CS 110
Computer Architecture
Superscalar CPUs

Instructors:

Sören Schwertfeger & Chundong Wang

School of Information Science and Technology SIST

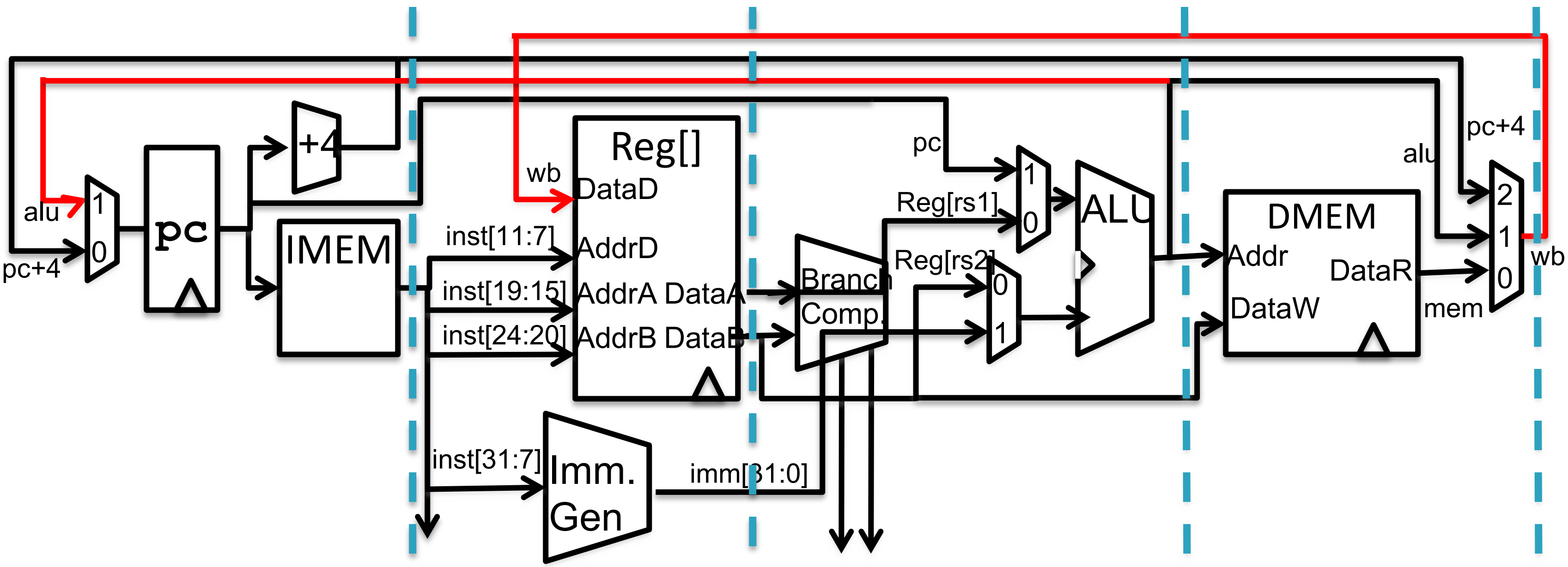
ShanghaiTech University

Slides based on UC Berkley's CS61C

Agenda

- Pipelining Review
- Processor Performance - Overview
- Complex Pipelines
- Static Multiple Issues (VLIW)
- Dynamic Multiple Issues (Superscalar)

Pipelining RISC-V RV32I Datapath



Instruction Fetch (F)

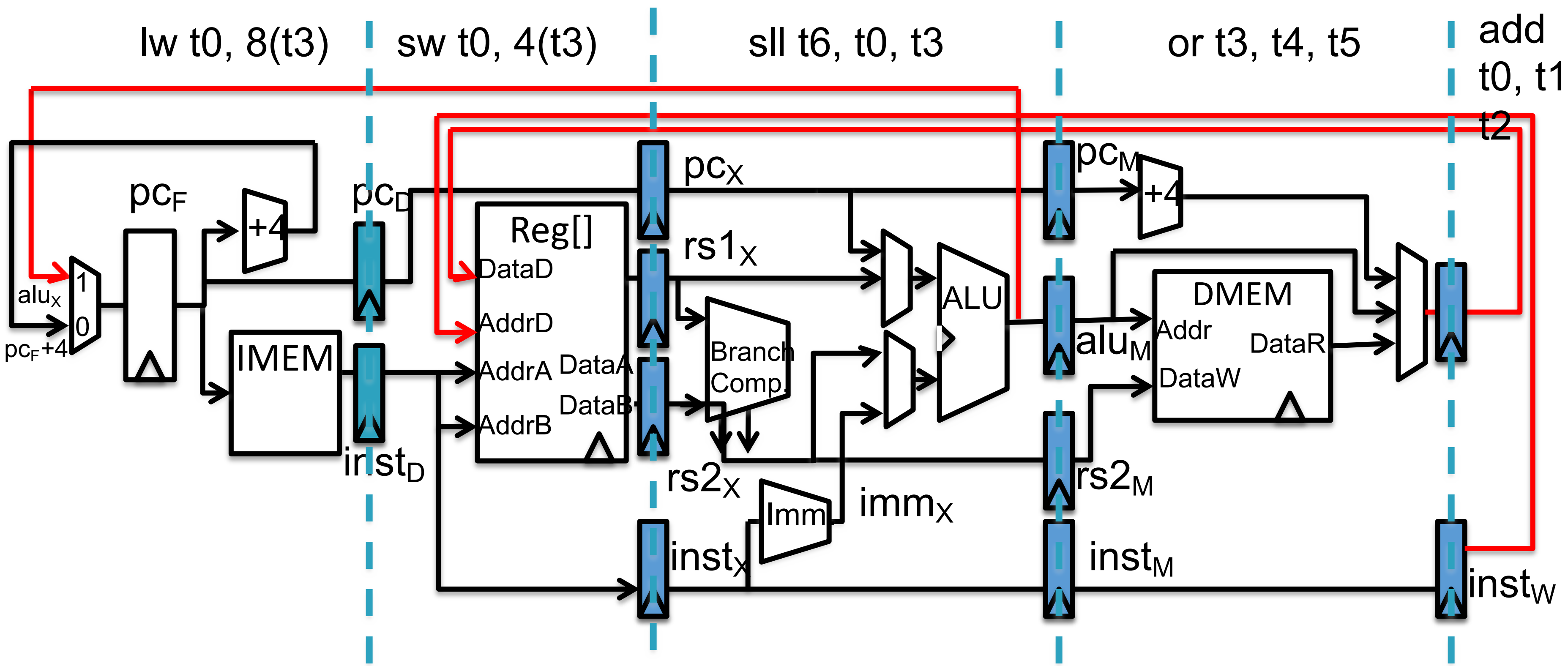
Instruction Decode/Register Read (D)

ALU Execute (X)

Memory Access (M)

Write Back (W)

Each stage operates on different instruction



Pipeline registers separate stages, hold data for each instruction in flight

Pipelining Hazards

A *hazard* is a situation that prevents starting the next instruction in the next clock cycle

1) Structural hazard

- A required resource is busy (e.g. needed in multiple stages)

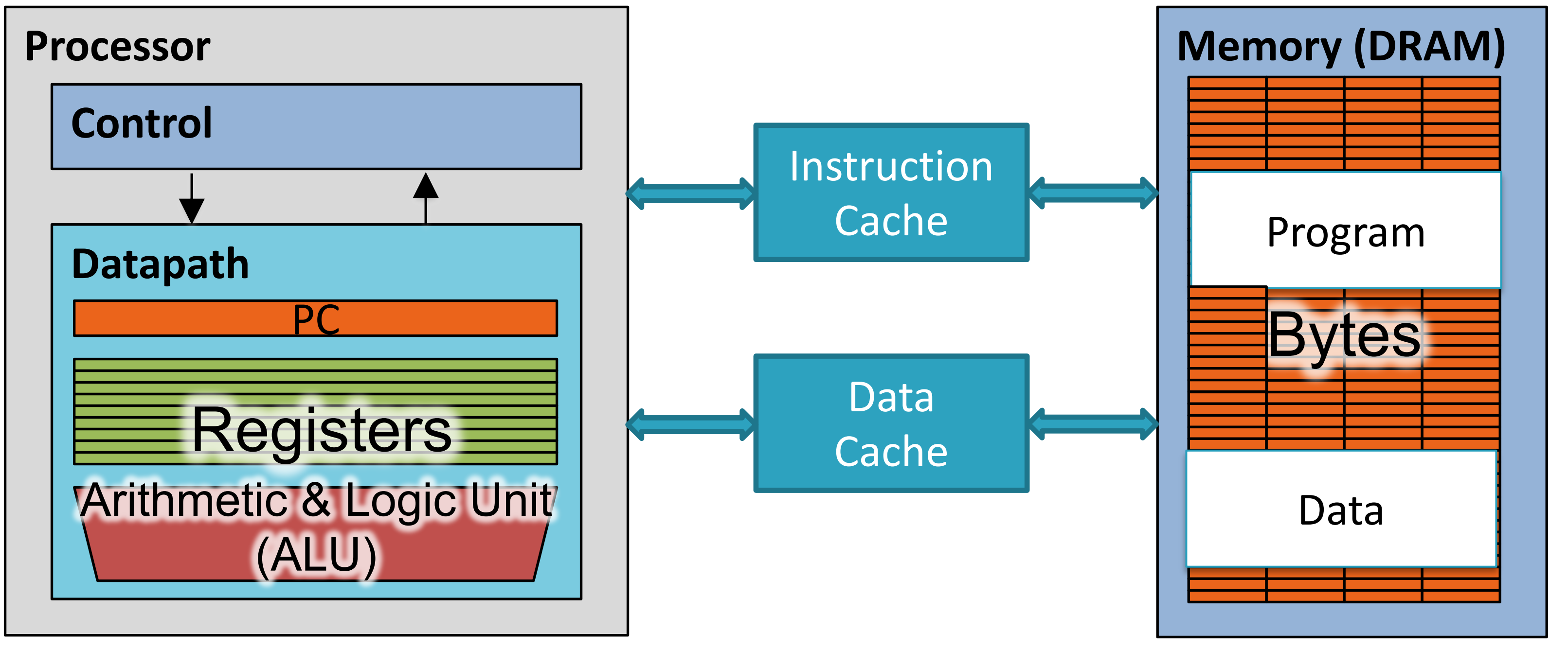
2) Data hazard

- Data dependency between instructions
- Need to wait for previous instruction to complete its data read/write

3) Control hazard

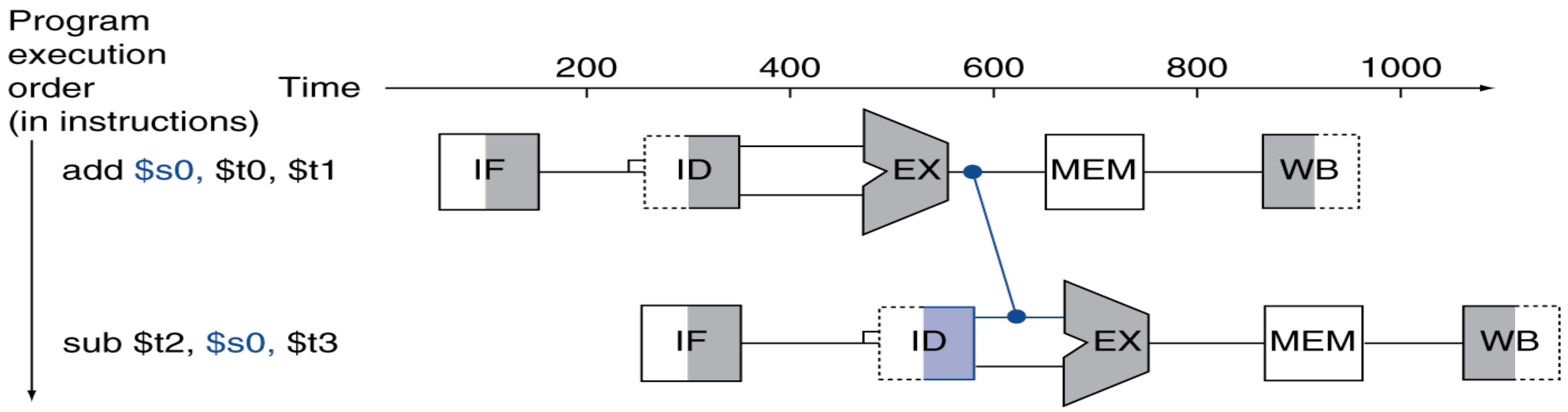
- Flow of execution depends on previous instruction

Structual Hazard Memory Access -> Instruction and Data Caches

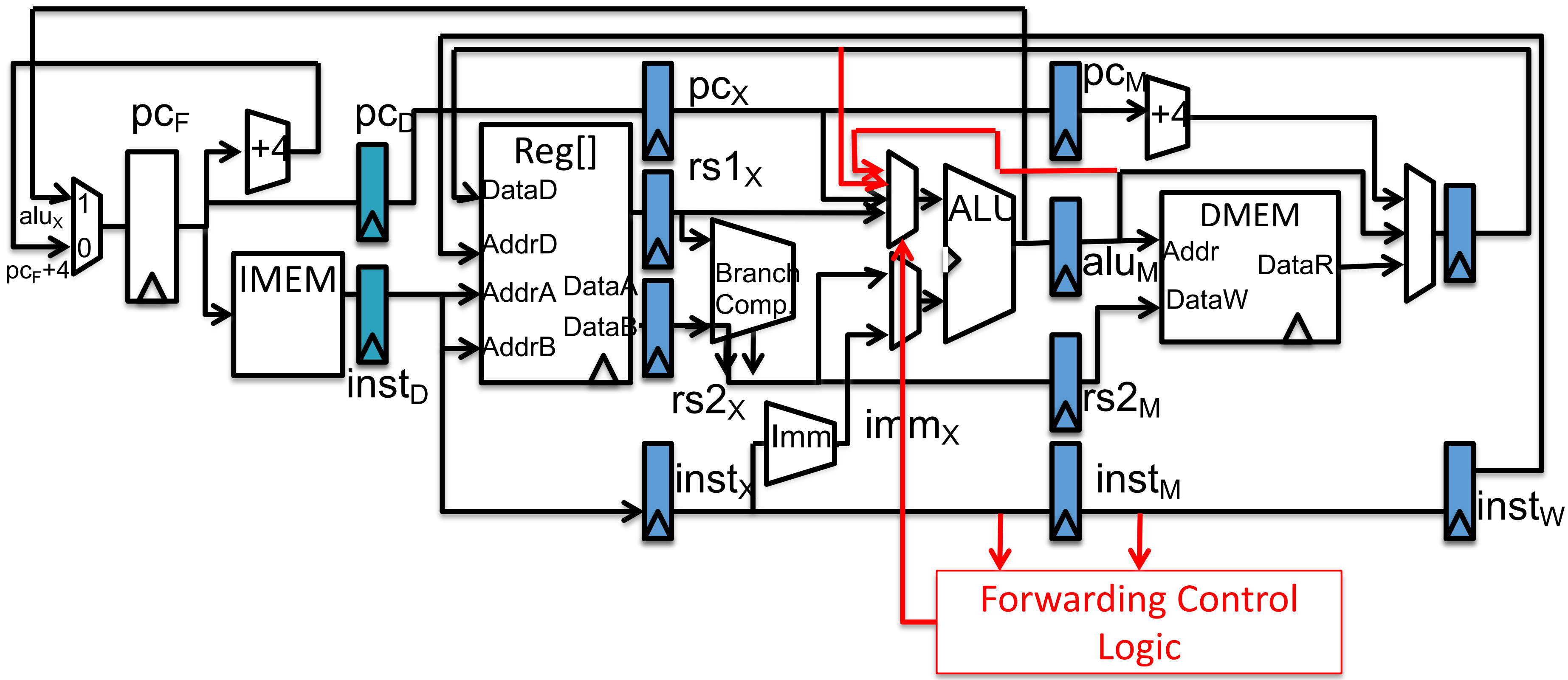


Data Hazard Example: Solved by Forwarding (aka Bypassing)

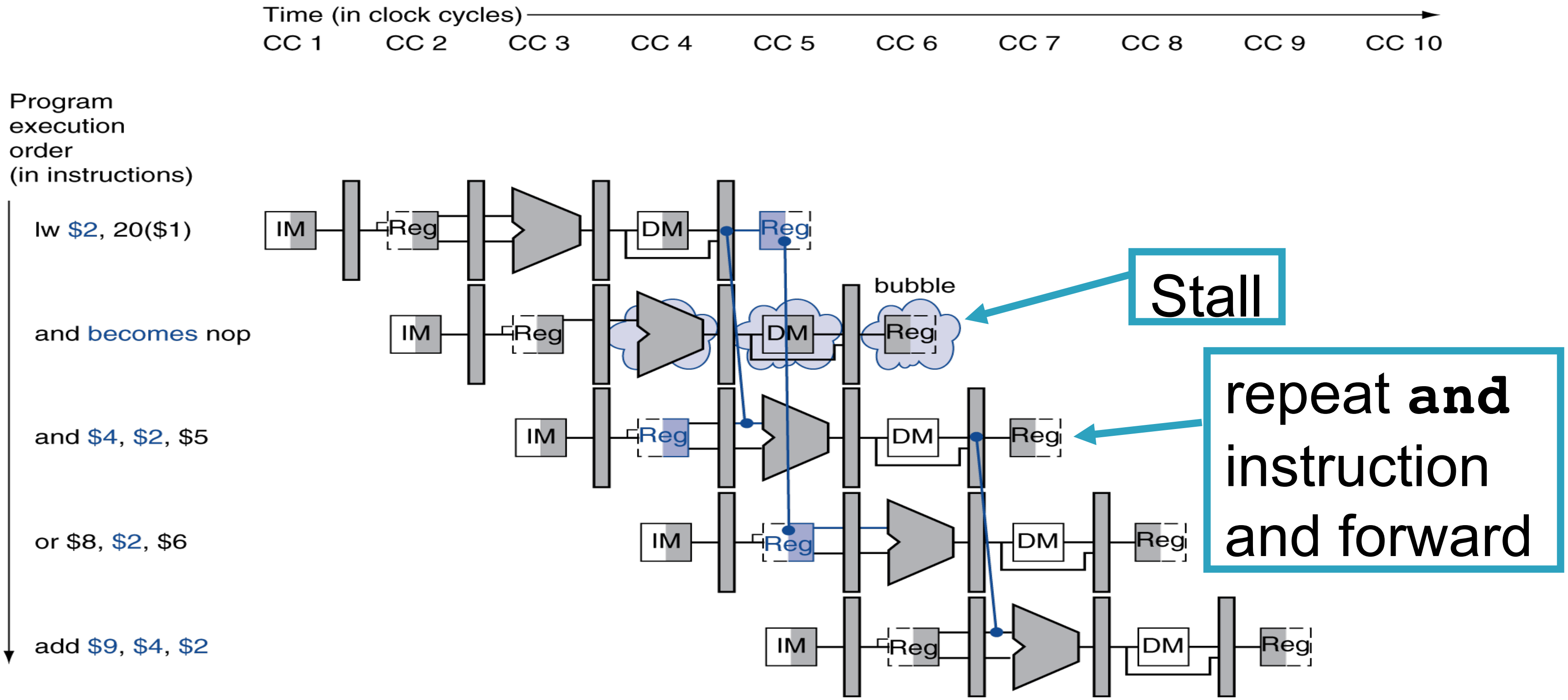
- Use result when it is computed
  - Don't wait for it to be stored in a register
  - Requires extra connections in the datapath



Forwarding Path



lw: Stall Pipeline

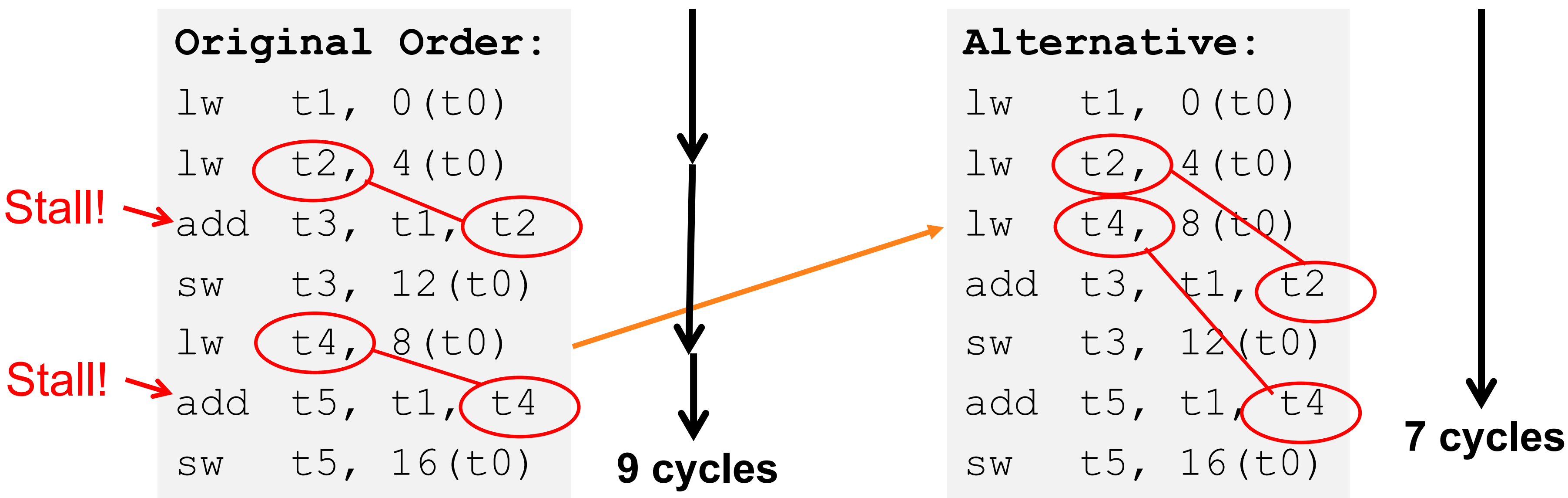


`lw` Data Hazard

- Slot after a load is called a *load delay slot*
  - If that instruction uses the result of the load, then the hardware will stall for one cycle
  - Equivalent to inserting an explicit **`nop`** in the slot
    - except the latter uses more code space
  - Performance loss
- Idea:
  - Put unrelated instruction into load delay slot
  - No performance loss!

Code Scheduling to Avoid Stalls

- Reorder code to avoid use of load result in the next instr!
- RISC-V code for **`A[3]=A[0]+A[1]; A[4]=A[0]+A[2]`**



Control Hazards

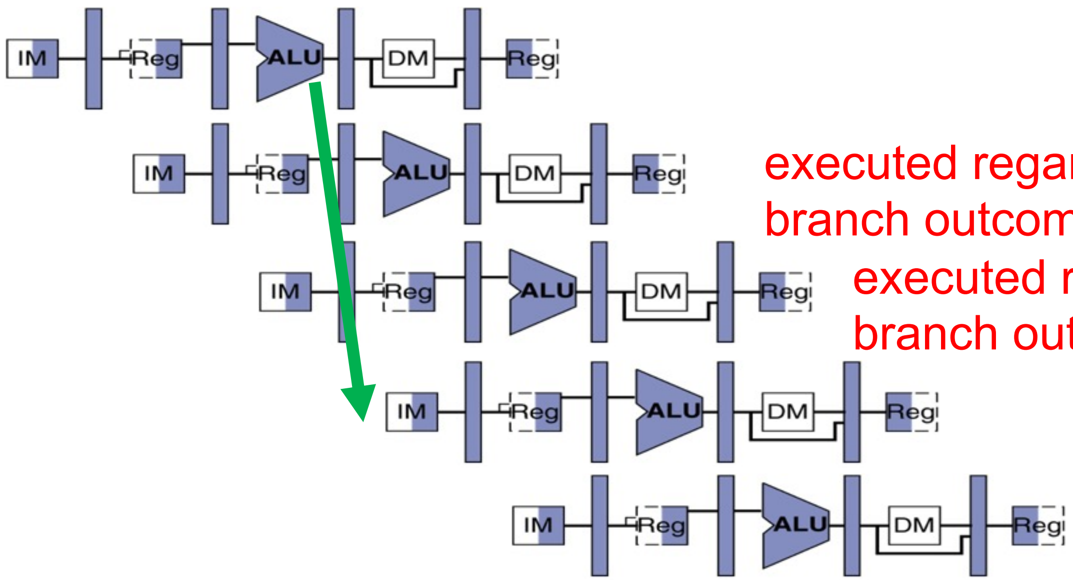
beq t0, t1, label

sub t2, s0, t5

or t6, s0, t3

xor t5, t1, s0

sw s0, 8(t3)



executed regardless of branch outcome!

executed regardless of branch outcome!!!

PC updated reflecting branch outcome

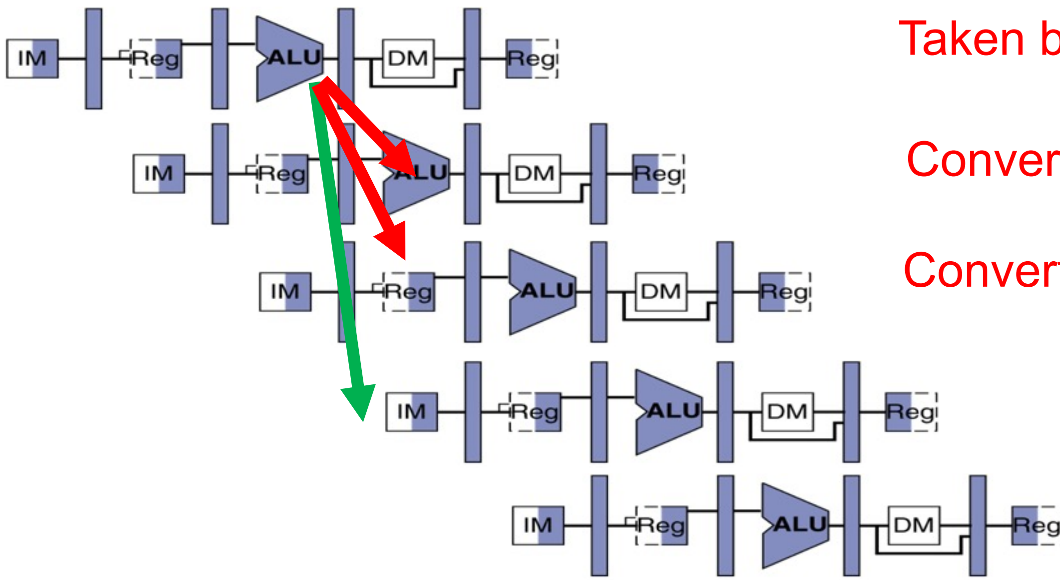
Kill Instructions after Branch if Taken

beq t0, t1, label

sub t2, s0, t5

or t6, s0, t3

label: xxxxxx



Taken branch

Convert to NOP

Convert to NOP

PC updated reflecting branch outcome

Pipelining Conclusion

- Pipelining increases throughput by overlapping execution of multiple instructions
- All pipeline stages have same duration
  - Choose partition that accommodates this constraint
- Hazards potentially limit performance
  - Maximizing performance requires programmer/compiler assistance

Agenda

- Pipelining Review
- Processor Performance - Overview
- Complex Pipelines
- Static Multiple Issues (VLIW)
- Dynamic Multiple Issues (Superscalar)

Increasing Processor Performance

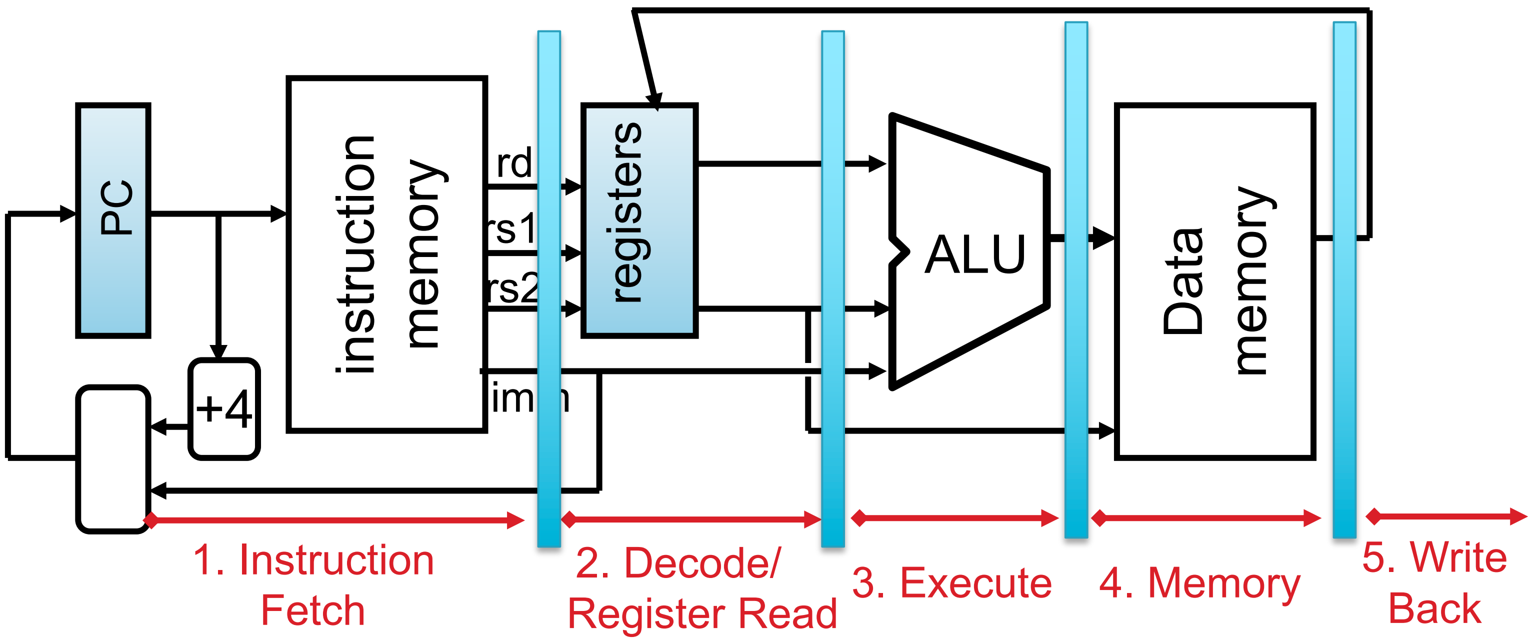
1. Clock rate
   – Limited by technology and power dissipation
2. Pipelining
   – "Overlap" instruction execution
   – Deeper pipeline: 5 => 10 => 15 stages
     - Less work per stage → shorter clock cycle
     - But more potential for hazards
     - Multi-issue "superscalar" processor

to issue:
发出

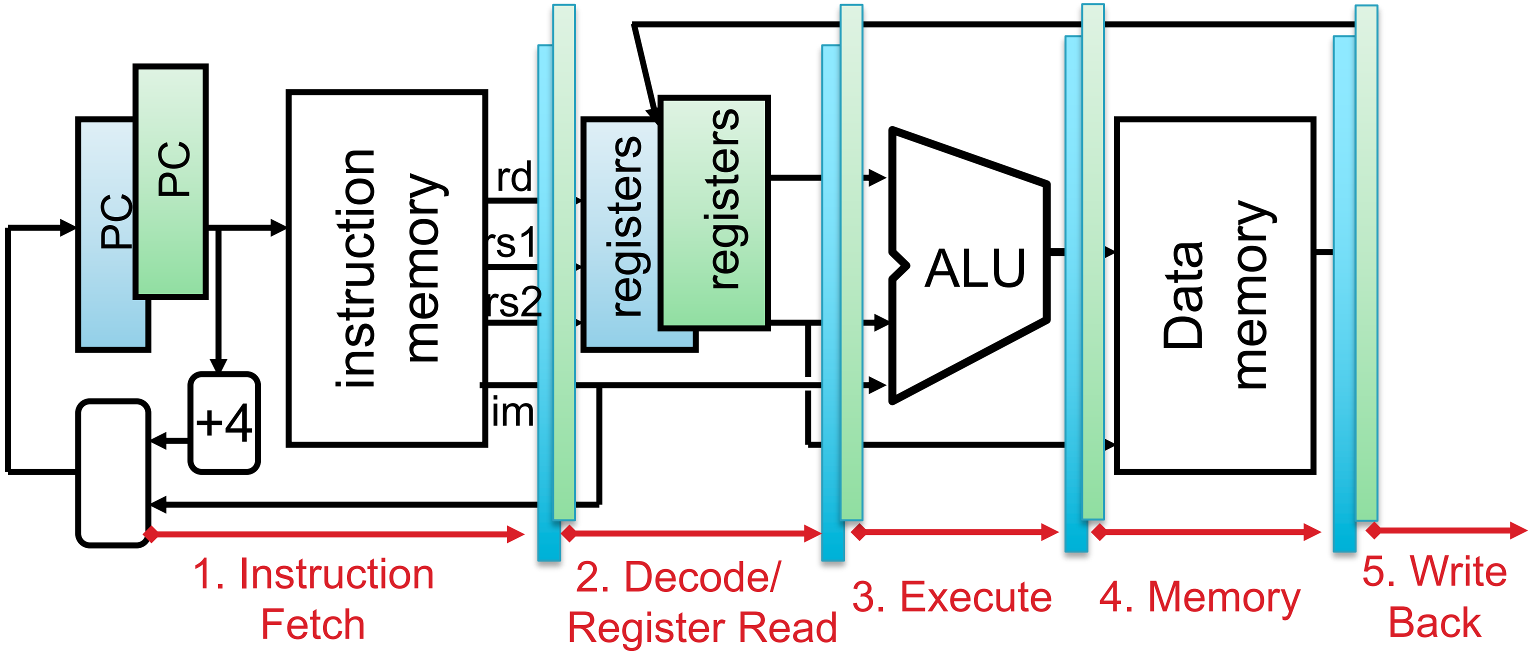
Greater Instruction-Level Parallelism (ILP)

- Multiple issue "superscalar"
  - Replicate pipeline stages => multiple pipelines
  - Start multiple instructions per clock cycle
  - CPI < 1, so use Instructions Per Cycle (IPC)
  - E.g., 4GHz 4-way multiple-issue
    - 16 BIPS, peak CPI = 0.25, peak IPC = 4
  - But dependencies reduce this in practice
- "Out-of-Order" execution
  - Reorder instructions dynamically in hardware to reduce impact of hazards
- Hyper-threading

Pipelined RISC-V RV32I Datapath



Hyper-threading (simplified)

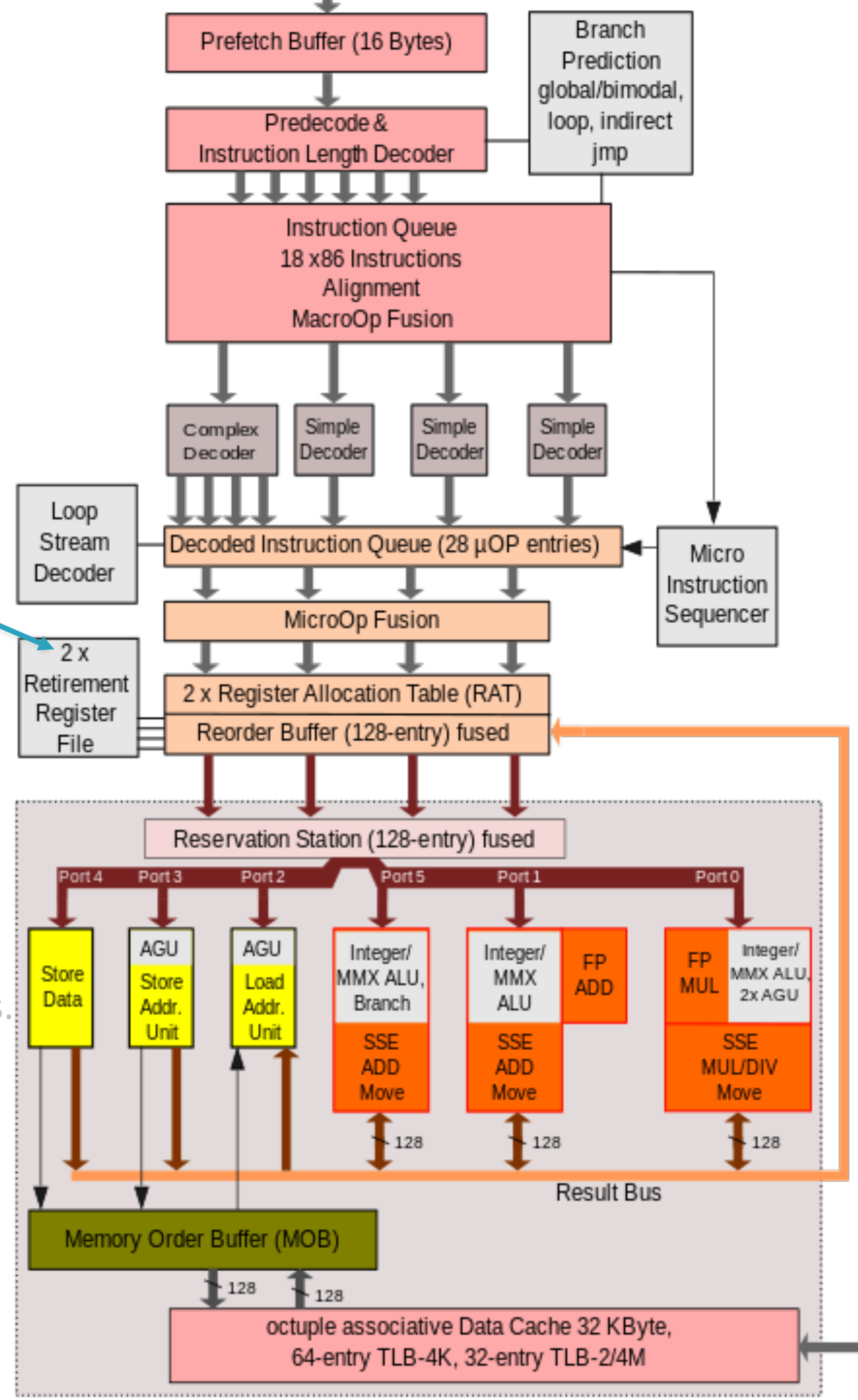


- Duplicate all elements that hold the state (registers)
- Use the same CL blocks
- Use muxes to select which state to use every clock cycle
- => run 2 independent processes
  - No Hazards: registers different; different control flow; memory different; Threads: memory hazard should be solved by software (locking, mutex, …)
- Speedup?
  - No obvious speedup; Complex pipeline: make use of CL blocks in case of unavailable resources (e.g. wait for memory)

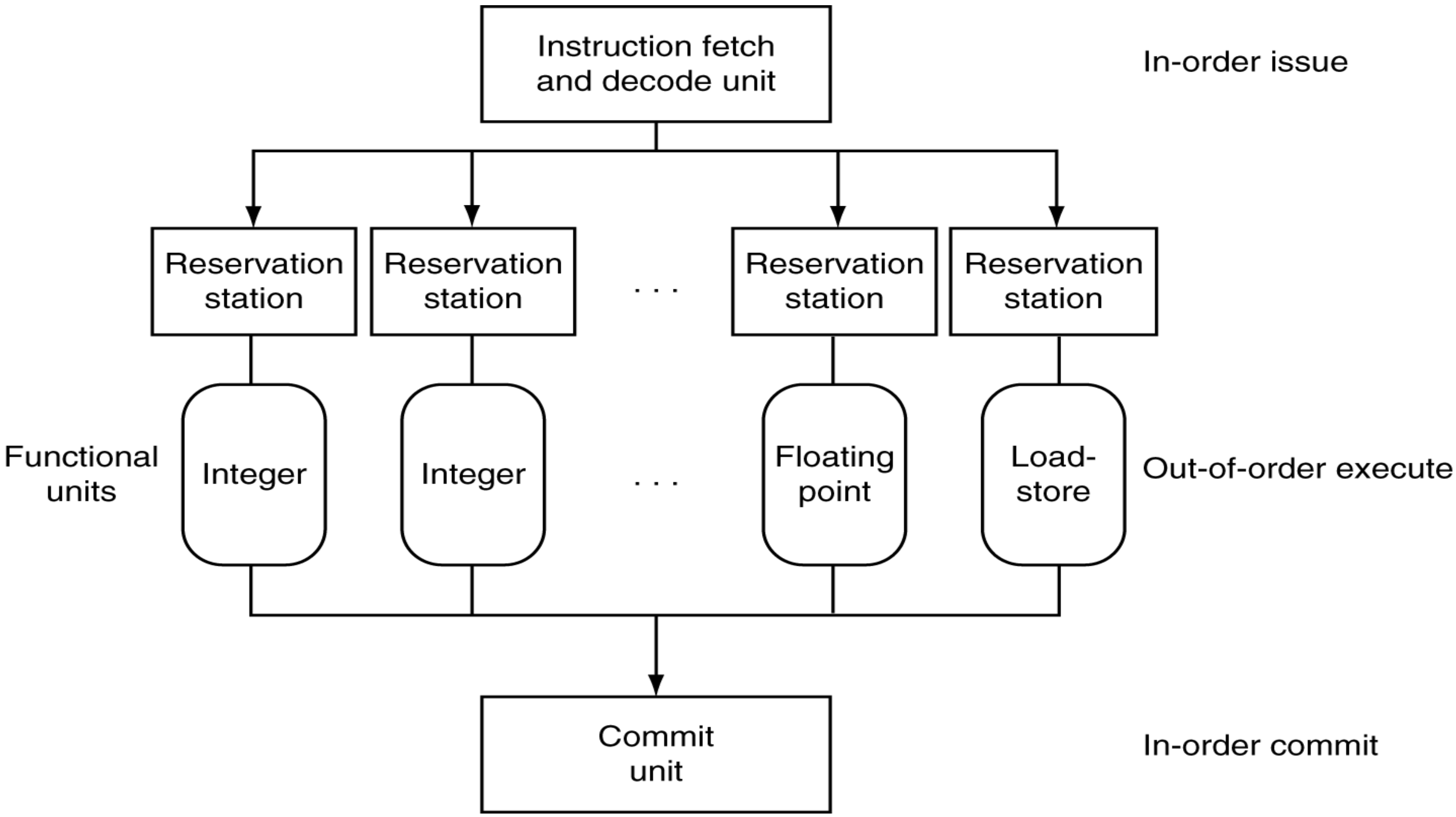
Intel Nehalem i7

(launched 2008)

- Hyperthreading:
  - About 5% die area
  - Up to 30% speed gain (BUT also < 0% possible)
- Pipeline: 20-24 stages!
- Out-of-order execution
  1. Instruction fetch.
  2. Instruction dispatch to an instruction queue
  3. Instruction: Wait in queue until input operands are available => instruction can **leave queue before earlier**, older instructions.
  4. The instruction is issued to the appropriate functional unit and executed by that unit.
  5. The results are queued.
  6. Write to register only after all older instructions have their results written.



Superscalar Processor



Superscalar = Multicore?

https://en.wikipedia.org/wiki/Superscalar_processor

- NO!
- **Superscalar**: More than one Instruction per clock cycle!
  - Computing not a different thread!
  - Computing instructions from the same program!

  => Higher throughput

- In Flynn's taxonomy (later in course):
  - **a single-core superscalar processor is classified as an SISD** processor (Single Instruction stream, Single Data stream)
  - **But**: most superscalar processors support short vector operations => those are then SIMD (Single Instruction stream, Multiple Data streams).
  - And: nowadays most superscalar processors are multicore, too.

"Iron Law" of Processor Performance

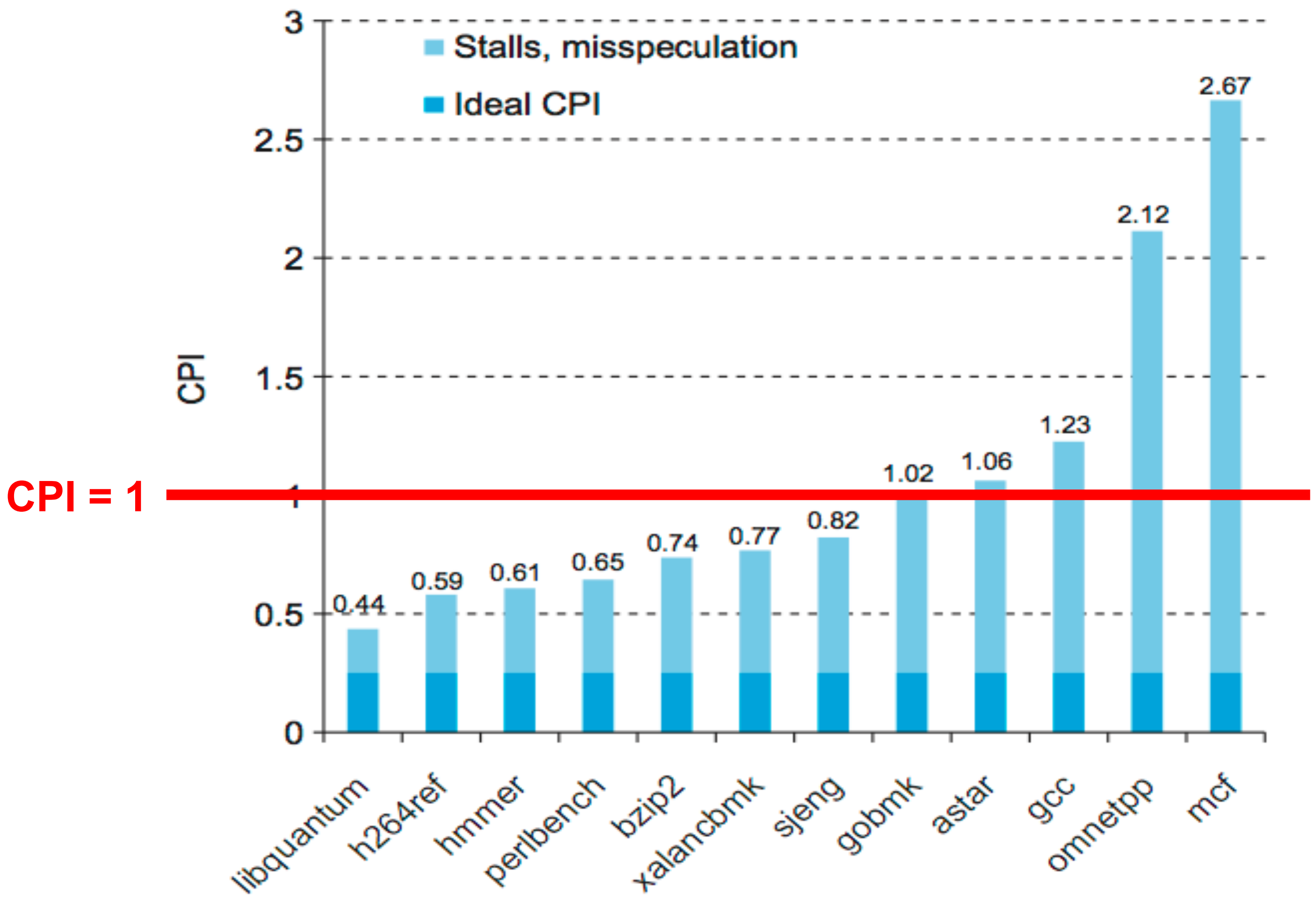
CPI = Cycles Per Instruction

Can time | **Can count** | **Can look up**

$$\frac{\text{Time}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Time}}{\text{Cycle}}$$

$$\textbf{CPI} = \frac{\text{Cycles}}{\text{Instruction}} = \frac{\text{Time}}{\text{Program}} \div \left(\frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Time}}{\text{Cycle}} \right)$$

Benchmark: CPI of Intel Core i7



CPI of Intel Core i7 920 running SPEC2006 integer benchmarks.

Calculating CPI Another Way

- First calculate CPI for each individual instruction (**`add`**, **`sub`**, **`and`**, etc.)
- Next calculate frequency of each individual instruction
- Finally multiply these two for each instruction and add them up to get final CPI (the weighted sum)

Example (RISC processor)

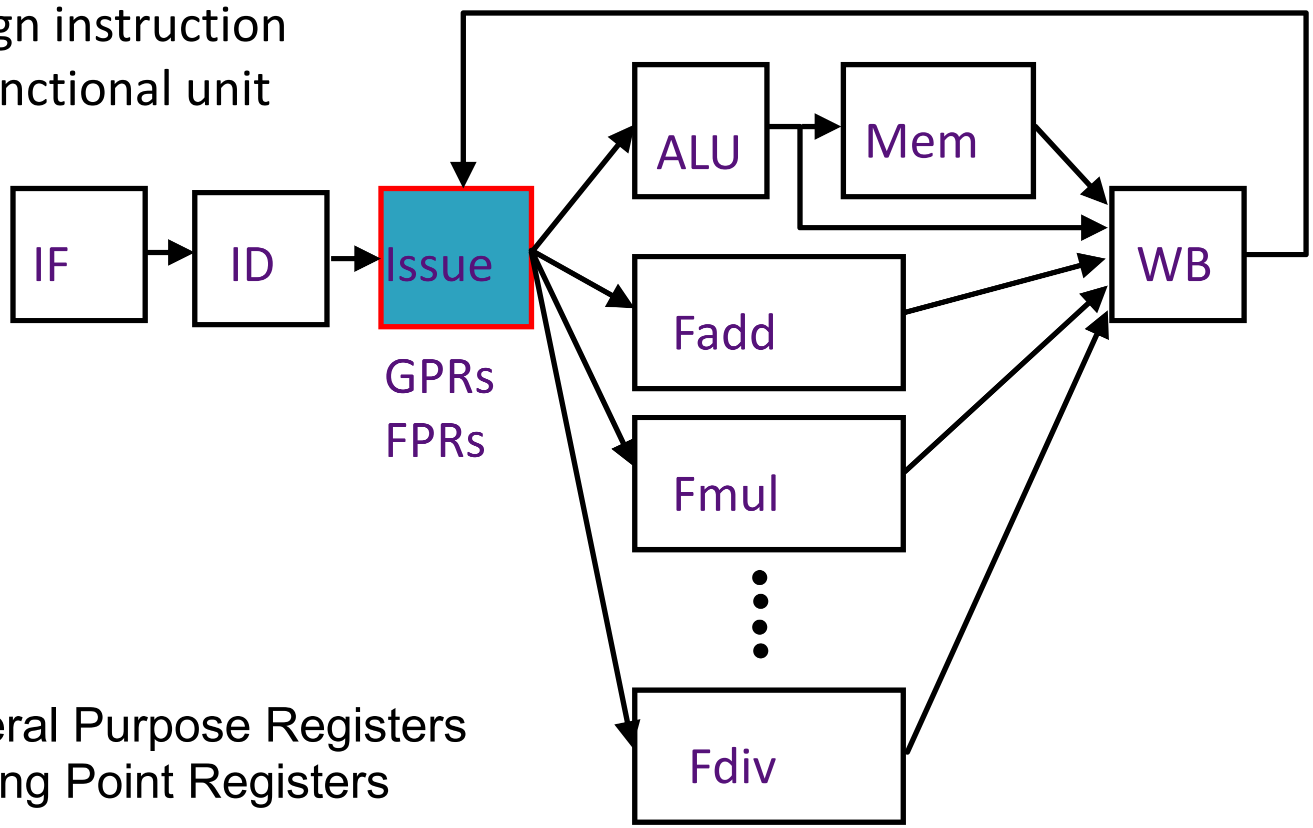
Op	$Freq_i$	CPI_i	Prod	(% Time)
ALU	50%	1	.5	(23%)
Load	20%	5	1.0	(45%)
Store	10%	3	.3	(14%)
Branch	20%	2	.4	(18%)
	Instruction Mix		2.2	(Where time spent)

Agenda

- Pipelining Review
- Processor Performance - Overview
- Complex Pipelines
- Static Multiple Issues (VLIW)
- Dynamic Multiple Issues (Superscalar)

Complex Pipeline

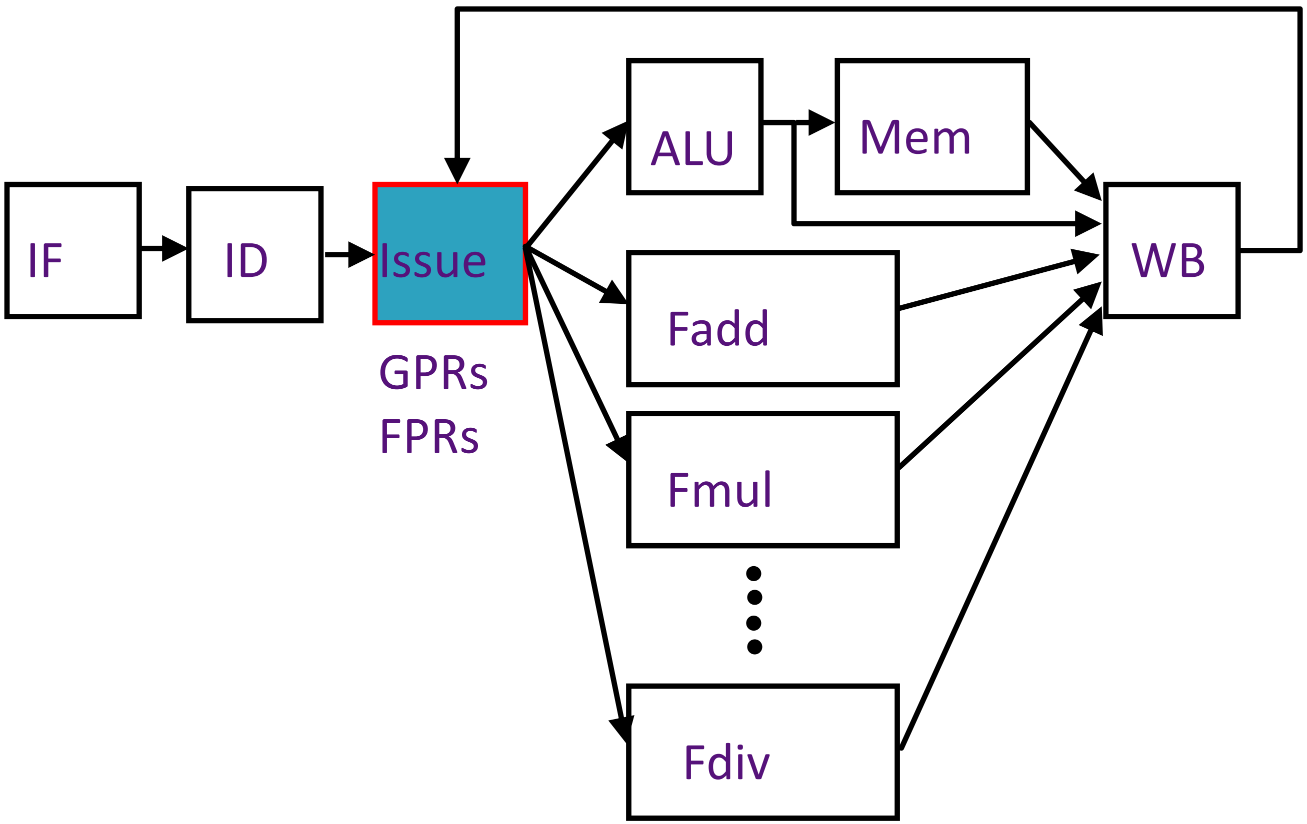
- More than one Functional Unit
- Floating point execution!
  - Fadd & Fmul: fixed number of cycles; > 1
  - Fdiv: unknown number of cycles!
- Memory access: on Cache miss unknown number of cycles
- Issue: Assign instruction
  to functional unit



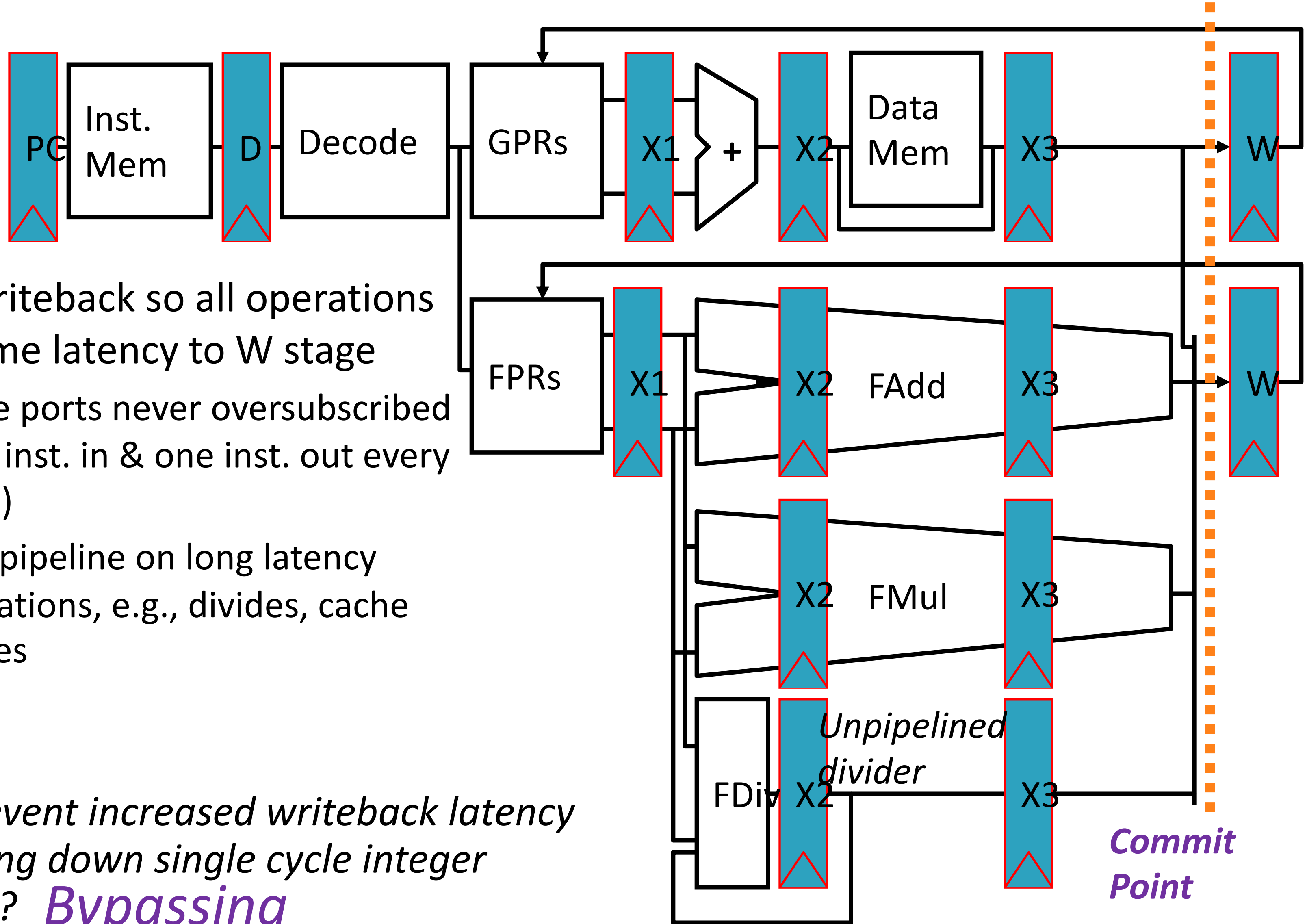
GPRs: General Purpose Registers
FPRs: Floating Point Registers

Issues in Complex Pipeline Control

- Structural conflicts at the execution stage if some FPU or memory unit is not pipelined and takes more than one cycle
- Structural conflicts at the write-back stage due to variable latencies of different functional units
- Out-of-order write hazards due to variable latencies of different functional units



Modern Complex In-Order Pipeline



- Delay writeback so all operations have same latency to W stage
  - Write ports never oversubscribed (one inst. in & one inst. out every cycle)
  - Stall pipeline on long latency operations, e.g., divides, cache misses

How to prevent increased writeback latency from slowing down single cycle integer operations? Bypassing

Agenda

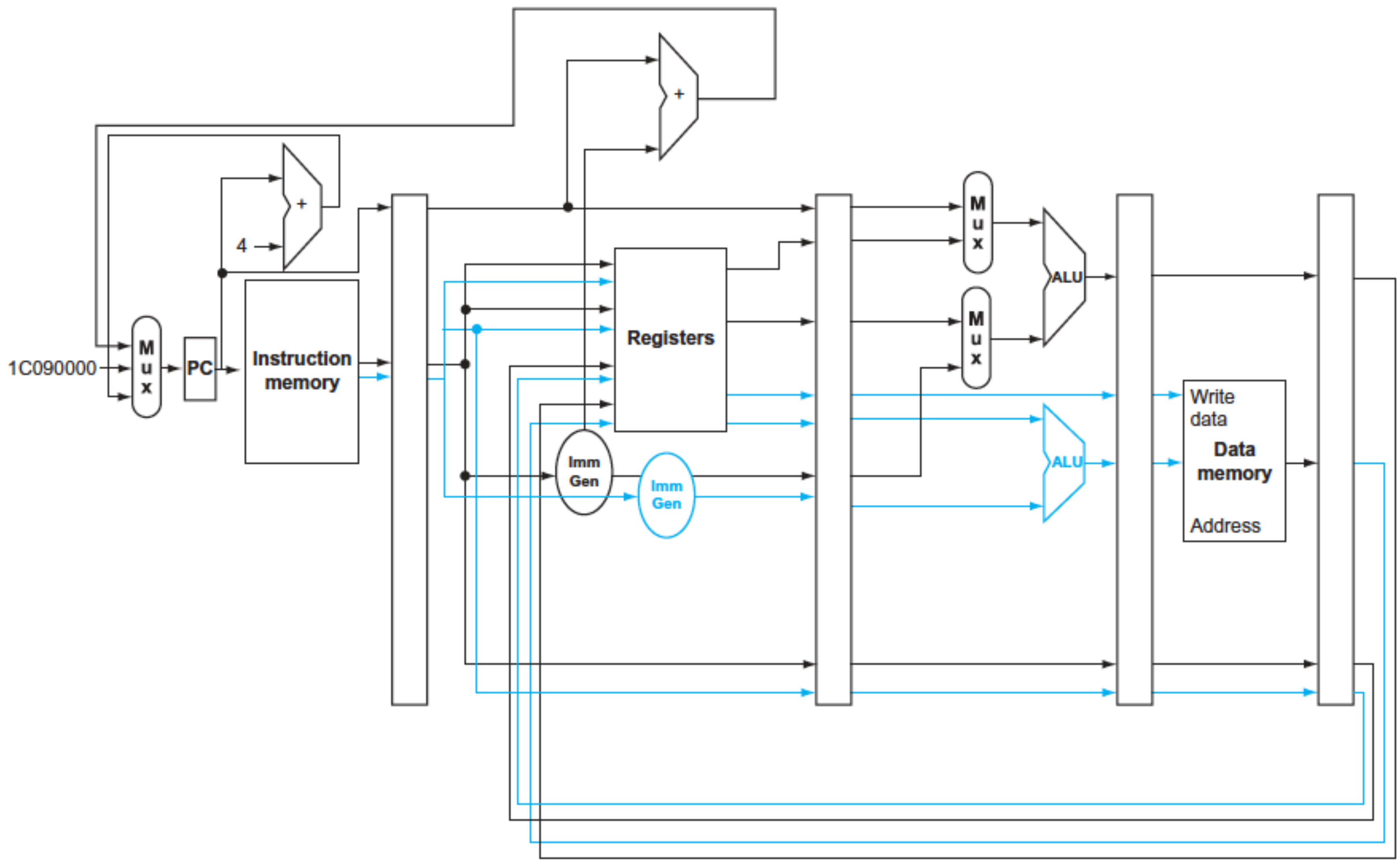
- Pipelining Review
- Processor Performance - Overview
- Complex Pipelines
- Static Multiple Issues (VLIW)
- Dynamic Multiple Issues (Superscalar)

Static Multiple Issue

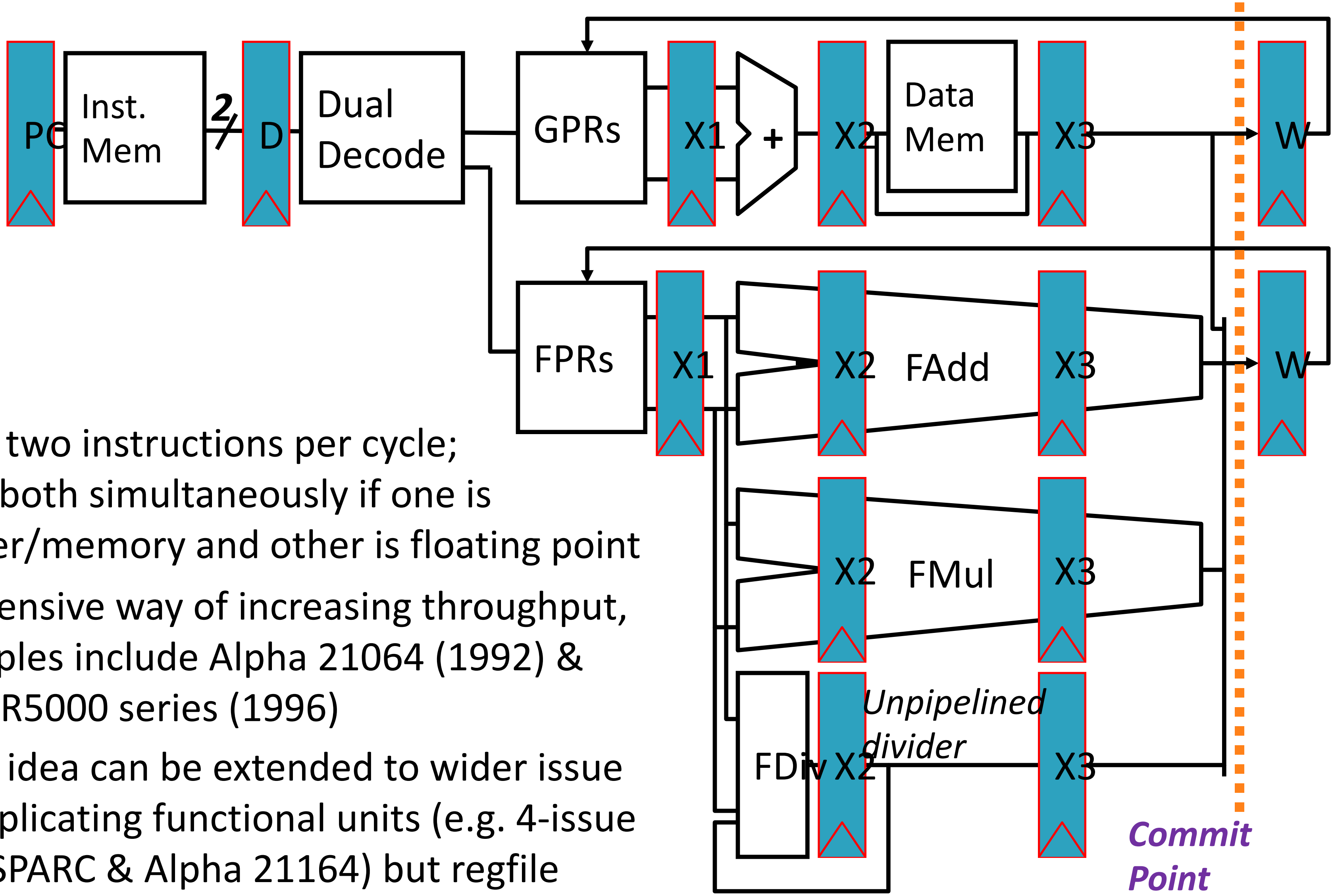
- aka.: Very Long Instruction Word (VLIW)
- Compiler bundles instructions together
- Compiler takes care of hazards
- CPU executes at the same time

Instruction type	Pipe stages							
ALU or branch instruction	IF	ID	EX	MEM	WB			
Load or store instruction	IF	ID	EX	MEM	WB			
ALU or branch instruction		IF	ID	EX	MEM	WB		
Load or store instruction		IF	ID	EX	MEM	WB		
ALU or branch instruction			IF	ID	EX	MEM	WB	
Load or store instruction			IF	ID	EX	MEM	WB	
ALU or branch instruction				IF	ID	EX	MEM	WB
Load or store instruction				IF	ID	EX	MEM	WB

Static Two-Issue RISC-V Datapath



In-Order Superscalar Pipeline



- Fetch two instructions per cycle; issue both simultaneously if one is integer/memory and other is floating point
- Inexpensive way of increasing throughput, examples include Alpha 21064 (1992) & MIPS R5000 series (1996)
- Same idea can be extended to wider issue by duplicating functional units (e.g. 4-issue UltraSPARC & Alpha 21164) but regfile ports and bypassing costs grow quickly

Question

- Which statements that are true?

A. The number of clock cycles a floating point multiplier needs depends on the values of the operands.
B. The number of clock cycles a floating point divider needs depends on the values of the operands.
C. A hyperthreading CPU can execute more than one process/ thread at a given time
D. A superscalar CPU can execute more than one process/ thread at a given time.
E. A multi-core CPU can execute more than one process/ thread at a given time.

Agenda

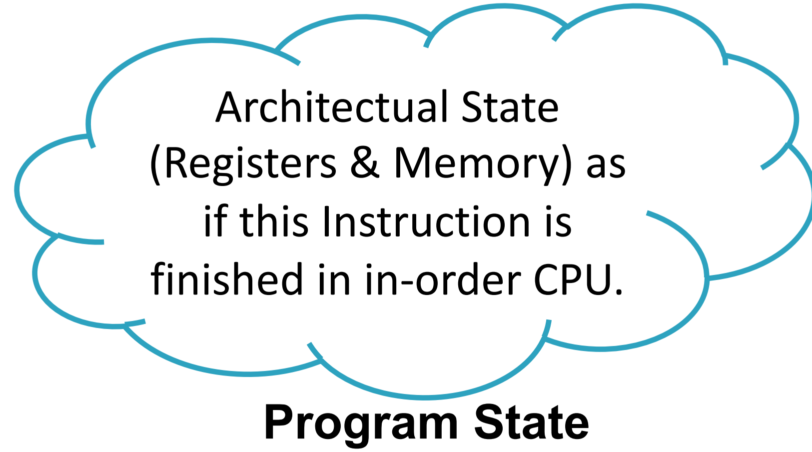
- Processor Performance - Overview
- Complex Pipelines
- Static Multiple Issues (VLIW)
- Dynamic Multiple Issues (Superscalar)

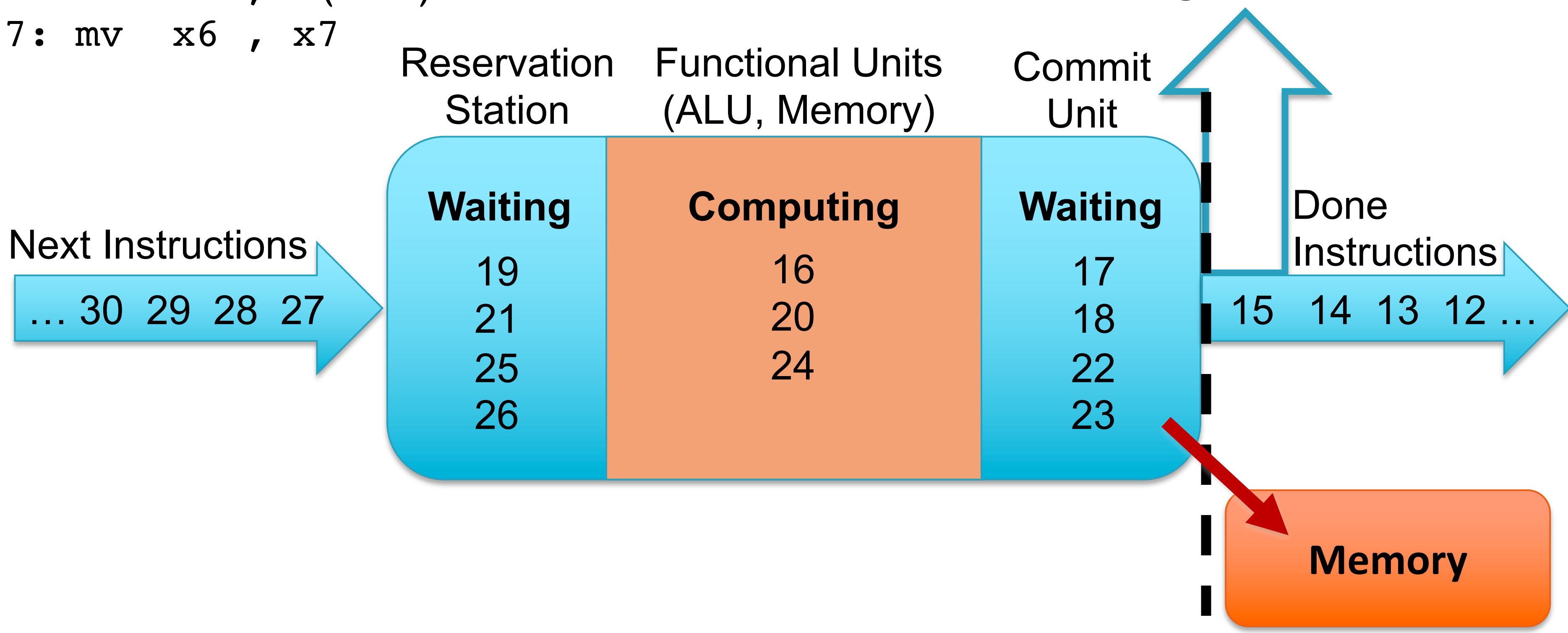
Superscalar: Dynamic Multiple Issue

- Hardware guarantees correct execution =>
  - Compiler does not need to (but can) optimize
- Dynamic pipeline scheduling:
  - Re-order instructions based on:
    - What functional units are free
    - Avoiding of data hazards
  - Reservation Station
    - Buffer of instructions waiting to be executed
    - With operands (Registers) needed
    - Once all operands are available: execute!
  - Commit Unit (Reorder buffer): supply the operands to reservation station; write to register
  - OR: Unified Physical Register File :
    Registers are renamed for use in reservation station and commit unit

Out of Order Issue

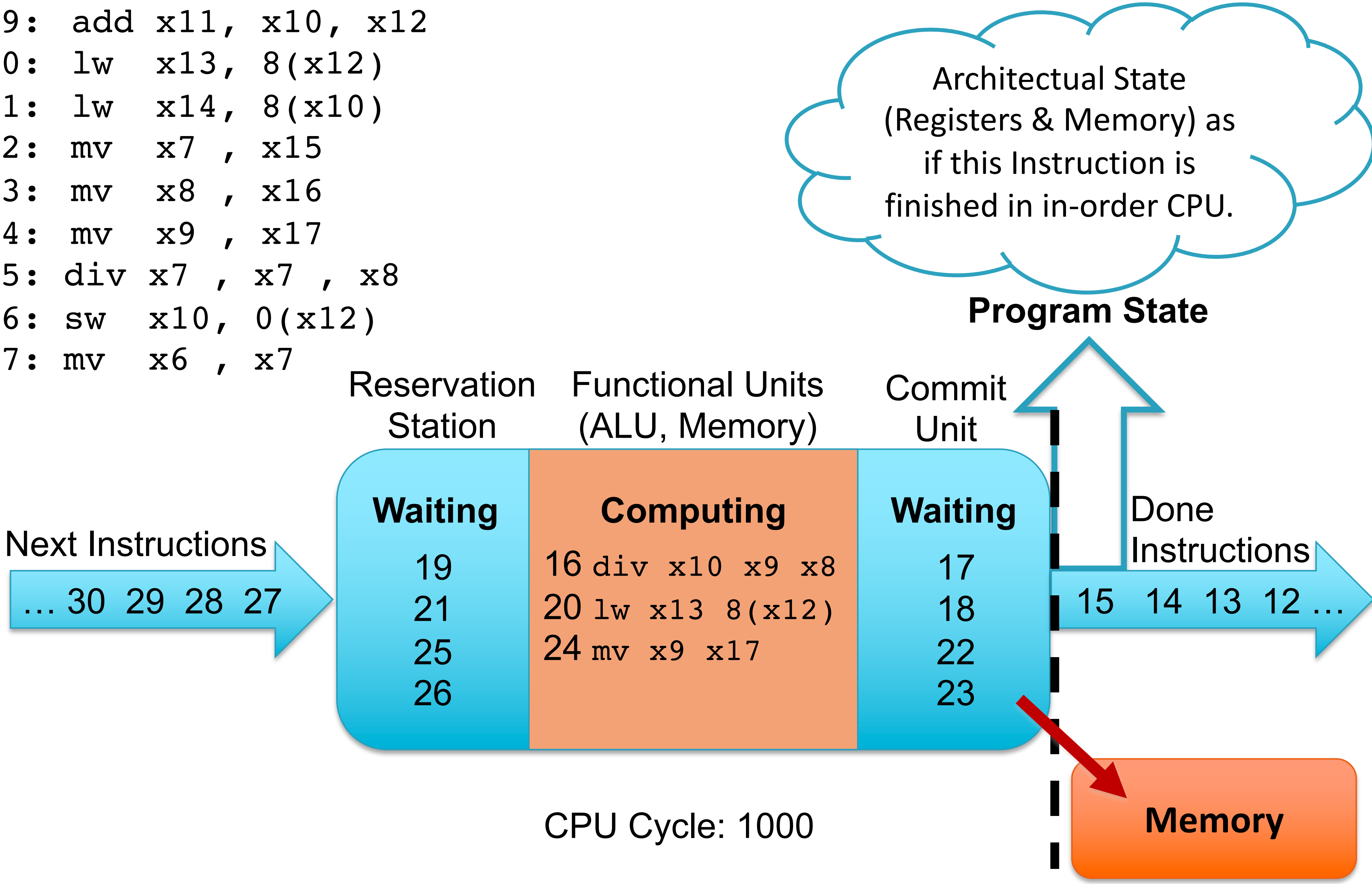
```
15:  add x9 , x9 , x9
16:  div x10, x9 , x8
17:  mv  x12, x6
18:  add x12, x12, x6
19:  add x11, x10, x12
20:  lw  x13, 8(x12)
21:  lw  x14, 8(x10)
22:  mv  x7 , x15
23:  mv  x8 , x16
24:  mv  x9 , x17
25: div x7 , x7 , x8
26: sw  x10, 0(x12)
27: mv  x6 , x7
```





Out of Order Issue

```
15:  add x9 , x9 , x9
16:  div x10, x9 , x8
17:  mv  x12, x6
18:  add x12, x12, x6
19:  add x11, x10, x12
20:  lw  x13, 8(x12)
21:  lw  x14, 8(x10)
22:  mv  x7 , x15
23:  mv  x8 , x16
24:  mv  x9 , x17
25: div x7 , x7 , x8
26: sw  x10, 0(x12)
27: mv  x6 , x7
```



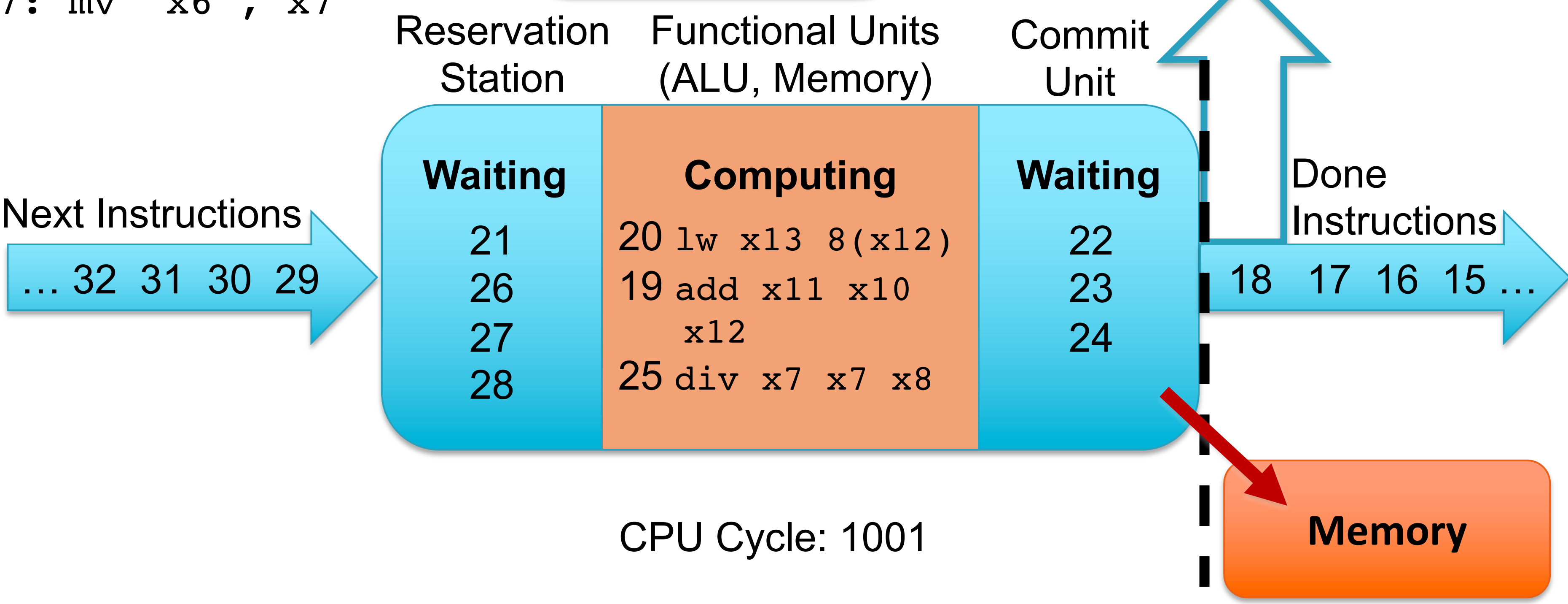
Out of Order Issue

```
15:  add x9 , x9 , x9
16:  div x10, x9 , x8
17:  mv  x12, x6
18:  add x12, x12, x6
19:  add x11, x10, x12
20:  lw  x13, 8(x12)
21:  lw  x14, 8(x10)
22:  mv  x7 , x15
23:  mv  x8 , x16
24:  mv  x9 , x17
25:  div x7 , x7 , x8
26:  sw  x10, 0(x12)
27:  mv  x6 , x7
```

* 16 finished => 16, 17, 18 committed
* 16 computed x10 => 19 can run
* division unit free => 25 can run
* 24 finished
* 27, 28 were fetched

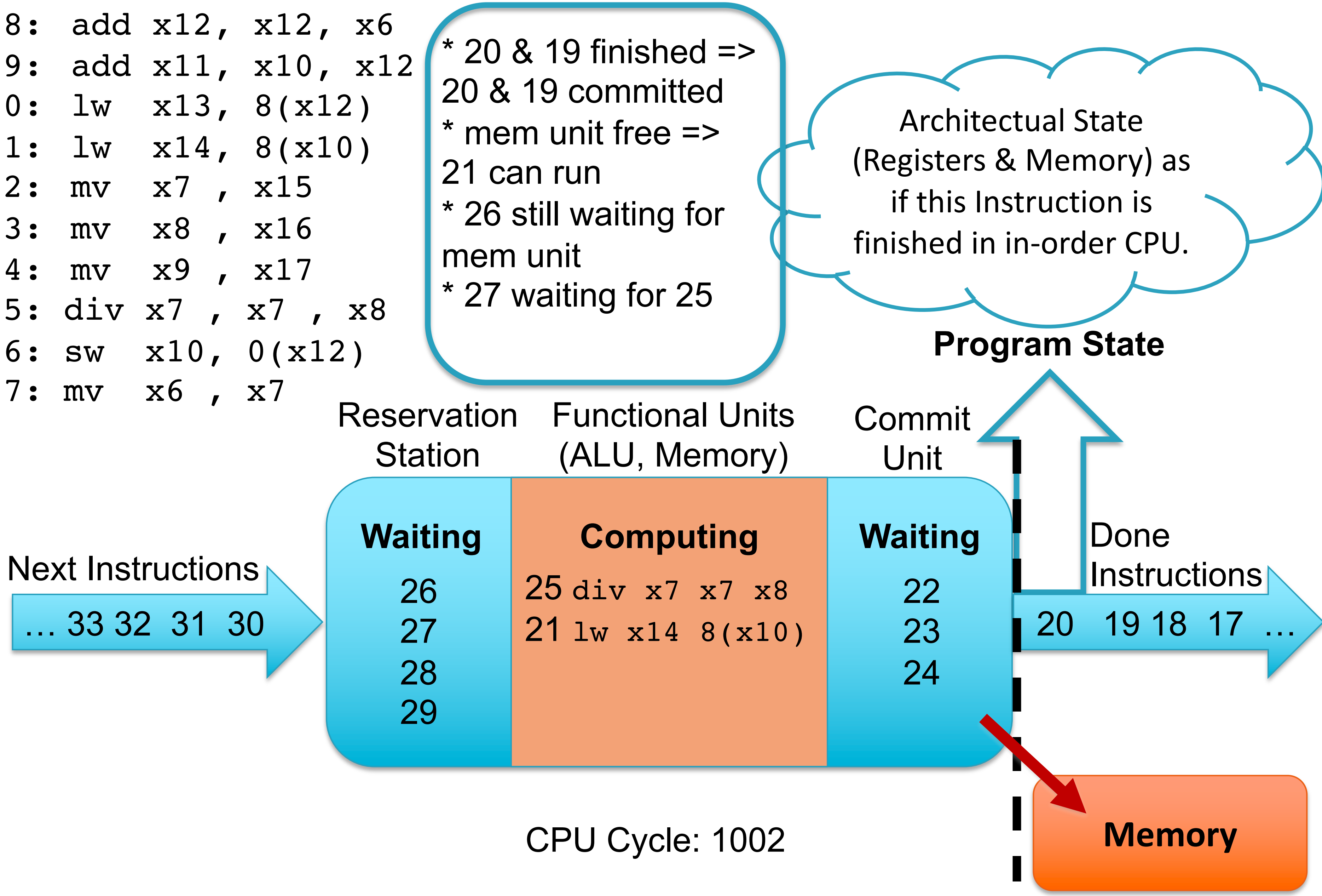
Architectual State (Registers & Memory) as if this Instruction is finished in in-order CPU.

Program State

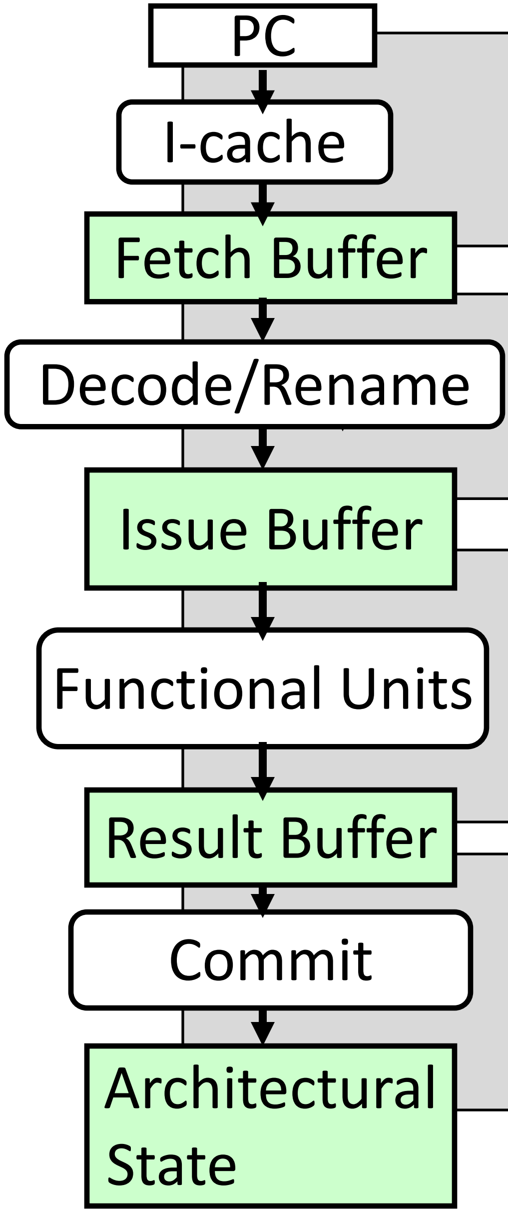


Out of Order Issue

```
15:  add x9 , x9 , x9
16:  div x10, x9 , x8
17:  mv  x12, x6
18:  add x12, x12, x6
19:  add x11, x10, x12
20:  lw  x13, 8(x12)
21:  lw  x14, 8(x10)
22:  mv  x7 , x15
23:  mv  x8 , x16
24:  mv  x9 , x17
25: div x7 , x7 , x8
26: sw  x10, 0(x12)
27: mv  x6 , x7
```



Phases of Instruction Execution



Fetch: *Instruction bits retrieved from instruction cache.*

Decode: *Instructions dispatched to appropriate issue buffer*

Execute: *Instructions and operands issued to functional units. When execution completes, all results and exception flags are available.*

Commit: *Instruction irrevocably updates architectural state (aka "graduation"), or takes precise trap/interrupt.*

Separating Completion from Commit

- Re-order buffer (ROB) holds register results from completion until commit
  - Entries allocated in program order during decode
  - Buffers completed values and exception state until in-order commit point
  - Completed values can be used by dependents before committed (bypassing)
  - Each entry holds program counter, instruction type, destination register specifier and value if any, and exception status (info often compressed to save hardware)

In-Order versus Out-of-Order Phases

- Instruction fetch/decode/rename always in-order
  - Need to parse ISA sequentially to get correct semantics
  - *Proposals for speculative OoO instruction fetch, e.g., Multiscalar. Predict control flow and data dependencies across sequential program segments fetched/decoded/executed in parallel, fixup if prediction wrong*
- Dispatch (place instruction into machine buffers to wait for issue) also always in-order
  - Some use "Dispatch" to mean "Issue"

In-Order Versus Out-of-Order Issue

- In-order (InO) issue:
  - Issue **stalls** on read after write (RAW), dependencies or structural hazards, or possibly write after read (WAR), write after write (WAW) hazards
  - Instruction cannot issue to execution units unless all preceding instructions have issued to execution units
- Out-of-order (OoO) issue:
  - Instructions dispatched in program order to *reservation stations (or other forms of instruction buffer)* to wait for operands to arrive, or other hazards to clear
  - While earlier instructions wait in issue buffers, following instructions can be dispatched and issued out-of-order

In-Order versus Out-of-Order Completion

- All but simplest machines have out-of-order completion, due to different latencies of functional units and desire to bypass values as soon as available
- Classic RISC V-stage integer pipeline just barely has in-order completion
  - Load takes two cycles, but following one-cycle integer op completes at same time, not earlier
  - Adding pipelined FPU immediately brings OoO completion

Superscalar Intel Processors

- Pentium 4: Marketing demanded higher clock rate => deeper pipelines & high power consumption
- Afterwards: Multi-core processors

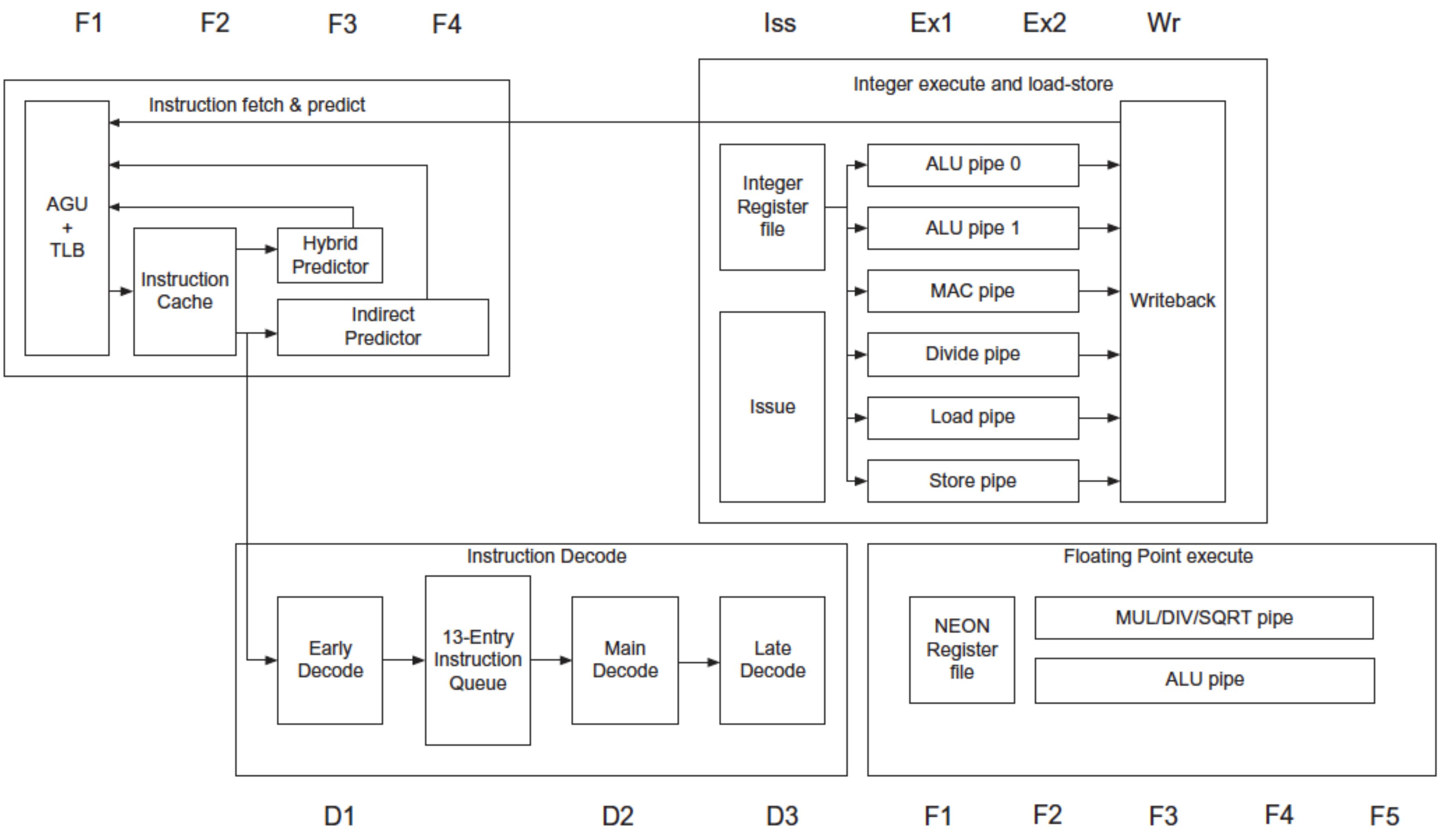
Microprocessor	Year	Clock Rate	Pipeline Stages	Issue Width	Out-of-Order/ Speculation	Cores/ Chip	Power	
Intel 486	1989	25 MHz	5	1	No	1	5	W
Intel Pentium	1993	66 MHz	5	2	No	1	10	W
Intel Pentium Pro	1997	200 MHz	10	3	Yes	1	29	W
Intel Pentium 4 Willamette	2001	2000 MHz	22	3	Yes	1	75	W
Intel Pentium 4 Prescott	2004	3600 MHz	31	3	Yes	1	103	W
Intel Core	2006	2930 MHz	14	4	Yes	2	75	W
Intel Core i5 Nehalem	2010	3300 MHz	14	4	Yes	2–4	87	W
Intel Core i5 Ivy Bridge	2012	3400 MHz	14	4	Yes	8	77	W

Arm Cortex A53 & Intel Core i7 920

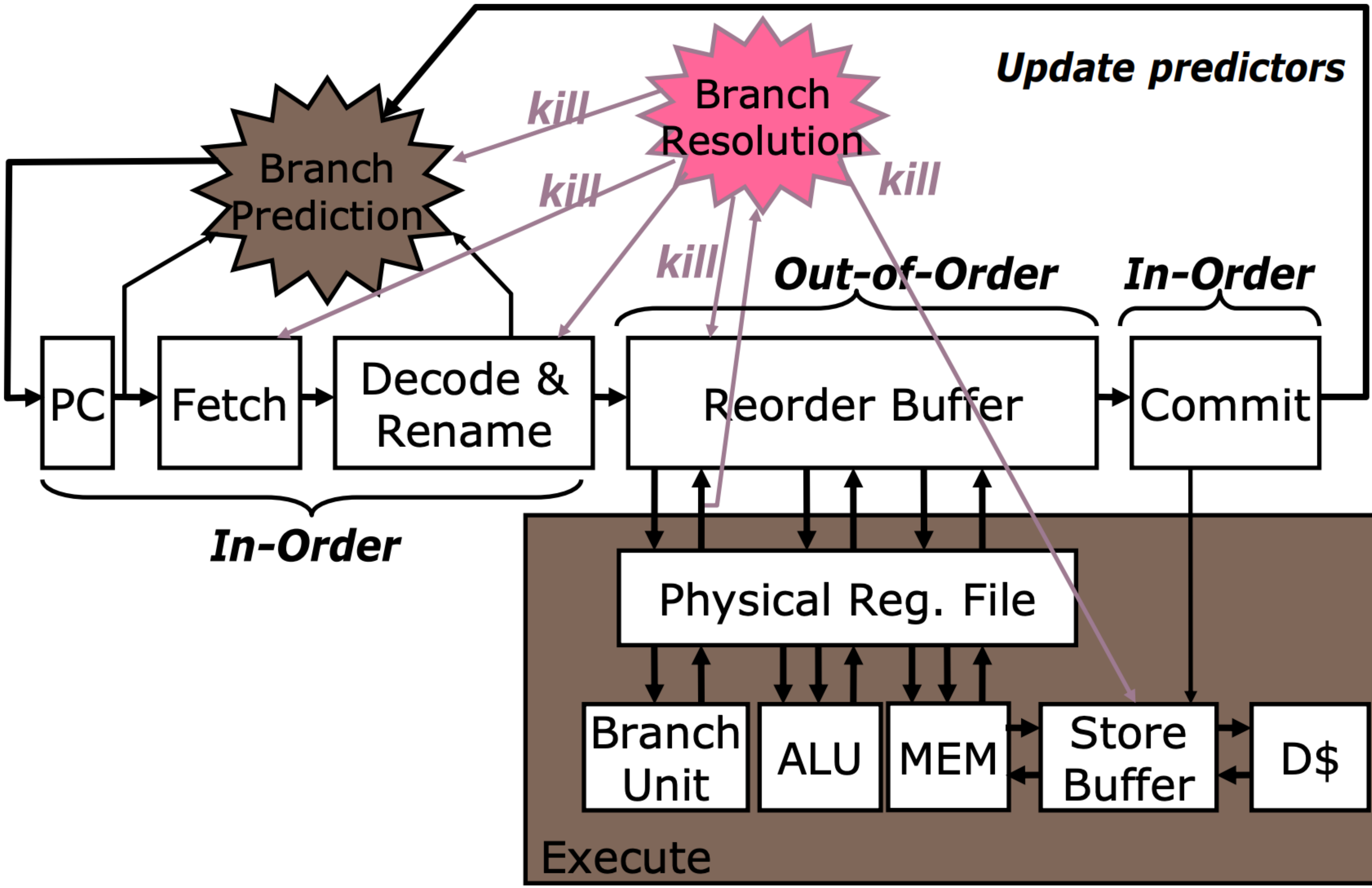
Processor	ARM A53	Intel Core i7 920
Market	Personal Mobile Device	Server, Cloud
Thermal design power	100 milliWatts (1 core @ 1 GHz)	130 Watts
Clock rate	1.5 GHz	2.66 GHz
Cores/Chip	4 (configurable)	4
Floating point?	Yes	Yes
Multiple Issue?	Dynamic	Dynamic
Peak instructions/clock cycle	2	4
Pipeline Stages	8	14
Pipeline schedule	Static In-order	Dynamic Out-of-order with Speculation
Branch prediction	Hybrid	2-level
1st level caches/core	16-64 KiB I, 16-64 KiB D	32 KiB I, 32 KiB D
2nd level cache/core	128–2048 KiB (shared)	256 KiB (per core)
3rd level cache (shared)	(platform dependent)	2–8 MiB

ARM Cortex A53 Pipeline

- Prediction 1 clock cycle! Predict: branches, future function returns; 8 clock cycles on mis-prediction (flush pipeline)

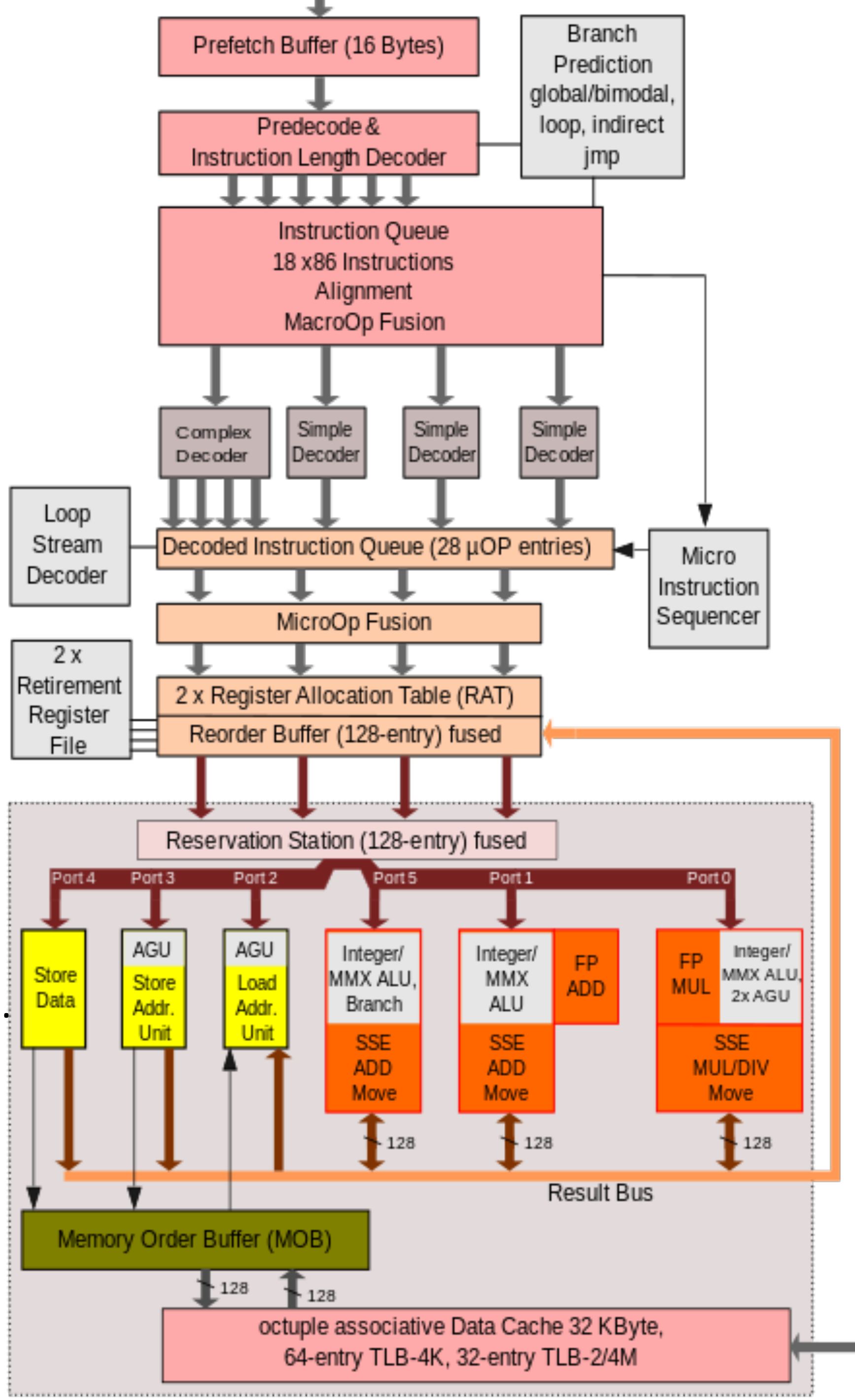


Speculative & Out-of-Order Execution

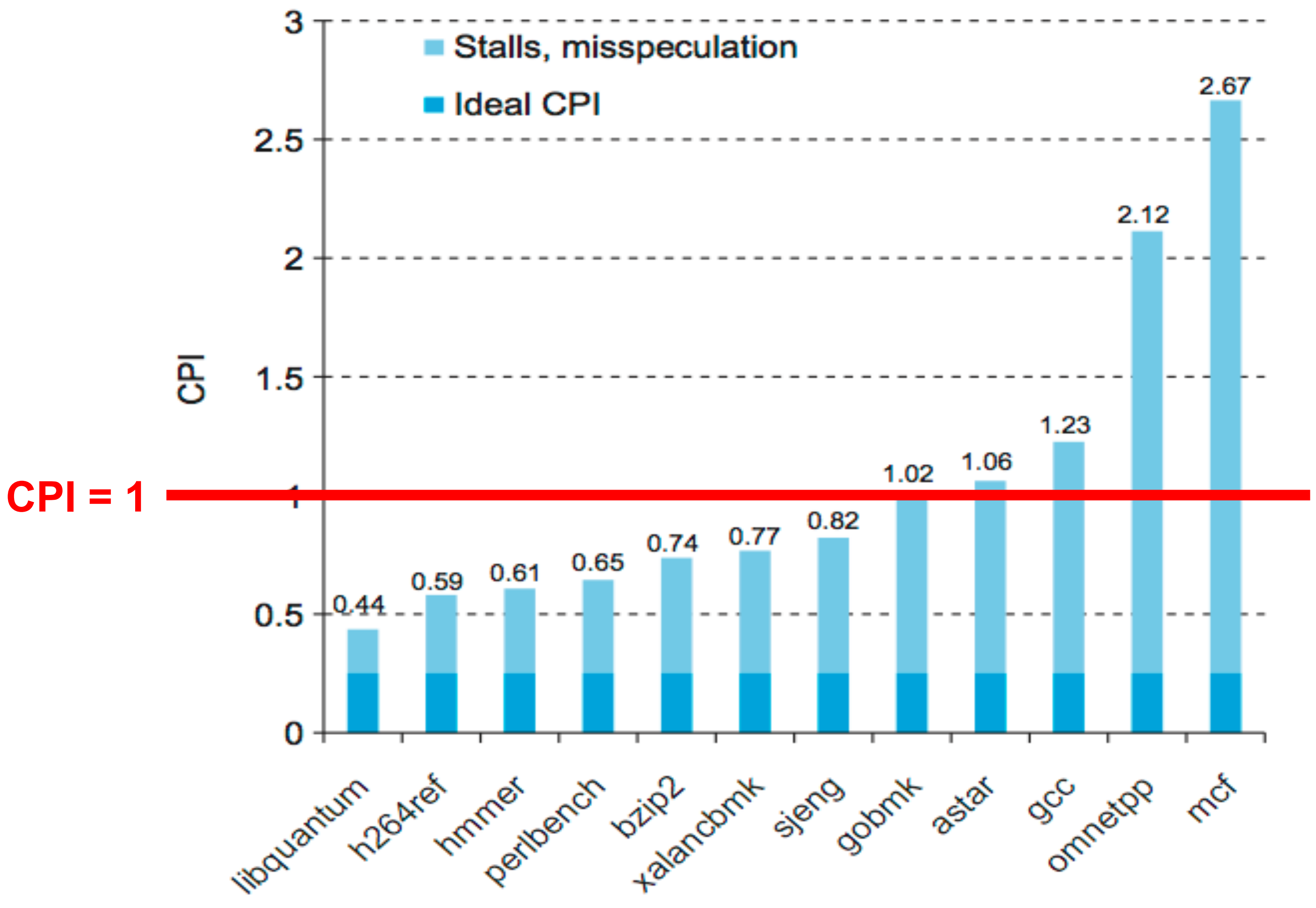


Intel Nehalem i7

- Hyperthreading:
  - About 5% die area
  - Up to 30% speed gain (BUT also < 0% possible)
- Pipeline: 20-24 stages!
- Out-of-order execution
  1. Instruction fetch.
  2. Instruction dispatch to an instruction queue
  3. Instruction: Wait in queue until input operands are available => instruction can **leave queue before earlier**, older instructions.
  4. The instruction is issued to the appropriate functional unit and executed by that unit.
  5. The results are queued.
  6. Write to register only after all older instructions have their results written.



Benchmark: CPI of Intel Core i7



CPI of Intel Core i7 920 running SPEC2006 integer benchmarks.

"Data-in-ROB" Design

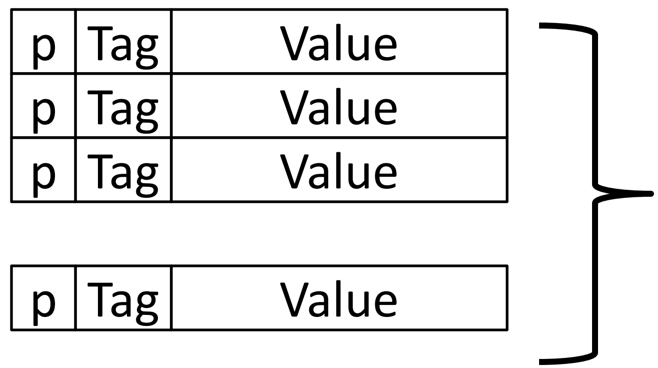
(HP PA8000, Pentium Pro, Core2Duo, Nehalem)

	v	i	Opcode	p	Tag	Src1	p	Tag	Src2	p	Reg	Result	Except?
Oldest →	v	i	Opcode	p	Tag	Src1	p	Tag	Src2	p	Reg	Result	Except?
	v	i	Opcode	p	Tag	Src1	p	Tag	Src2	p	Reg	Result	Except?
Free →	v	i	Opcode	p	Tag	Src1	p	Tag	Src2	p	Reg	Result	Except?
	v	i	Opcode	p	Tag	Src1	p	Tag	Src2	p	Reg	Result	Except?

- Managed as circular buffer in program order, new instructions dispatched to free slots, oldest instruction committed/reclaimed when done ("p" bit set on result)
- Tag is given by index in ROB (Free pointer value)
- In dispatch, non-busy source operands read from architectural register file and copied to Src1 and Src2 with presence bit "p" set. Busy operands copy tag of producer and clear "p" bit.
- Set valid bit "v" on dispatch, set issued bit "i" on issue
- On completion, search source tags, set "p" bit and copy data into src on tag match. Write result and exception flags to ROB.
- On commit, check exception status, and copy result into architectural register file if no trap.

Managing Rename for Data-in-ROB

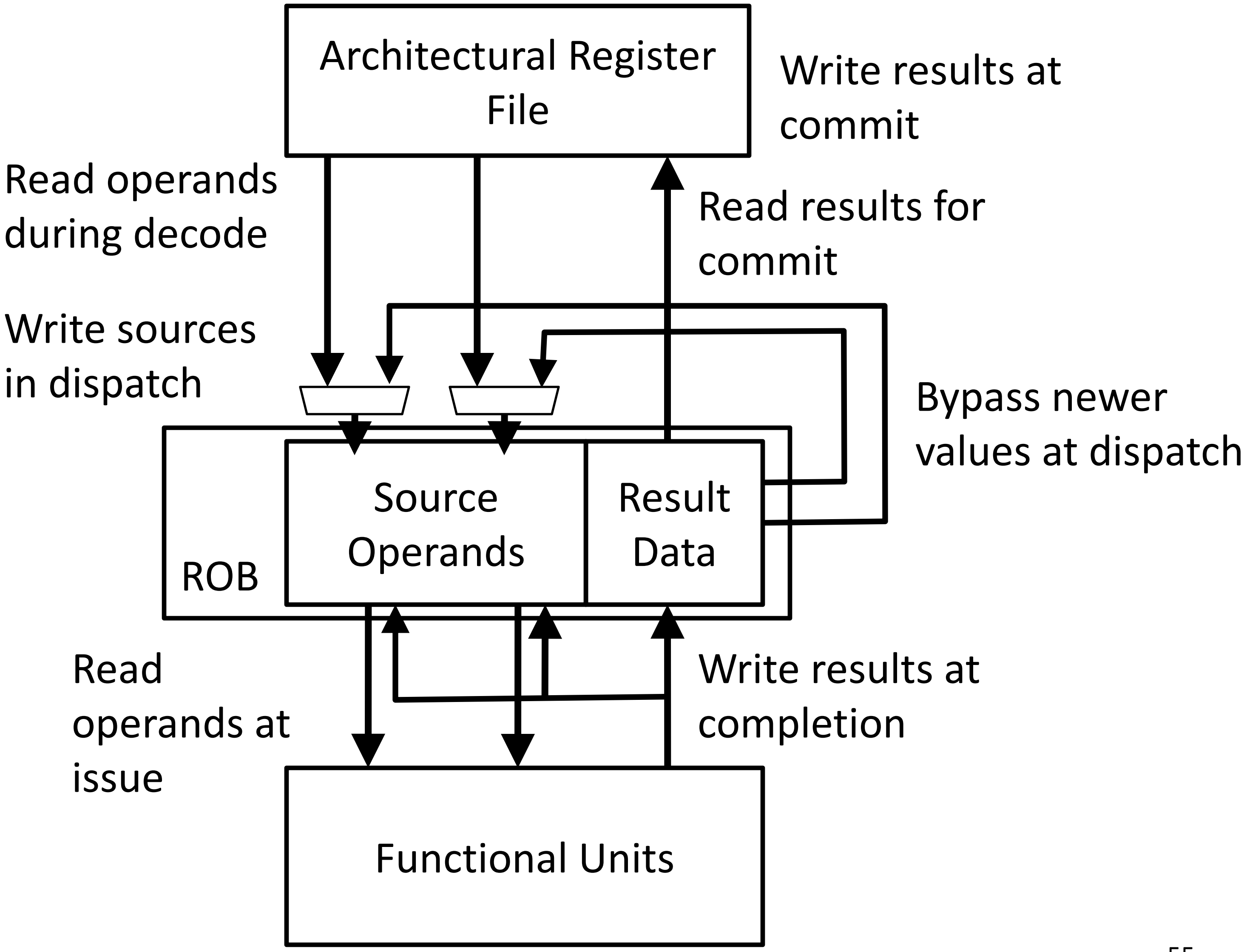
Rename table associated with architectural registers, managed in decode/dispatch



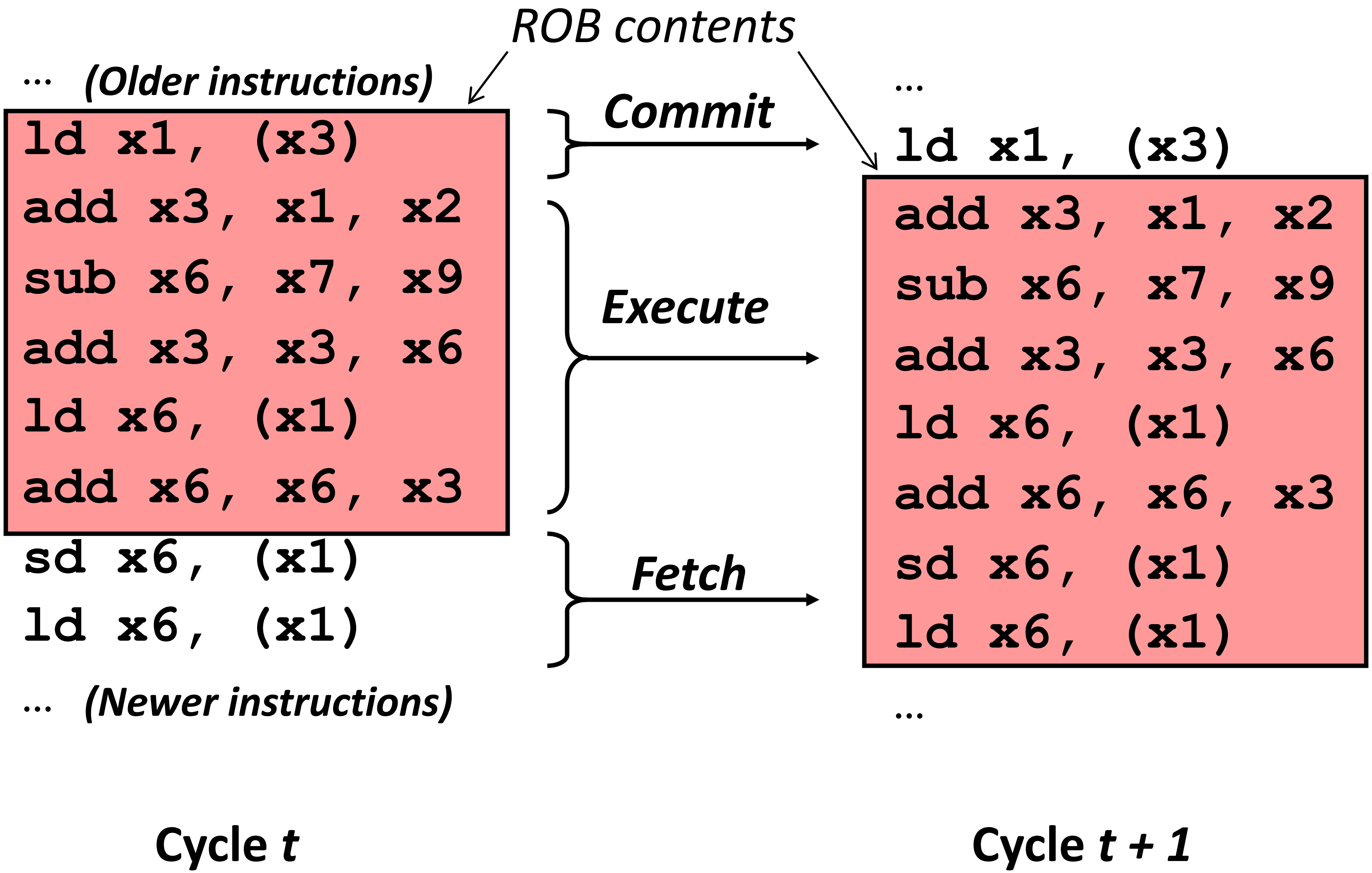
One entry per architectural register

- If “p” bit set, then use value in architectural register file
- Else, tag field indicates instruction that will/has produced value
- For dispatch, read source operands <p,tag,value> from arch. regfile, then also read <p,result> from producing instruction in ROB at tag index, bypassing as needed. Copy operands to ROB.
- Write destination arch. register entry with <0,Free,_>, to assign tag to ROB index of this instruction
- On commit, update arch. regfile with <1, _, Result>
- On trap, reset table (All p=1)

Data Movement in Data-in-ROB Design



Reorder Buffer Holds Active Instructions (Decoded but not Committed)



Register Renaming

- Programmers/ Compilers (have to) re-use registers for different, unrelated purposes
- Idea: Re-name on the fly to resolve (fake) dependencies (anti-dependency)
- Additional benefit: CPU can have more physical registers than ISA!
  - Alpha 21264 CPU has 80 integer register; ISA only 32

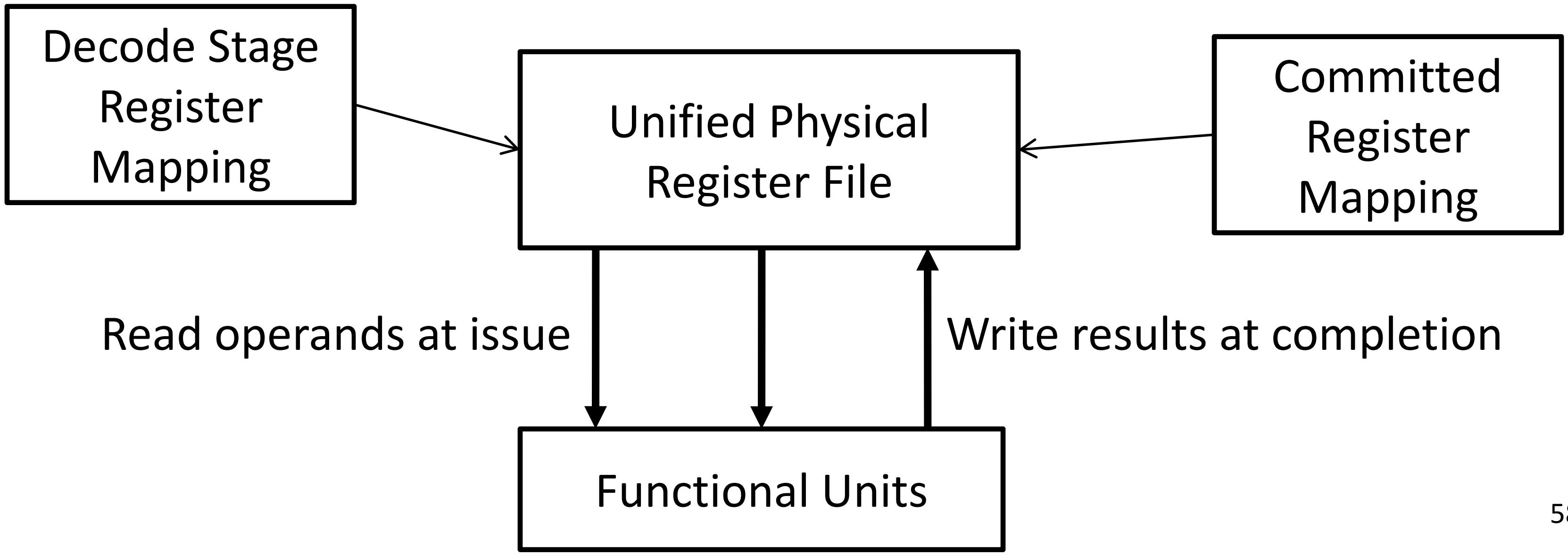
```
r1 ≔ m[1024]
r1 ≔ r1 + 2
m[1032] ≔ r1
r1 ≔ m[2048]
r1 ≔ r1 + 4
m[2056] ≔ r1
```

```
r1 ≔ m[1024]
r1 ≔ r1 + 2
m[1032] ≔ r1
r2 ≔ m[2048]
r2 ≔ r2 + 4
m[2056] ≔ r2
```

Alternative to "Data-in-ROB": Unified Physical Register File

(MIPS R10K, Alpha 21264, Intel Pentium 4 & Sandy/Ivy Bridge)

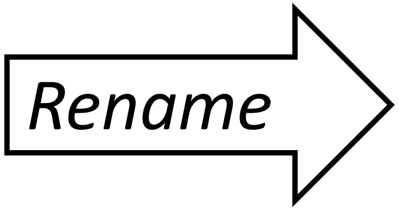
- Rename all architectural registers into a single *physical* register file during decode, no register values read
- Functional units read and write from single unified register file holding committed and temporary registers in execute
- Commit only updates mapping of architectural register to physical register, no data movement



Lifetime of Physical Registers

- Physical regfile holds committed and speculative values
- Physical registers decoupled from ROB entries *(no data in ROB)*

```
ld x1, (x3)
addi x3, x1, #4
sub x6, x7, x9
add x3, x3, x6
ld x6, (x1)
add x6, x6, x3
sd x6, (x1)
ld x6, (x11)
```




```
ld P1, (Px)
addi P2, P1, #4
sub P3, Py, Pz
add P4, P2, P3
ld P5, (P1)
add P6, P5, P4
sd P6, (P1)
ld P7, (Pw)
```

When can we reuse a physical register?

When next writer of same architectural register commits

Conclusion

- “Iron Law” of Processor Performance to estimate speed
- Complex Pipelines: more in CA II
    - Multiple Functional Units => Parallel execution
  - Static Multiple Issues (VLIW)
    - E.g. 2 instructions per cycle
  - Dynamic Multiple Issues (Superscalar)
    - Re-order instructions
    - Issue Buffer; Re-order Buffer; Commit Unit
    - Re-naming of registeres