CS 110
Computer Architecture
Lecture 15:
Performance

Instructors:
Sören Schwertfeger & Chundong Wang

https://robotics.shanghaitech.edu.cn/courses/ca/20s/

School of Information Science and Technology SIST

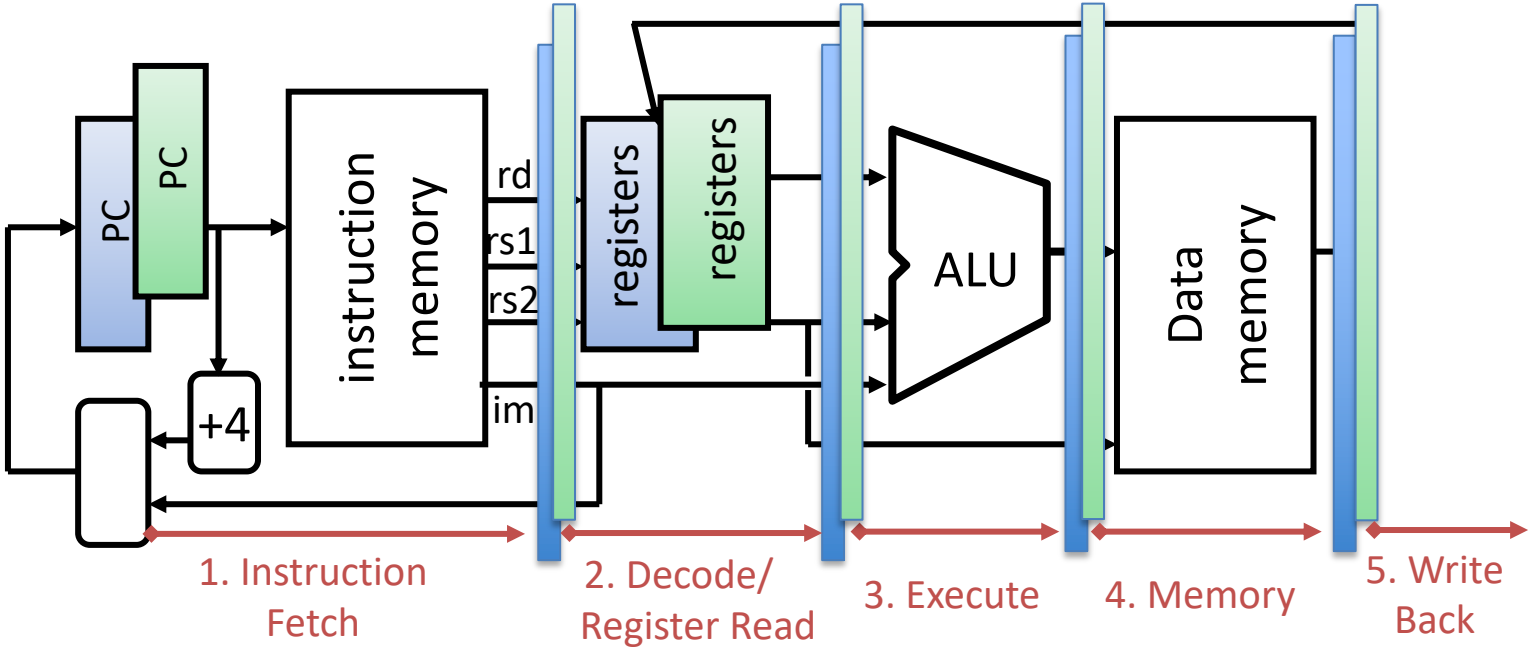
ShanghaiTech University

Slides based on UC Berkley's CS61C

Greater Instruction-Level Parallelism (ILP)

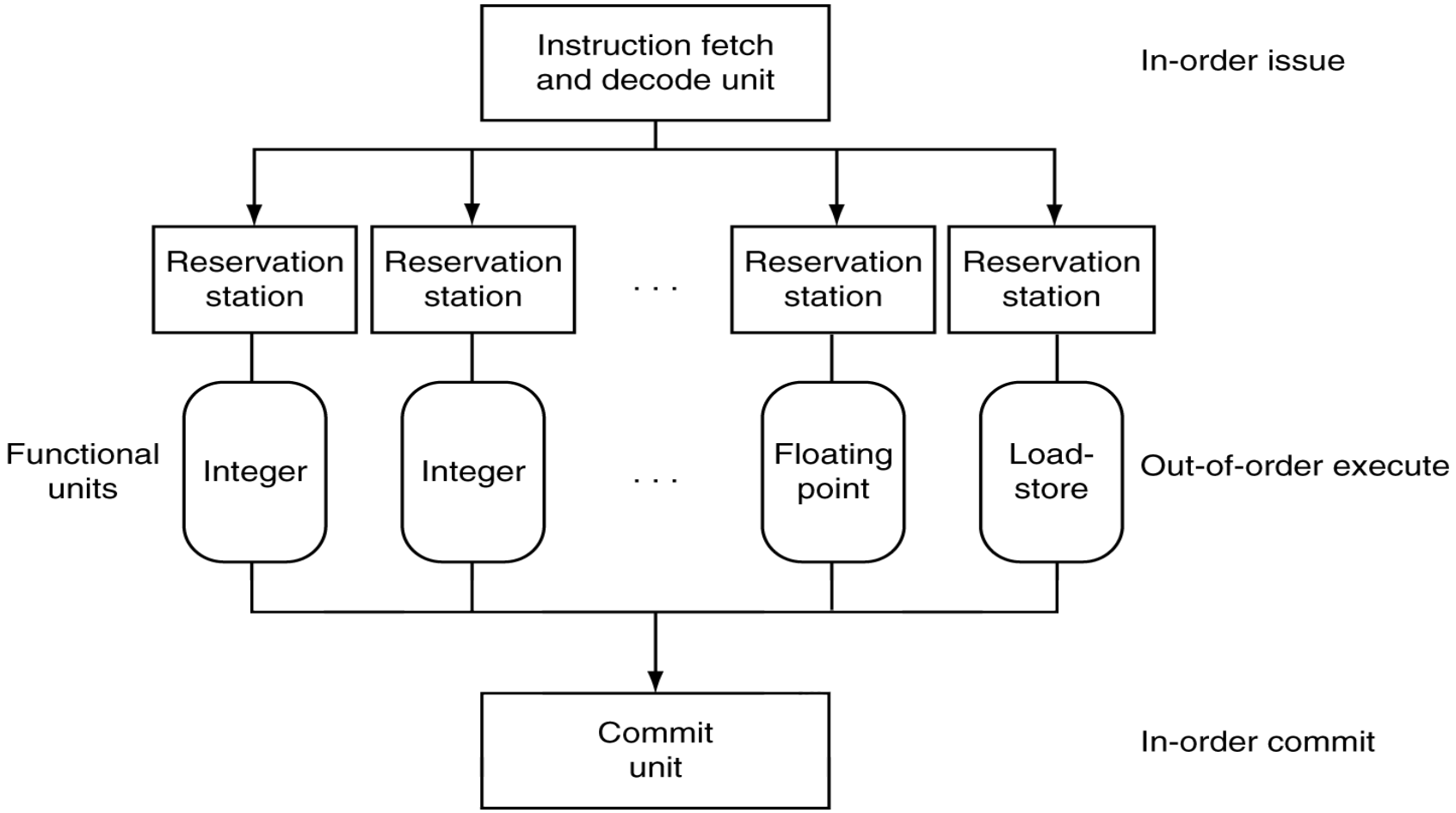
- Multiple issue "superscalar"
  - Replicate pipeline stages => multiple pipelines
  - Start multiple instructions per clock cycle
  - CPI < 1, so use Instructions Per Cycle (IPC)
  - E.g., 4GHz 4-way multiple-issue
    - 16 BIPS, peak CPI = 0.25, peak IPC = 4
  - But dependencies reduce this in practice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

Hyper-threading (simplified)

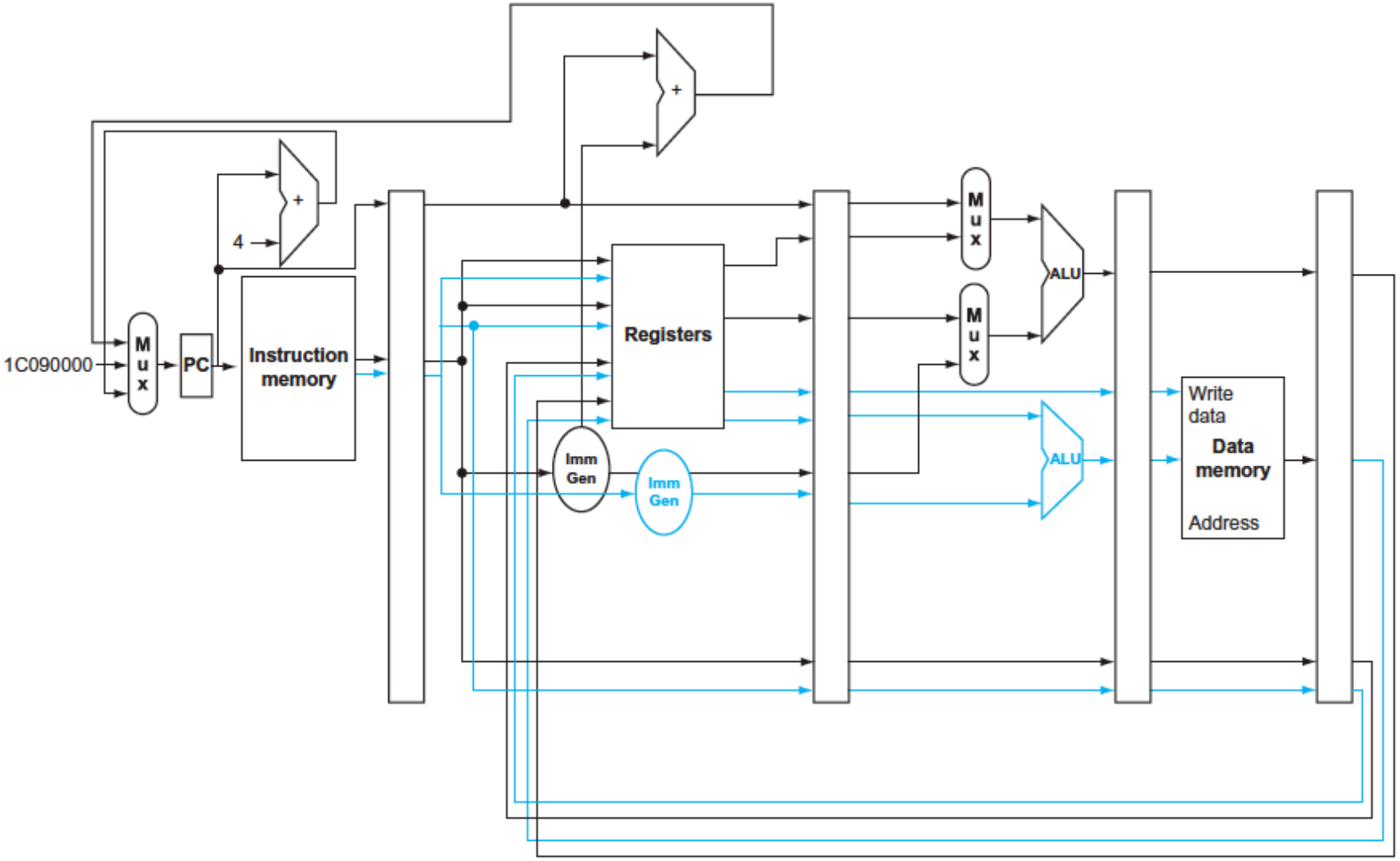


- Duplicate all elements that hold the state (registers)
- Use the same CL blocks
- Use muxes to select which state to use every clock cycle
- => run 2 independent processes
  - No Hazards: registers different; different control flow; memory different; Threads: memory hazard should be solved by software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

Superscalar Processor



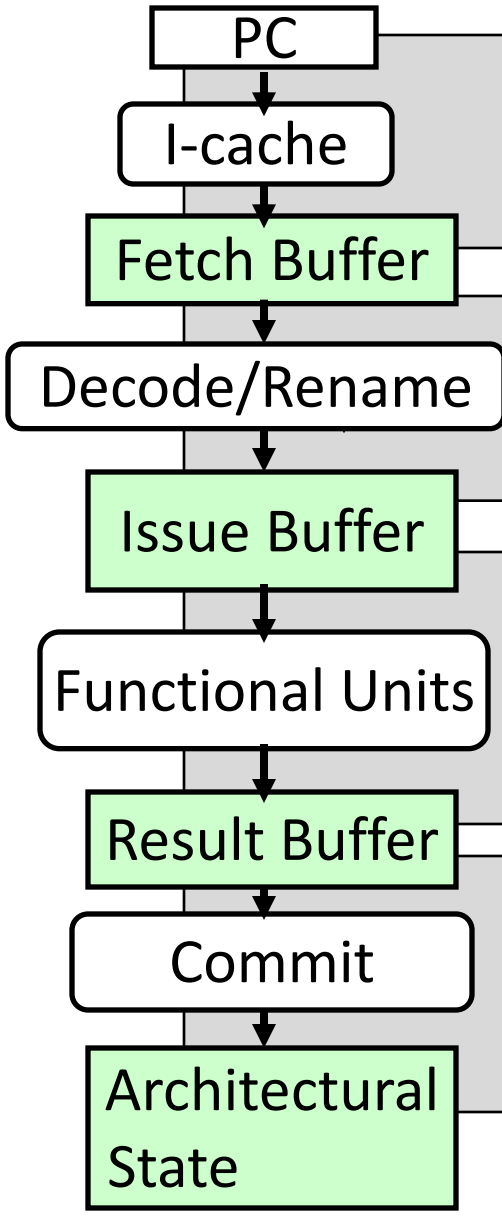
Static Two-Issue RISC-V Datapath



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

Phases of Instruction Execution



Fetch: *Instruction bits retrieved from instruction cache.*

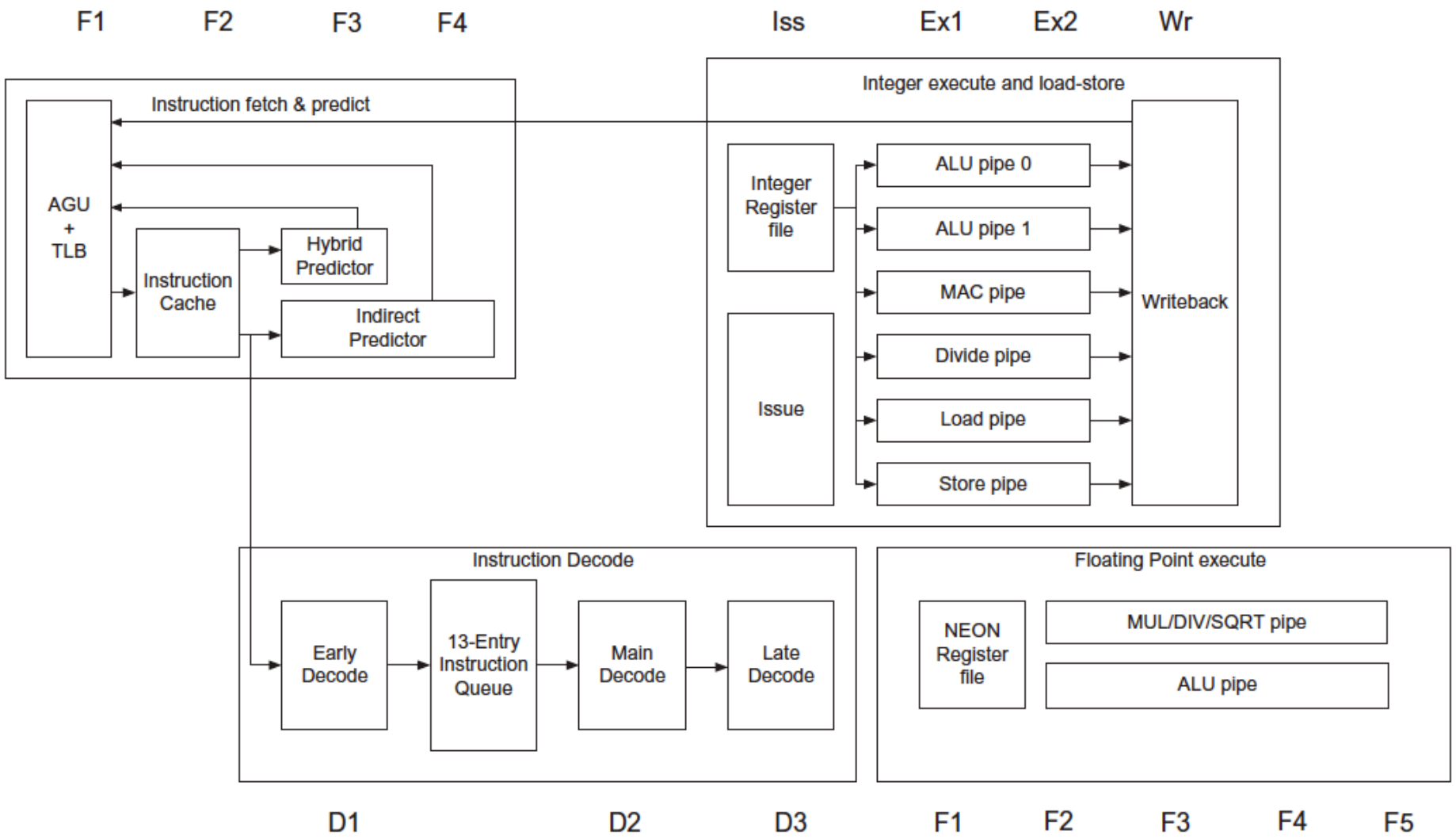
Decode: *Instructions dispatched to appropriate issue buffer*

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

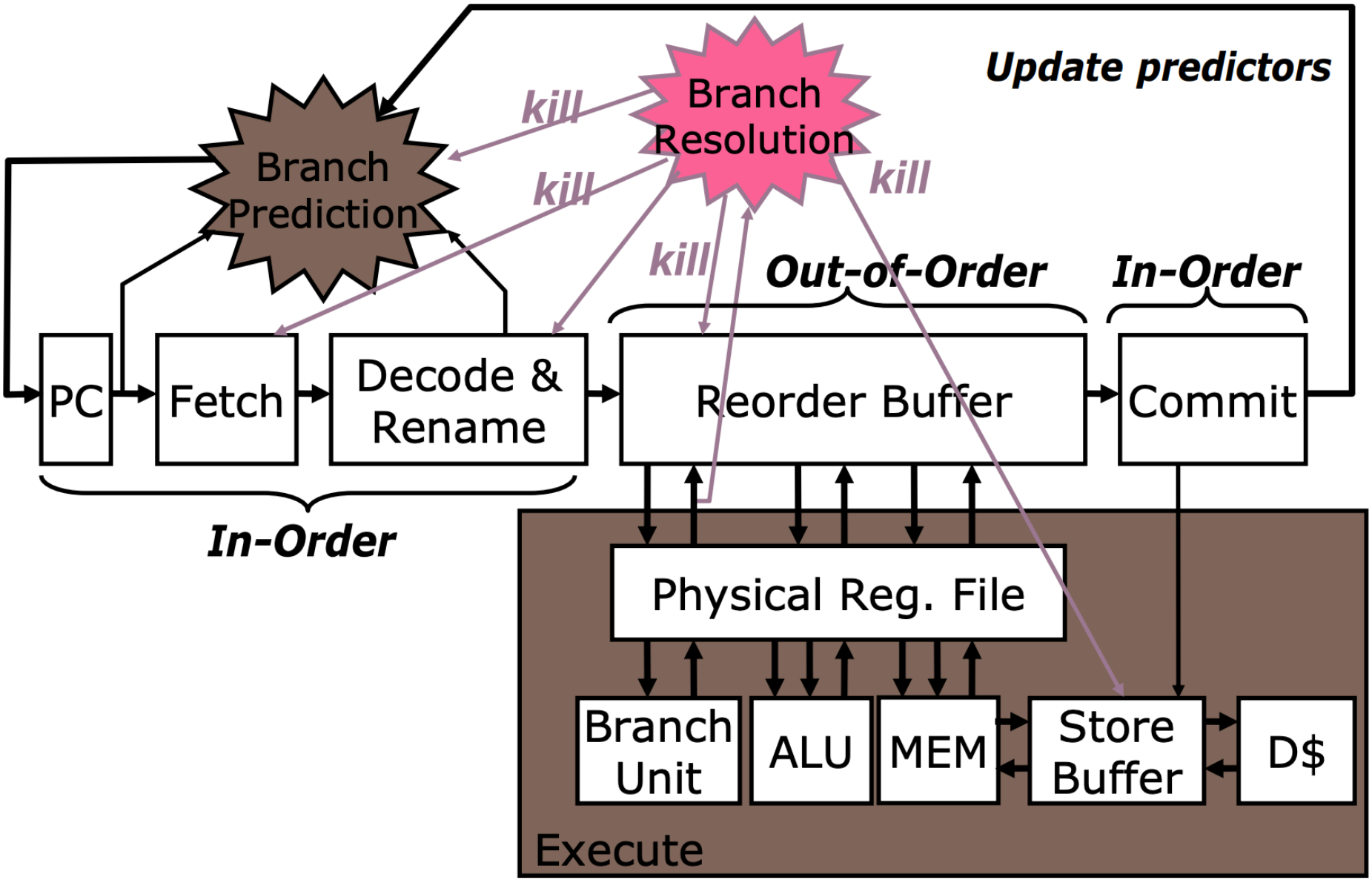
Commit: *Instruction irrevocably updates architectural state (aka "graduation"), or takes precise trap/interrupt.*

ARM Cortex A53 Pipeline

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

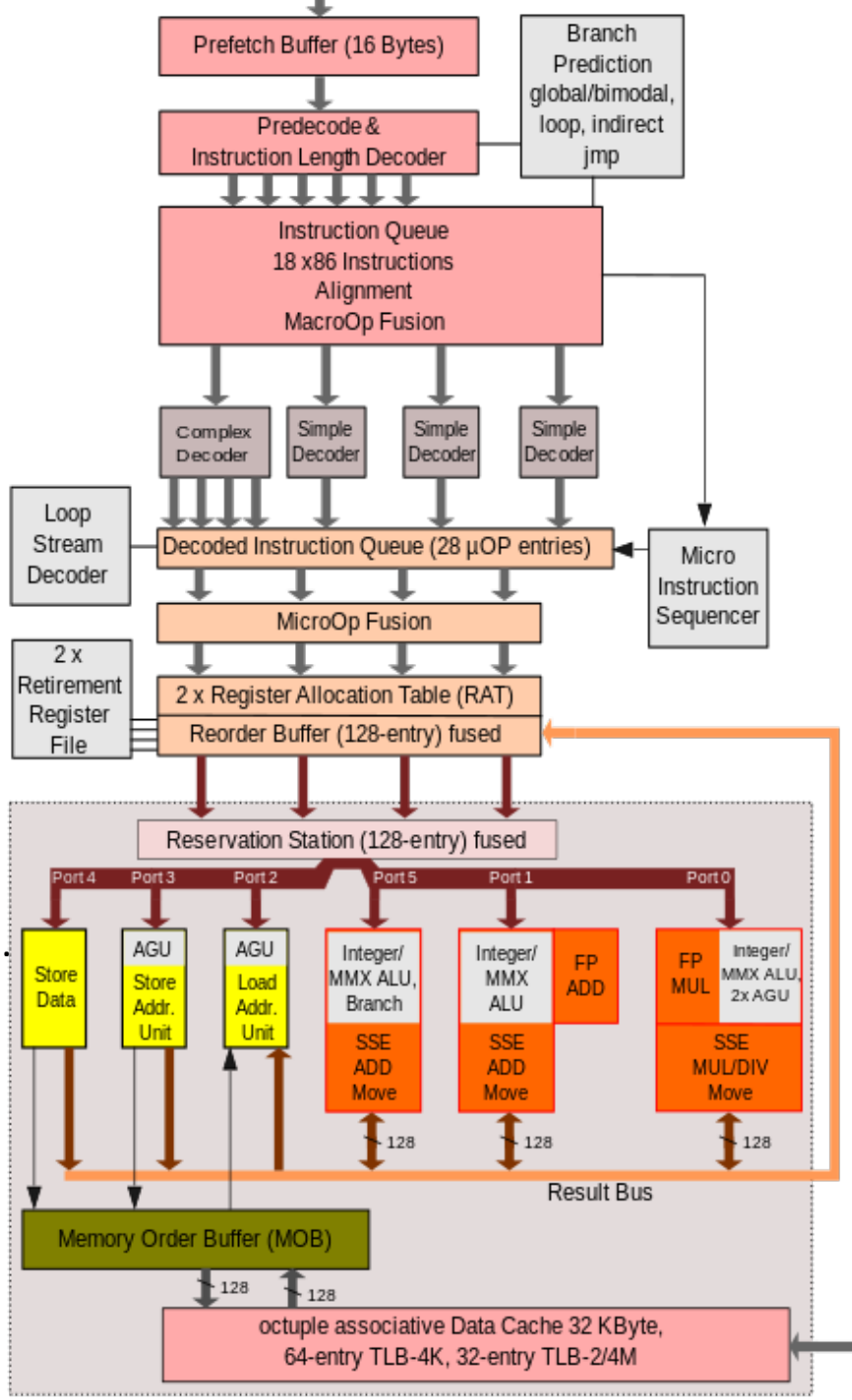


Speculative & Out-of-Order Execution



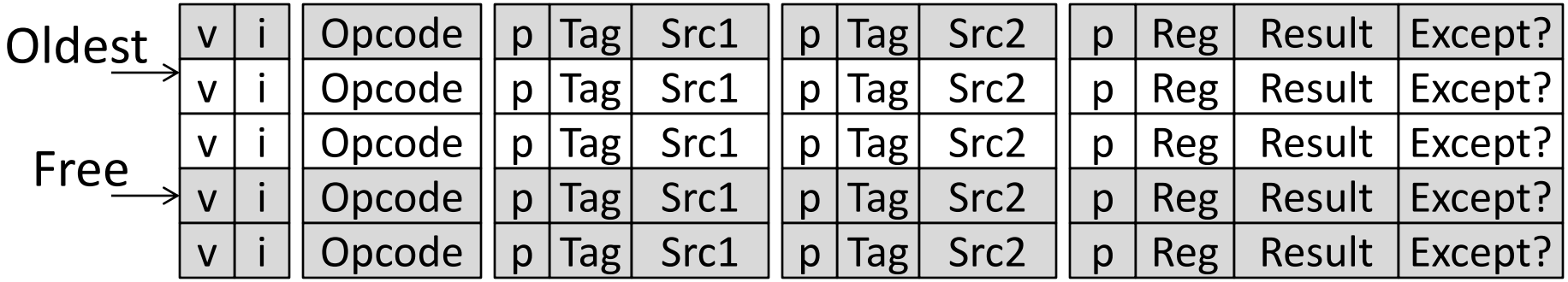
Intel Nehalem i7

- 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.



“Data-in-ROB” Design

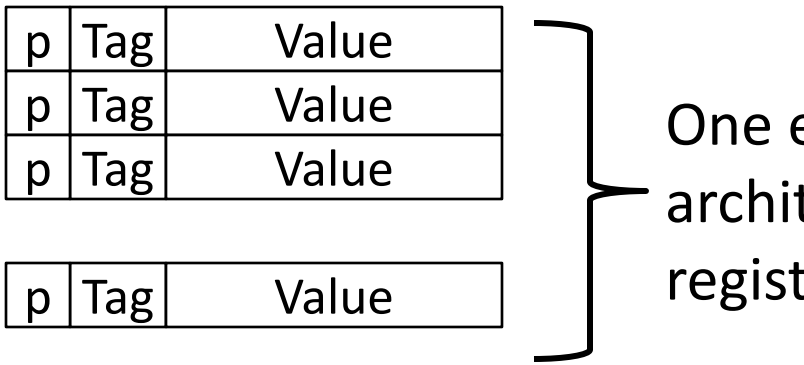
(HP PA8000, Pentium Pro, Core2Duo, Nehalem)



- ROB: Reorder Buffer
- 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.

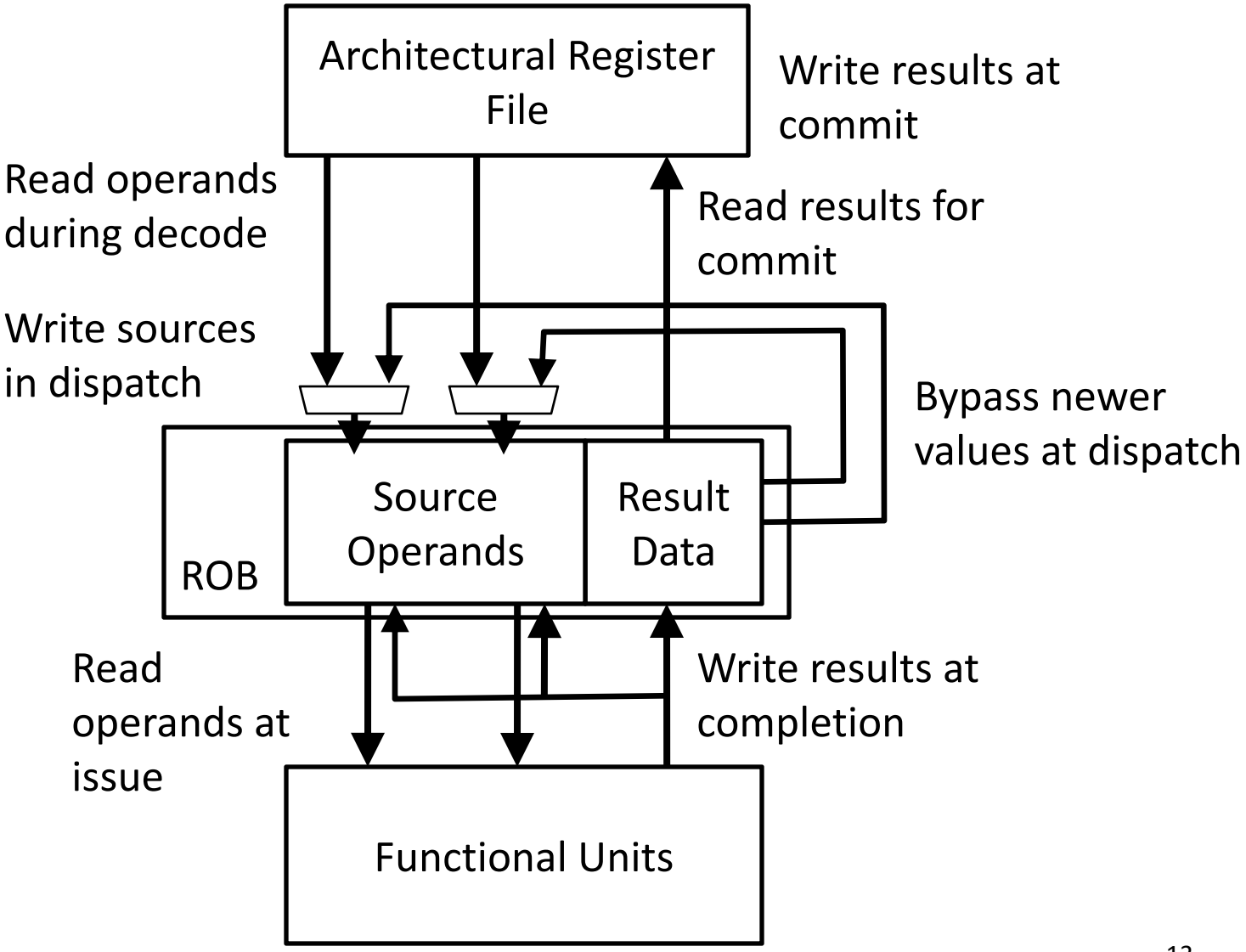
Managing Rename for Data-in-ROB

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

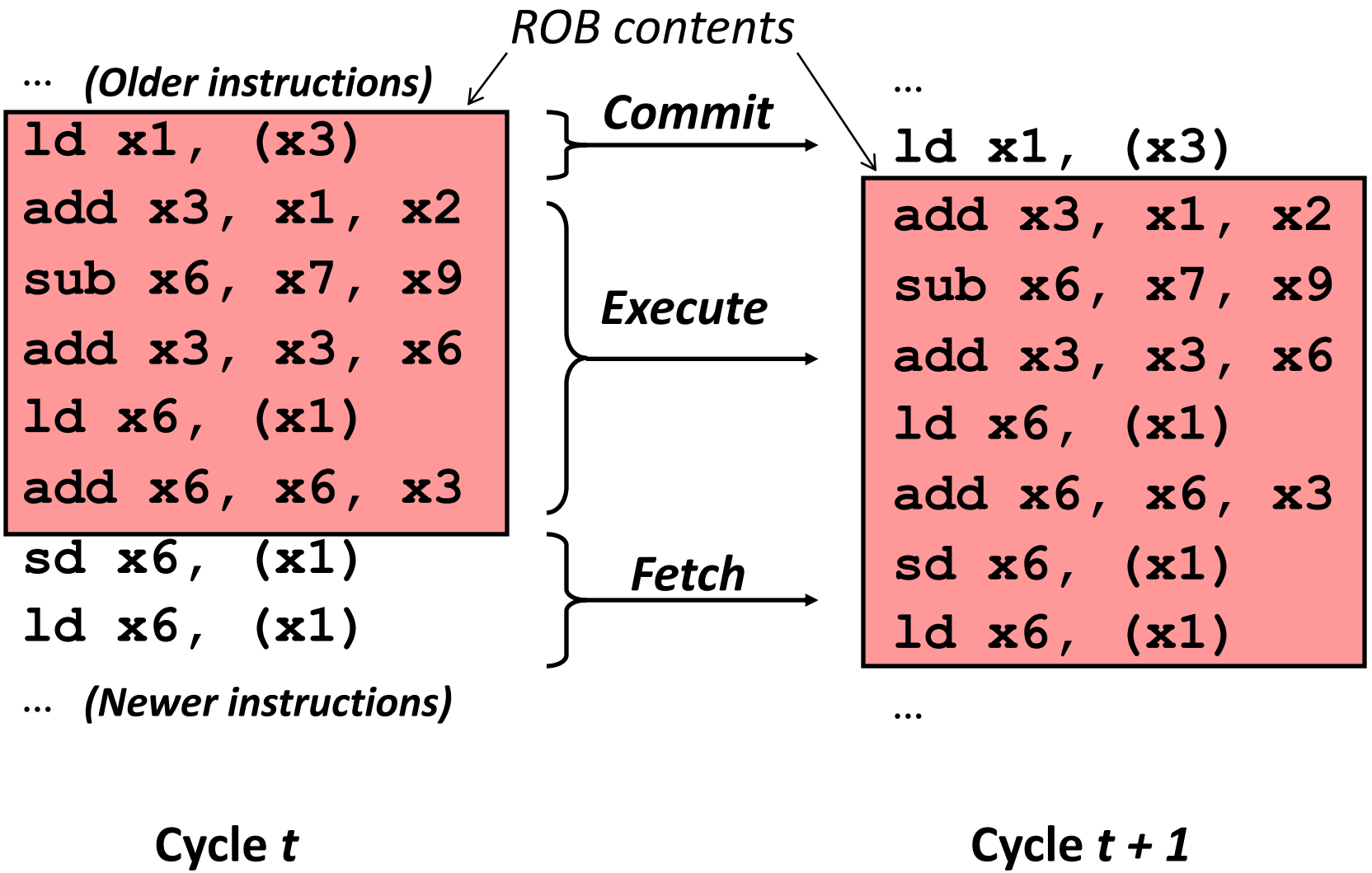


- If “p” bit set, then use value in architectural register file
- Else, tag field indicates instruction that will/has produced value
- For dispatch, read source operands <p,tag,value> from arch. regfile, then also read <p,result> from producing instruction in ROB at tag index, bypassing as needed. Copy 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



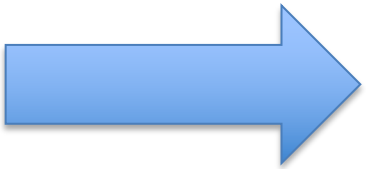
Reorder Buffer Holds Active Instructions (Decoded but not Committed)



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

```
1      r1 := m[1024]
2      r1 := r1 + 2
3      m[1032] := r1
4      r1 := m[2048]
5      r1 := r1 + 4
6      m[2056] := r1
```

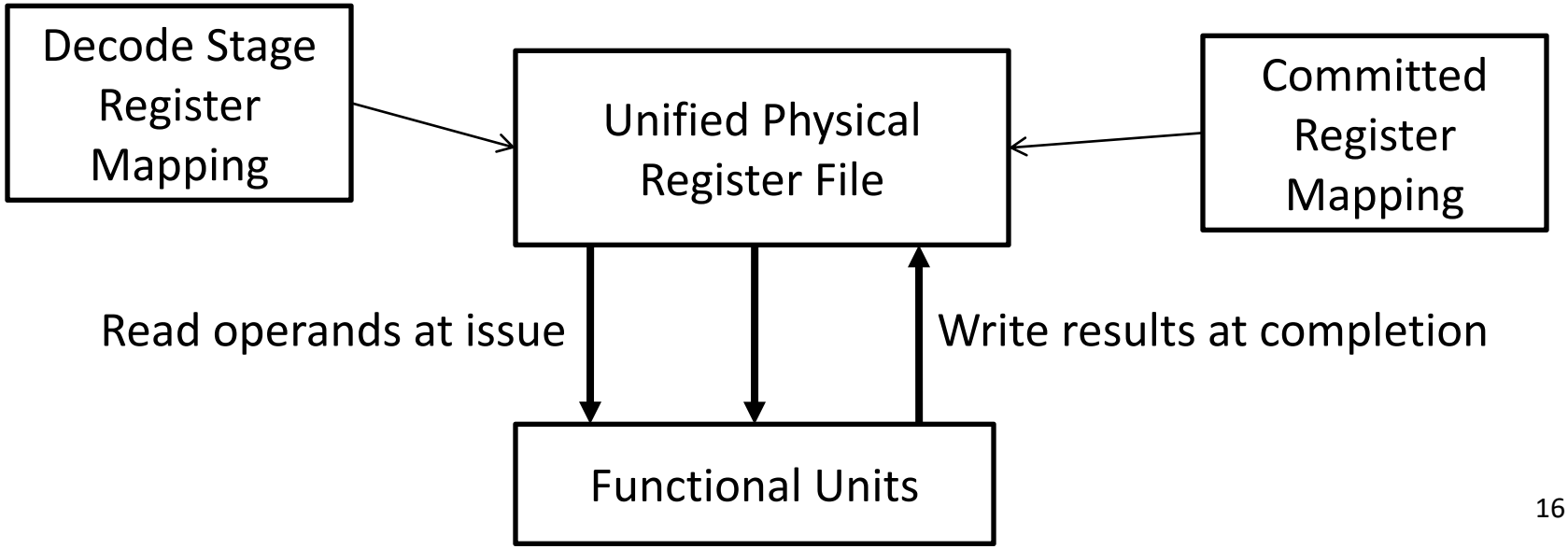


```
1      r1 := m[1024]
2      r1 := r1 + 2
3      m[1032] := r1
4      r2 := m[2048]
5      r2 := r2 + 4
6      m[2056] := r2
```

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

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

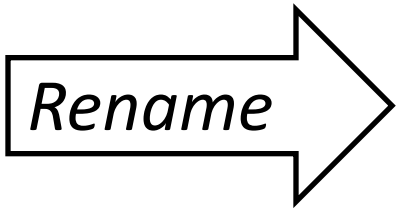
- Rename all architectural registers into a single *physical* register file during decode, no register values read
- Functional units read and write from single unified register file holding committed and temporary registers in execute
- Commit only updates mapping of architectural register to physical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

New-School Machine Structures (It's a bit more complicated!)

Software | *Hardware*

- Parallel Requests
  - Assigned to computer
  - e.g., Search "Katz"
- Parallel Threads
  - Assigned to core
  - e.g., Lookup, Ads
- Parallel Instructions
  - >1 instruction @ one time
  - e.g., 5 pipelined instructions
- Parallel Data
  - >1 data item @ one time
  - e.g., Add of 4 pairs of words
- Hardware descriptions
  - All gates @ one time
- Programming Languages

Harness Parallelism & Achieve High Performance

How do we know?

Warehouse Scale Computer

Smart Phone

What is Performance?

- *Latency* (or *response time* or *execution time*)
  - Time to complete one task
- *Bandwidth* (or *throughput*)
  - Tasks completed per unit time

Transportation Analogy

	Sports Car	Bus
Passenger Capacity	2	50
Travel Speed	250 km/h	100 km/h
Fuel consumption	20 l/100km	20 l/100km

Schwerin => Berlin trip: 200 km

	Sports Car	Bus
Travel Time	48 min	120 min
Time for 100 passengers	40 h	4 h
Fuel per passenger	2000 l	80 l

Latency & Throughput

Transportation	Computer
Travel Time	Program execution time (**latency**) e.g. time to update display
Time for 100 passengers	**Throughput**: e.g. number of server requests handled per hour
Fuel per passenger	Energy per task*: e.g.: - how many movies can you watch per battery charge - energy bill for datacenter

* Note: power is not a good measure, since low-power CPU might run for a long time to complete one task consuming more energy than faster computer running at higher power for a shorter time

Cloud Performance: Why Application Latency Matters

Server Delay (ms)	Increased time to next click (ms)	Queries/ user	Any clicks/ user	User satisfac- tion	Revenue/ User
50	--	--	--	--	--
200	500	--	-0.3%	-0.4%	--
500	1200	--	-1.0%	-0.9%	-1.2%
1000	1900	-0.7%	-1.9%	-1.6%	-2.8%
2000	3100	-1.8%	-4.4%	-3.8%	-4.3%

Figure 6.10 Negative impact of delays at Bing search server on user behavior [Brutlag and Schurman 2009].

- Key figure of merit: application responsiveness
  - Longer the delay, the fewer the user clicks, the less the user happiness, and the lower the revenue per user

Defining Relative CPU Performance

- $\text{Performance}_X = 1/\text{Program Execution Time}_X$
- $\text{Performance}_X > \text{Performance}_Y \Rightarrow$
  $1/\text{Execution Time}_X > 1/\text{Execution Time}_y \Rightarrow$
  $\text{Execution Time}_Y > \text{Execution Time}_X$
- Computer X is N times faster than Computer Y
  $\text{Performance}_X / \text{Performance}_Y = N$ or
  $\text{Execution Time}_Y / \text{Execution Time}_X = N$

Measuring CPU Performance

- Computers use a clock to determine when events takes place within hardware
- *Clock cycles:* discrete time intervals
  - aka clocks, cycles, clock periods, clock ticks
- *Clock rate* or *clock frequency:* clock cycles per second (inverse of clock cycle time)
- 3 GigaHertz clock rate
  => clock cycle time = $1/(3 \times 10^9)$ seconds
  clock cycle time = 333 picoseconds (ps)

CPU Performance Factors

- To distinguish between processor time and I/O, *CPU time* is time spent in processor
- CPU Time/Program
  = Clock Cycles/Program
  x Clock Cycle Time
- Or

  CPU Time/Program
  = Clock Cycles/Program ÷ Clock Rate

Iron Law of Performance

- A program executes instructions
- `CPU Time/Program`
  `= Clock Cycles/Program x Clock Cycle Time`
  `= Instructions/Program`
  `x Average Clock Cycles/Instruction`
  `x Clock Cycle Time`
- 1st term called *Instruction Count*
- 2nd term abbreviated *CPI* for average ***C****lock Cycles* ***P****er* ***I****nstruction*
- 3rd term is 1 / Clock rate

Restating Performance Equation

- $$\text{Time} = \frac{\text{Seconds}}{\text{Program}}$$

$$= \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock Cycle}}$$

What Affects Each Component? Instruction Count, CPI, Clock Rate

	Affects What?
Algorithm	
Programming Language	
Compiler	
Instruction Set Architecture	

What Affects Each Component? Instruction Count, CPI, Clock Rate

	Affects What?
Algorithm	Instruction Count, CPI
Programming Language	Instruction Count, CPI
Compiler	Instruction Count, CPI
Instruction Set Architecture	Instruction Count, Clock Rate, CPI

Question

Computer	Clock frequency	Clock cycles per instruction	#instructions per program	
A	1GHz	2	1000	
B	2GHz	5	800	
C	500MHz	1.25	400	
D	5GHz	10	2000	

- Which computer has the highest performance for a given program?

Question

Computer	Clock frequency	Clock cycles per instruction	#instructions per program	Calculation
A	1GHz	2	1000	1ns * 2 * 1000 = 2µs
B	2GHz	5	800	0.5ns 5 *800 = 2µs
C	500MHz	1.25	400	2ns 1.25 * 400 = 1µs
D	5GHz	10	2000	0.2ns * 10 * 2000 = 4µs

- Which computer has the highest performance for a given program?

Workload and Benchmark

- *Workload:* Set of programs run on a computer
  - Actual collection of applications run or made from real programs to approximate such a mix
  - Specifies programs, inputs, and relative frequencies
- *Benchmark:* Program selected for use in comparing computer performance
  - Benchmarks form a workload
  - Usually standardized so that many use them

SPEC
(System Performance Evaluation Cooperative)

- Computer Vendor cooperative for benchmarks, started in 1989
- SPECCPU2006
  - 12 Integer Programs
  - 17 Floating-Point Programs
- Often turn into number where bigger is faster
- *SPECratio*: reference execution time on old reference computer divide by execution time on new computer to get an effective speed-up

SPEC CPU 2017

SPECrate 2017 Integer	SPECspeed 2017 Integer	Language [1]	KLOC [2]	Application Area
500.perlbench_r	600.perlbench_s	C	362	Perl interpreter
502.gcc_r	602.gcc_s	C	1,304	GNU C compiler
505.mcf_r	605.mcf_s	C	3	Route planning
520.omnetpp_r	620.omnetpp_s	C++	134	Discrete Event simulation - computer network
523.xalancbmk_r	623.xalancbmk_s	C++	520	XML to HTML conversion via XSLT
525.x264_r	625.x264_s	C	96	Video compression
531.deepsjeng_r	631.deepsjeng_s	C++	10	Artificial Intelligence: alpha-beta tree search (Chess)
541.leela_r	641.leela_s	C++	21	Artificial Intelligence: Monte Carlo tree search (Go)
548.exchange2_r	648.exchange2_s	Fortran	1	Artificial Intelligence: recursive solution generator (Sudoku)
557.xz_r	657.xz_s	C	33	General data compression

SPECrate 2017 Floating Point	SPECspeed 2017 Floating Point	Language [1]	KLOC [2]	Application Area
503.bwaves_r	603.bwaves_s	Fortran	1	Explosion modeling
507.cactuBSSN_r	607.cactuBSSN_s	C++, C, Fortran	257	Physics: relativity
508.namd_r		C++	8	Molecular dynamics
510.parest_r		C++	427	Biomedical imaging: optical tomography with finite elements
511.povray_r		C++, C	170	Ray tracing
519.lbm_r	619.lbm_s	C	1	Fluid dynamics
521.wrf_r	621.wrf_s	Fortran, C	991	Weather forecasting
526.blender_r		C++, C	1,577	3D rendering and animation
527.cam4_r	627.cam4_s	Fortran, C	407	Atmosphere modeling
	628.pop2_s	Fortran, C	338	Wide-scale ocean modeling (climate level)
538.imagick_r	638.imagick_s	C	259	Image manipulation
544.nab_r	644.nab_s	C	24	Molecular dynamics
549.fotonik3d_r	649.fotonik3d_s	Fortran	14	Computational Electromagnetics
554.roms_r	654.roms_s	Fortran	210	Regional ocean modeling

[1] For multi-language benchmarks, the first one listed determines library and link options (details)

[2] KLOC = line count (including comments/whitespace) for source files used in a build / 1000

SPECINT2006 on AMD Barcelona

Description	Instruc-tion Count (B)	CPI	Clock cycle time (ps)	Execu-tion Time (s)	Refer-ence Time (s)	SPEC-ratio
Interpreted string processing	2,118	0.75	400	637	9,770	15.3
Block-sorting compression	2,389	0.85	400	817	9,650	11.8
GNU C compiler	1,050	1.72	400	724	8,050	11.1
Combinatorial optimization	336	10.0	400	1,345	9,120	6.8
Go game	1,658	1.09	400	721	10,490	14.6
Search gene sequence	2,783	0.80	400	890	9,330	10.5
Chess game	2,176	0.96	400	837	12,100	14.5
Quantum computer simulation	1,623	1.61	400	1,047	20,720	19.8
Video compression	3,102	0.80	400	993	22,130	22.3
Discrete event simulation library	587	2.94	400	690	6,250	9.1
Games/path finding	1,082	1.79	400	773	7,020	9.1
XML parsing	1,058	2.70	400	1,143	6,900	6.0

Summarizing Performance …

System	Rate (Task 1)	Rate (Task 2)
A	10	20
B	20	10

Clickers: Which system is faster?

A: System A

B: System B

C: Same performance

D: Unanswerable question!

… Depends Who's Selling

System	Rate (Task 1)	Rate (Task 2)	Average
A	10	20	15
B	20	10	15

Average throughput

System	Rate (Task 1)	Rate (Task 2)	Average
A	0.50	2.00	1.25
B	1.00	1.00	1.00

Throughput relative to B

System	Rate (Task 1)	Rate (Task 2)	Average
A	1.00	1.00	1.00
B	2.00	0.50	1.25

Throughput relative to A

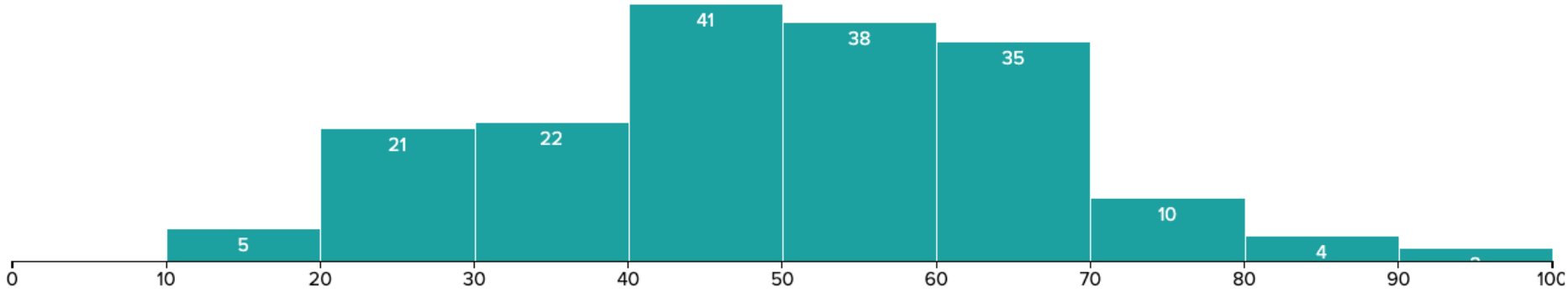
Summarizing SPEC Performance

- Varies from 6x to 22x faster than reference computer
- *Geometric mean* of ratios:
  N-th root of product
  of N ratios

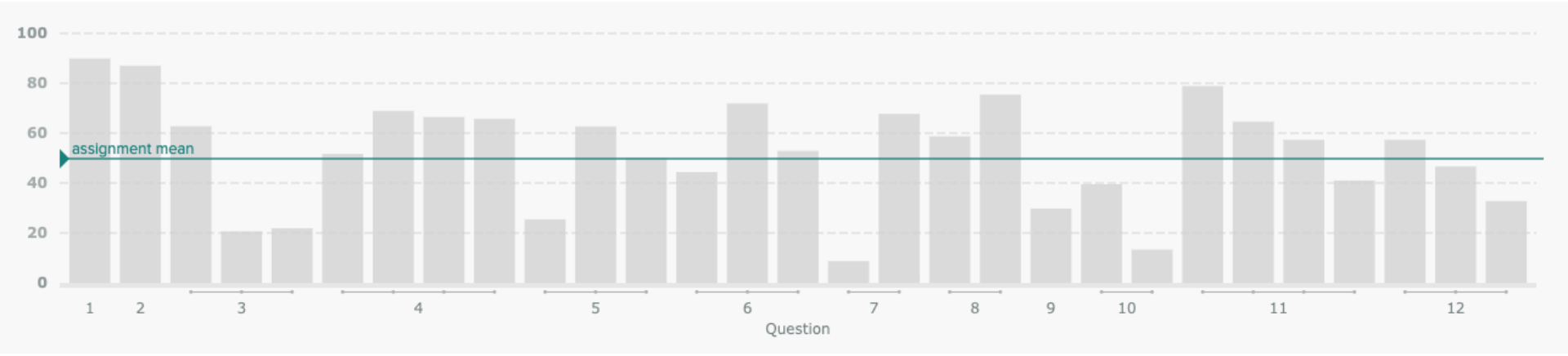
$$\sqrt[n]{\prod_{i=1}^{n} \text{Execution time ratio}_i}$$

  - Geometric Mean gives same relative answer no matter what computer is used as reference
- Geometric Mean for Barcelona is 11.7

Midterm I Review



MINIMUM	MEDIAN	MAXIMUM	MEAN	STD DEV
16.0	49.5	94.5	49.87	16.68



3 b) & c): TA will cover in discussion

A **quarter** is a single byte split into the following fields (1 sign, 3 exponent, 4 mantissa): `SEEEMMMM`. It has all the properties of **IEEE 754** (including denormal numbers, NaNs and $\pm\infty$) just with different ranges, precision and representations.

For a **quarter**, the bias of the exponent is 3, and the implicit exponent for denormal numbers are -2.

What is the largest number smaller than ∞?

In binary _____

In decimal _____

Which negative denormal number is closest to 0?

In binary _____

In decimal _____

(a) Memory of C

4 a)

```
1 #include <stdlib.h>
2
3 int main() {
4     static int p = 5;
5
6     char *str = ____________________________;
7     /* some other codes, and you can skip it. */
8     return 0;
9 }
```

1. You need to allocate a string `str` containing `p` characters. Write the code above (please use `malloc`).

Solution: `char *str = malloc(sizeof(char) * (p + 1));`

- Control characters like ”\0” are characters BUT:
- No guarantee that a “\0” will be at the end is given, so:
- Space for “\0” after the 5 characters needs to be allocated!

(a) **Idea I**: *Little Dragon* wants to directly retrieve the i^{th} and j^{th} byte of *num*, then swap them.

First of all, define a MACRO to get the i^{th} byte of *num*. Read the following C code, then help *Little Dragon* to fill in the blank lines (Line 4 and 10) so the output should be `0x34`. When defining the MACRO, use `&`, `|`, `^`, `~`, `>>`, `<<` operators **only**. Remember to write a meaningful MACRO such that *Little Dragon* can reuse it again (directly return `0x34` is not allowed)!

5 a)

```
1  #include <stdio.h>
2  #include <stdint.h>
3
4  #define GET_BYTE(num, ind) ____________________________________
5
6  int main(){
7      int number, index;
8      int8_t byte;
9      number = 0x12345678;
10
11     index = ___________; /* index is one of {0, 1, 2, 3} */
12     byte = GET_BYTE(number, index);
13     printf("%#x\n", byte); /* should print 0x34 */
14     return 0;
15 }
```

Write your answer above.

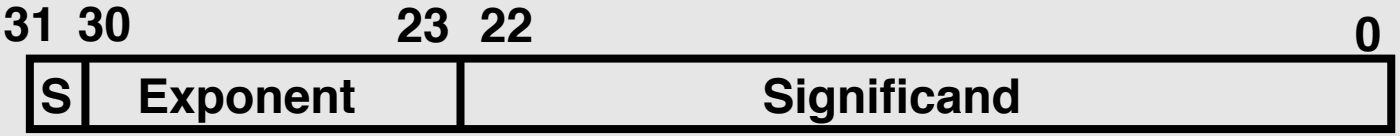
Solution: `(((num) >> ((ind) << 3)) & 0xFF)`; 2.

Notice that there should be brackets around `num` and `ind`!

7. RISC-V Basic

(a) Write a function in RISC-V code to return 0 if the input 32-bit float is an infinite value, else a non-zero value. The input and output will be stored in `a0`, as usual. Do not use pseduo instructions!

- **Summary (single precision):**



- **$(-1)^S$ x (1 + Significand) x $2^{(Exponent-127)}$**

Exponent	Significand	Object
0	0	0
0	nonzero	Denorm
1-254	anything	+/- fl. pt. #
255	0	+/- ∞
255	nonzero	NAN

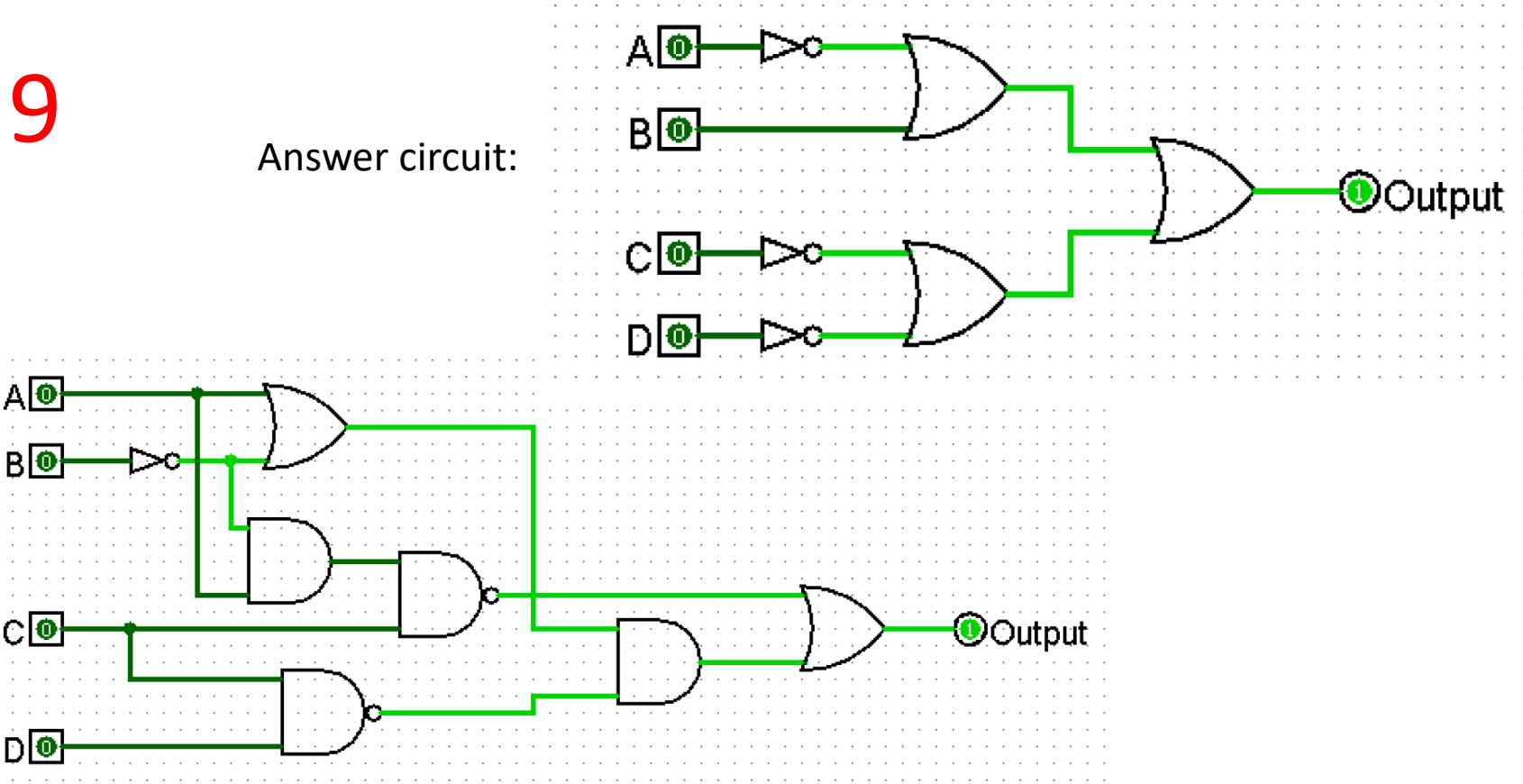
Solution:

```
1 is_not_infinity:
2         lui a1 0x80000
3         addi a1 a1 -1
4         and a0 a0 a1
5         lui a1 0x7F800
6         xor a0 a0 a1
7     ret
```

AND: 0111 1111 1111 1111 1111 1111 1111 1111
= 1000 0000 0000 0000 0000 0000 0000 0000 -1
XOR: S111 1111 1000 0000 0000 0000 0000 0000
= 0x7F 80 00 00

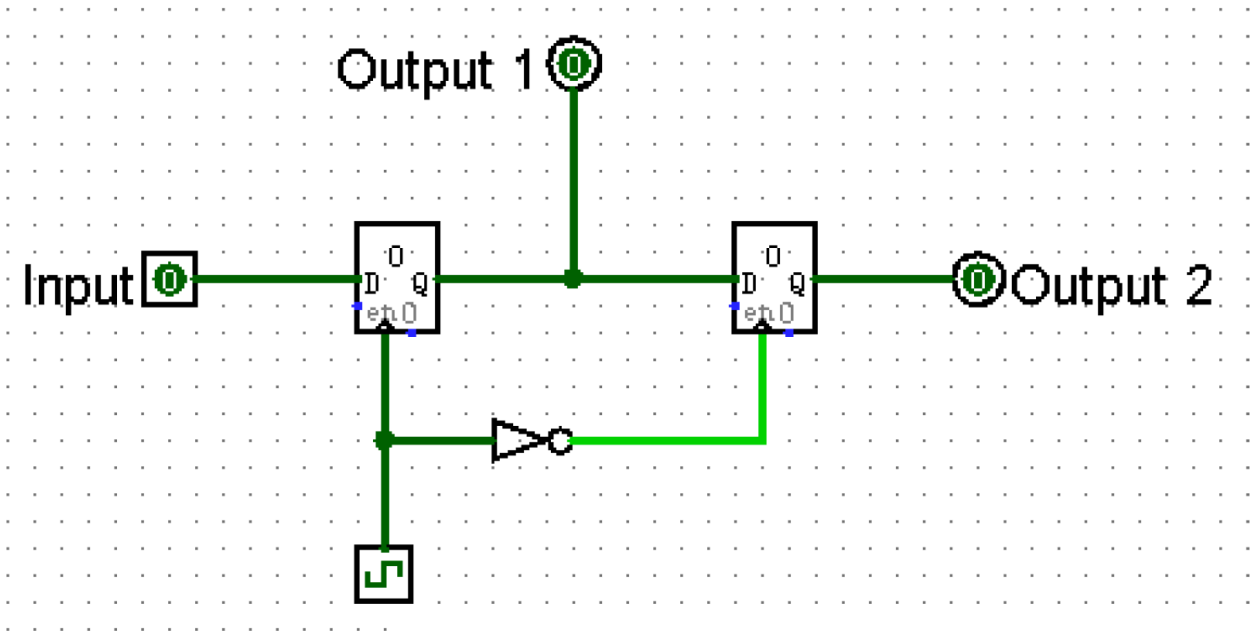
9

Answer circuit:



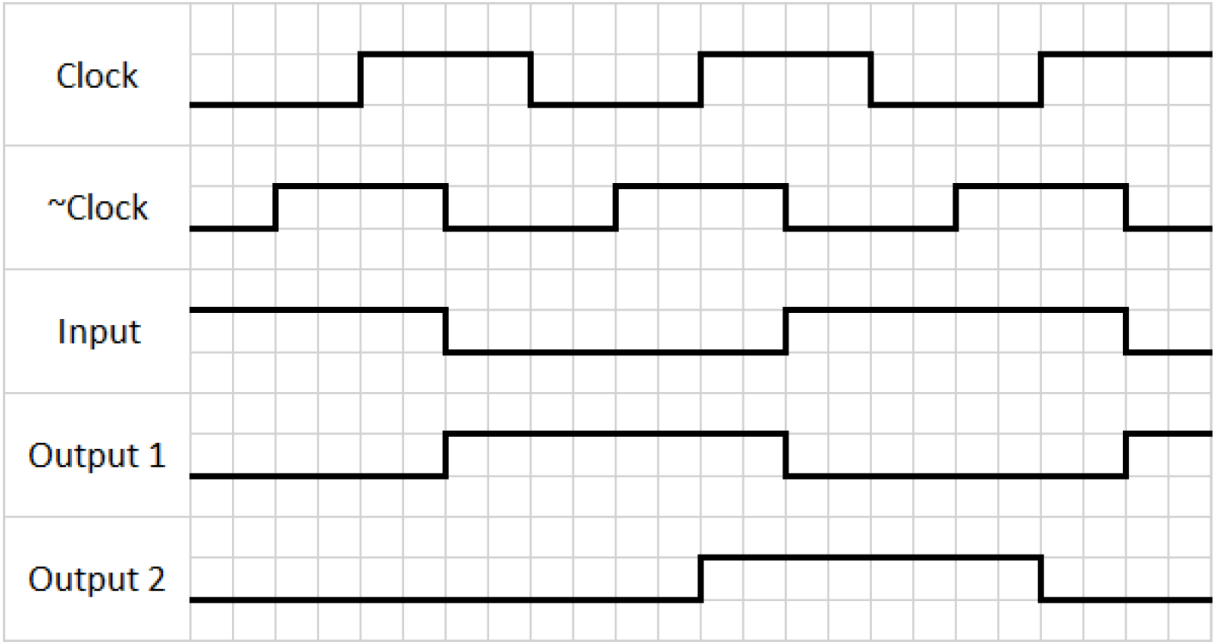
Solution:

$$\begin{aligned}
\overline{A\overline{B}C} + (A+\overline{B})(\overline{C}\,\overline{D}) &= \overline{A} + B + \overline{C} + (A+\overline{B})(\overline{C}+\overline{D}) \\
&= \overline{A} + B + \overline{C} + A\overline{C} + A\overline{D} + \overline{B}\,\overline{C} + \overline{B}\,\overline{D} \\
&= \overline{A} + B + \overline{C} + A\overline{D} + \overline{B}\,\overline{D} \\
&= \overline{A} + B + \overline{C} + \overline{D}
\end{aligned}$$



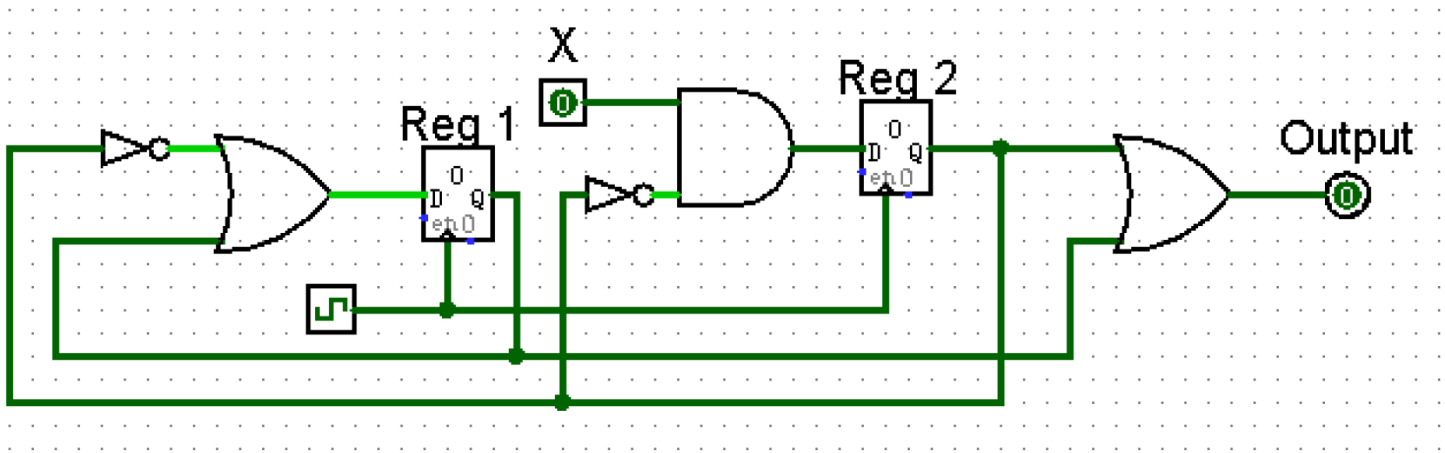
Solution:

10 a)



- NOT: 2ns
- Clk-to-Q: 2n
- Setup: 2ns
- Clock: 8ns

10 (b) Consider the following circuit. Assume the clock has a frequency of 50 MHz, all gates have a propagation delay of 6 ns, X changes 10 ns after the rising edge of clk, Reg1 and Reg2 have a clk-to-q delay of 1 ns.



What is the **longest possible setup time** such that there are no setup time violations?

Solution:

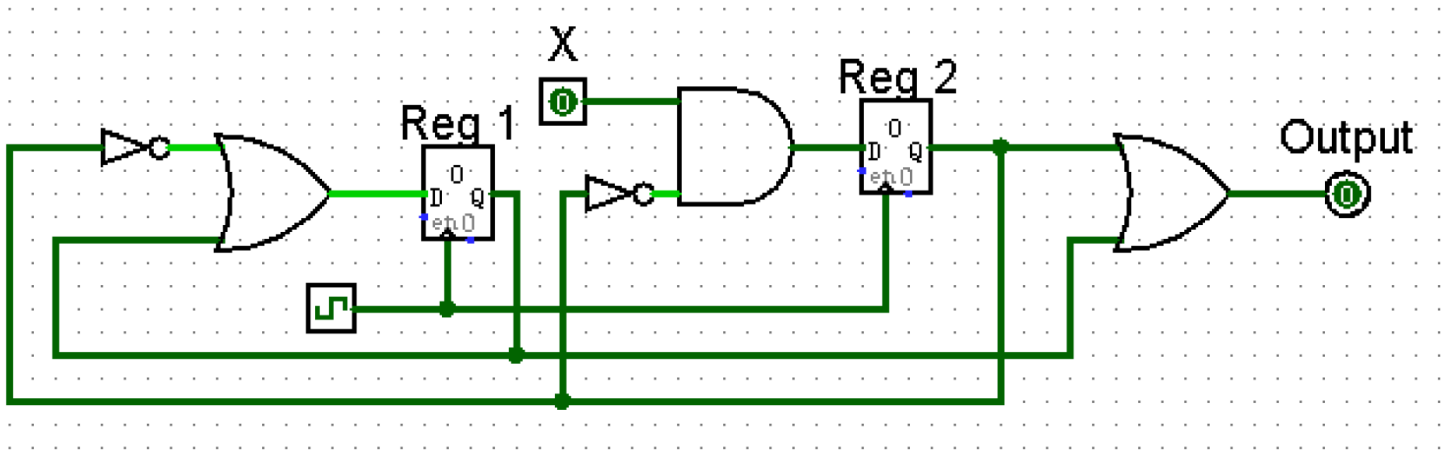
The clock period is $\frac{1}{50\times10^6}s = 20\ ns$.

Reg1 longest possible setup time: the path is *the output of Reg2* $\rightarrow$ *NOT* $\rightarrow$ *OR*, with a delay of 1 ns + 6 ns + 6 ns =13 ns. So 20 - 13 = 7 ns.

Reg2 longest possible setup time: the path is *X changes* $\rightarrow$ *AND*, with a delay of 10 ns + 6 ns = 16 ns. So 20 - 16 = 4 ns.

So longest setup time: min(7 ns, 4 ns) = 4 ns.

10 (b) Consider the following circuit. Assume the clock has a frequency of 50 MHz, all gates have a propagation delay of 6 ns, X changes 10 ns after the rising edge of clk, Reg1 and Reg2 have a clk-to-q delay of 1 ns.



What is the **longest possible hold time** such that there are no hold time violations?

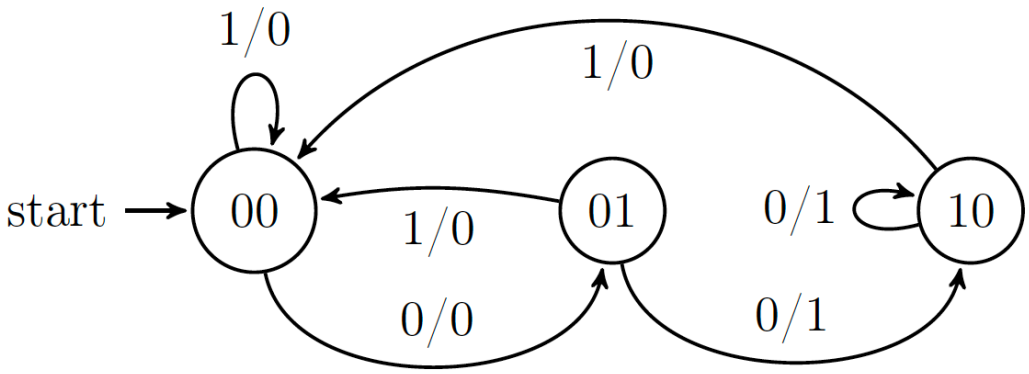
Reg1 longest possible hold time: the path is *the output of* $Reg1 \rightarrow OR$, with a delay of 1 ns + 6 ns = 7 ns.

Reg2 longest possible hold time: the path is *the output of* $Reg2 \rightarrow NOT \rightarrow AND$, with a delay of 1 ns + 6 ns + 6ns = 13 ns.

So longest hold time: min(7 ns, 13 ns) = 7 ns.

11(d) Draw a FSM that outputs 1 when it receives two or more successive '0'.

Solution:

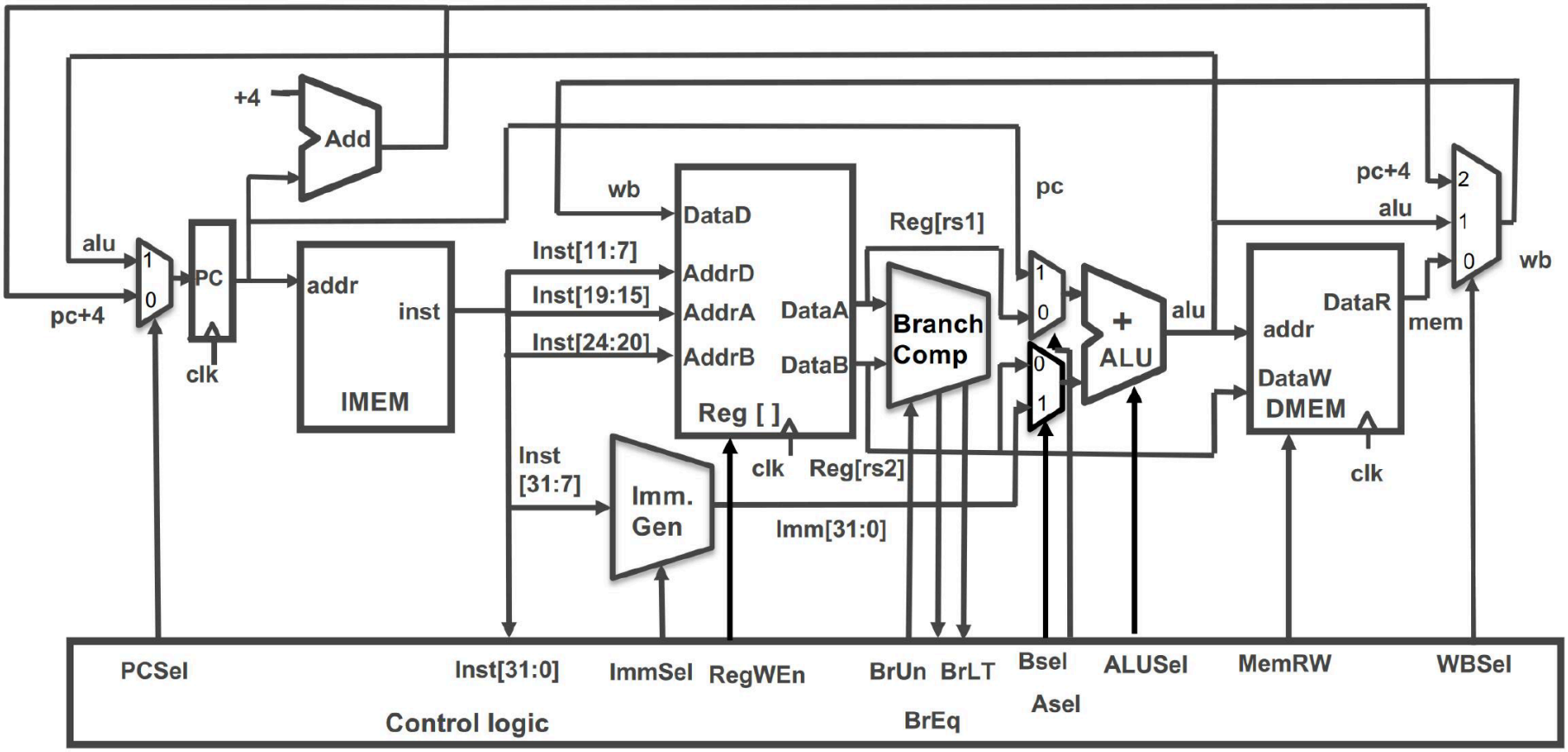


12 b)

(b) Which of following instructions involves all stages of execution?

A. addi B. jalr C. lw D. auipc

Solution: C



3 (c) Assume t3 = 0x8fffffff, t4 = 0x0fffffff. Write down control signals for **blt t3, t4, label**. Please use * to indicate that what this signal is does not matter.

11 c)

PCSel	ImmSel	RegWEn	BrUn	BrEq	BrLT	ASel	BSel	ALUSel	MemRW	WBSel

Solution: PCSel = 1 ImmSel = B RegWEn = 0 BrUn = 0 BrEq = 0 BrLT = 1 ASel = 1 BSel = 1 ALUSel = Add MemRW = Read WBSel = *