeline. Large window size ➔ high ILP
- No more WAR and WAW hazards because of rename registers – must only worry about RAW hazards

OoO Structures in Intel Processors





Register Alias Table (RAT)
Decode Queue (128; 2x64/thread)
µOP
Reorder Buffer (224 entries)
Renamer
Allocate
Retirement
Branch Order Buffer (48-entry)
Unified Scheduler (97 entries)
Integer PRF (180 registers)
Vector PRF (168 registers)

Intel Skylake (Client)

https://en.wikichip.org/wiki/intel/microarchitectures/skylake_(client)

Waking up Dependent Instructions

- Inorder pipelines – an instruction can leave the decode stage based on the knowledge of when the input will be available, not when it is computed

- In OoO pipelines, instruction can leave the issue queue (woken up) before its inputs are know, based on knowledge of when they will be available – wakeup is speculative based on expected latency of producer

Loads and Stores

Ld	R1 ← [R2]
Ld	R3 ← [R4]
St	R5 → [R6]
Ld	R7 ← [R8]
Ld	R9←[R10]

What if the issue queue also had load/store instructions?
Can we continue executing instructions out-of-order?

Memory Dependence Checking

Ld	0x abcdef
Ld	
St	
Ld	
Ld	0x abcdef
St	0x abcd00
Ld	0x abc000
Ld	0x abcd00

- The issue queue checks for register dependences and executes instructions as soon as registers are ready

- Loads/stores access memory as well – must check for hazards for memory as well

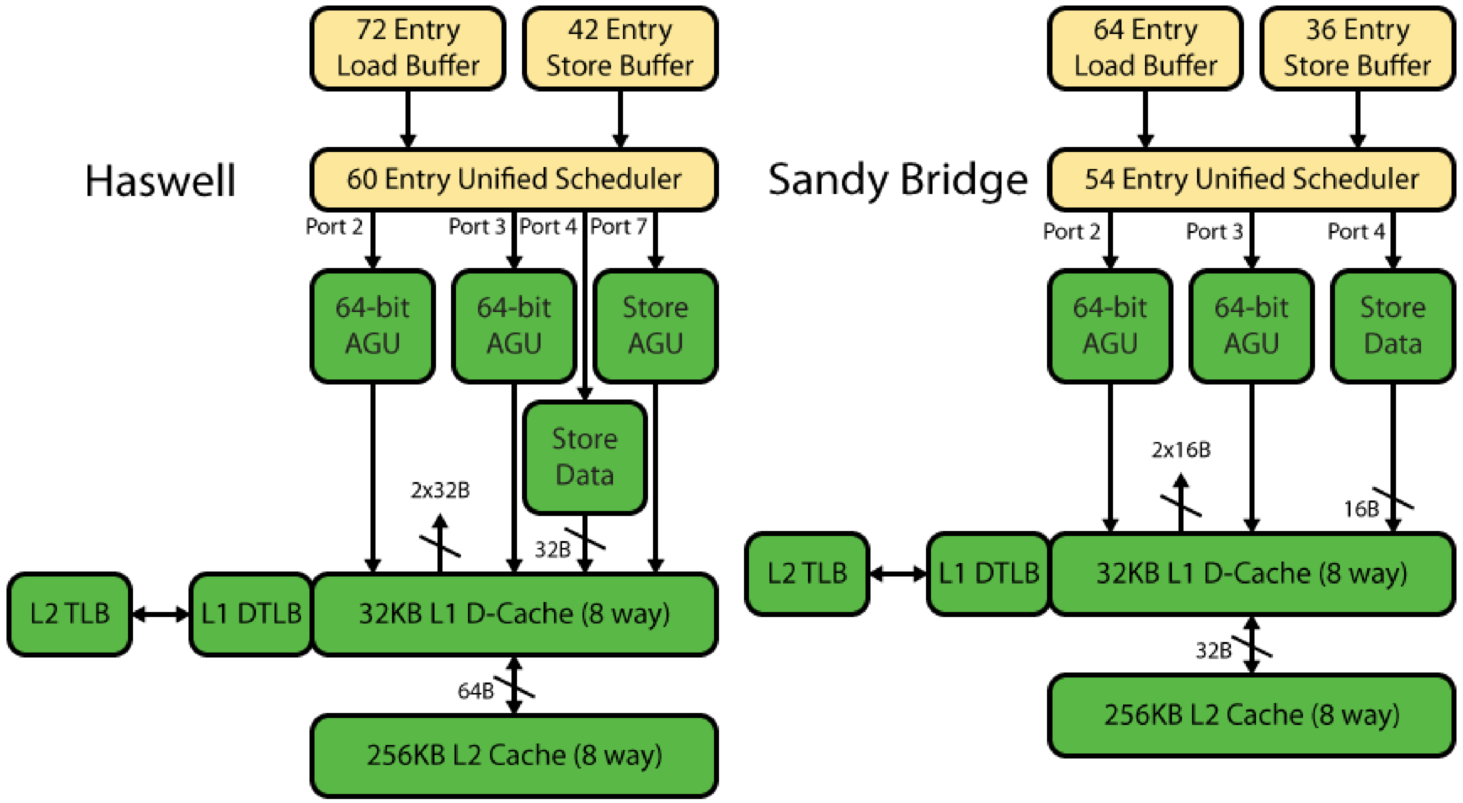
- Hence, first check for register dependences to compute effective addresses; then check for memory dependences

Memory Dependence Checking

Ld	0x abcdef
Ld	
St	
Ld	
Ld	0x abcdef
St	0x abcd00
Ld	0x abc000
Ld	0x abcd00

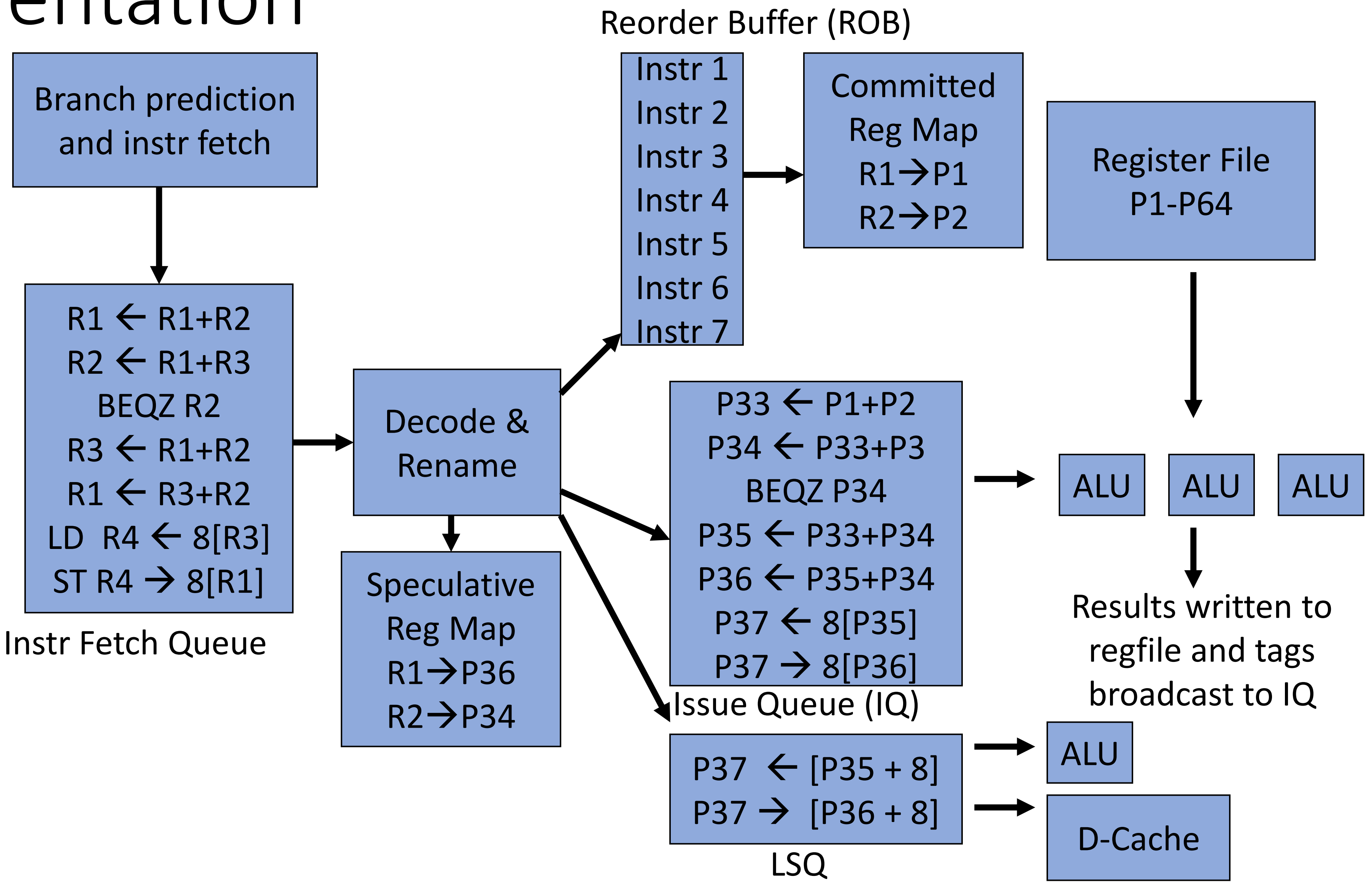
- Load and store addresses are maintained in program order in the Load/Store Queue (LSQ)
- Loads can issue if they are guaranteed to not have true dependences with earlier stores
- Stores can issue only if we are ready to modify memory (can not recover if an earlier instruction raises an exception)

LSQs Intel Processors



https://www.realworldtech.com/haswell-cpu/5/

The Alpha 21264 Out-of-Order Implementation


Reorder Buffer (ROB)
Branch prediction and instr fetch
Instr 1
Instr 2
Instr 3
Instr 4
Instr 5
Instr 6
Instr 7
Committed Reg Map
R1→P1
R2→P2
Register File P1-P64
R1 ← R1+R2
R2 ← R1+R3
BEQZ R2
R3 ← R1+R2
R1 ← R3+R2
LD R4 ← 8[R3]
ST R4 → 8[R1]
Instr Fetch Queue
Decode & Rename
Speculative Reg Map
R1→P36
R2→P34
P33 ← P1+P2
P34 ← P33+P3
BEQZ P34
P35 ← P33+P34
P36 ← P35+P34
P37 ← 8[P35]
P37 → 8[P36]
Issue Queue (IQ)
ALU
ALU
ALU
Results written to regfile and tags broadcast to IQ
P37 ← [P35 + 8]
P37 → [P36 + 8]
LSQ
ALU
D-Cache