



CS211

Advanced Computer Architecture

L05 Pipeline II

Chundong Wang
October 10th, 2025

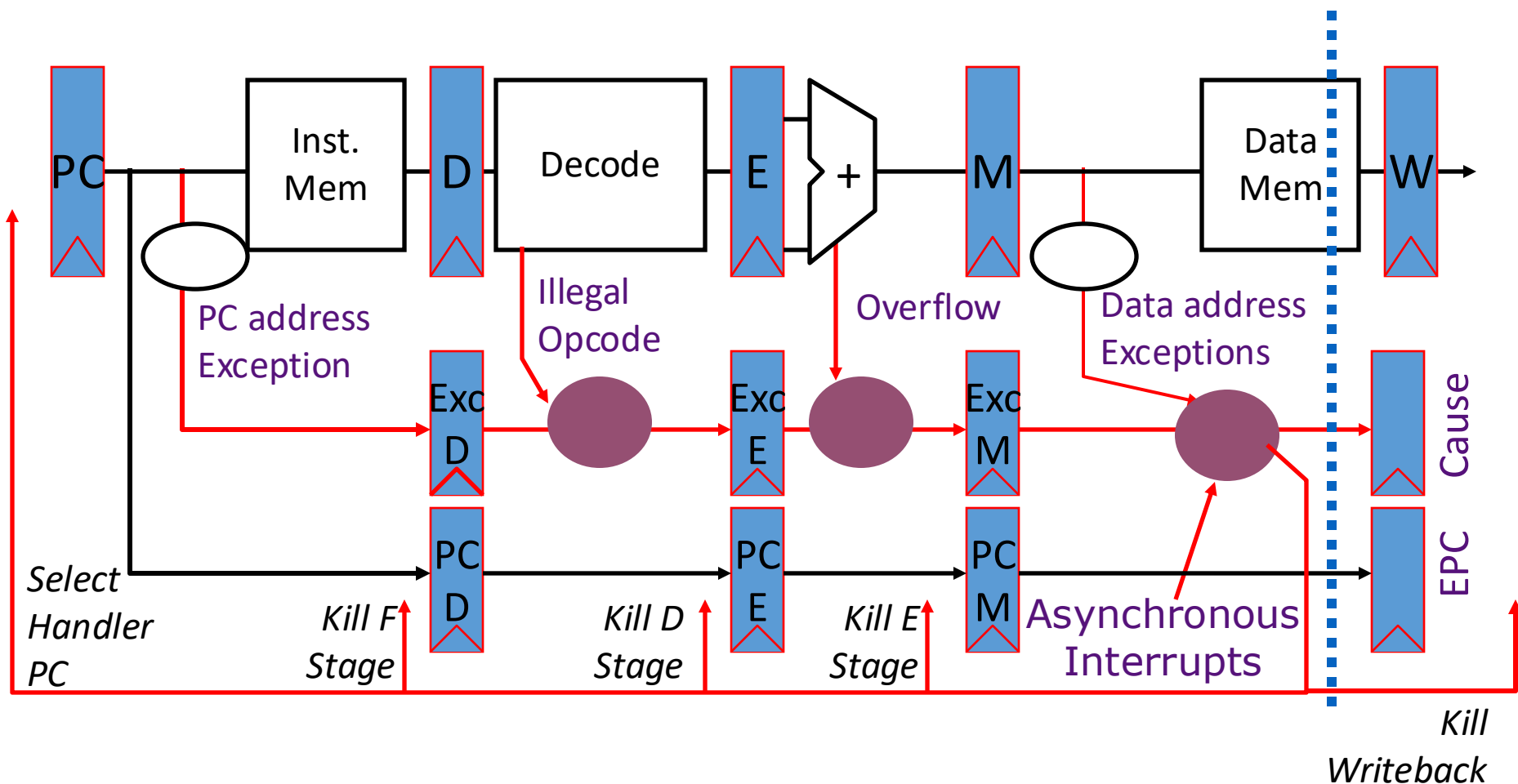


Previously in CS211

- Iron law of performance:
 - $\text{time/program} = \text{insts/program} * \text{cycles/inst} * \text{time/cycle}$
- Classic 5-stage RISC pipeline
- Structural, data, and control hazards
- Structural hazards handled with interlock or more hardware
- Data hazards include RAW, WAR, WAW
 - Handle data hazards with interlock, bypass, or speculation
- Control hazards (branches, interrupts) most difficult as change which is next instruction
 - Branch prediction commonly used
- Precise traps: stop cleanly on one instruction, all previous instructions completed, no following instructions have changed architectural state

Recap: Exception Handling 5-Stage Pipeline

Commit Point





Recap: Exception Handling 5-Stage Pipeline

- Hold exception flags in pipeline until commit point (M stage)
- Exceptions in earlier pipe stages override later exceptions *for a given instruction*
- Exceptions in earlier instructions override exceptions in later instructions
- Inject external interrupts at commit point (override others)
- If trap at commit: update Cause and EPC registers, kill all stages, inject handler PC into fetch stage



Recap: Speculating on Exceptions

- Prediction mechanism
 - Exceptions are rare, so simply predicting no exceptions is very accurate!
- Check prediction mechanism
 - Exceptions detected at end of instruction execution pipeline, special hardware for various exception types
- Recovery mechanism
 - Only write architectural state at commit point, so can throw away partially executed instructions after exception
 - Launch exception handler after flushing pipeline
- Bypassing allows use of uncommitted instruction results by following instructions

Deeper Pipelines: MIPS R4000

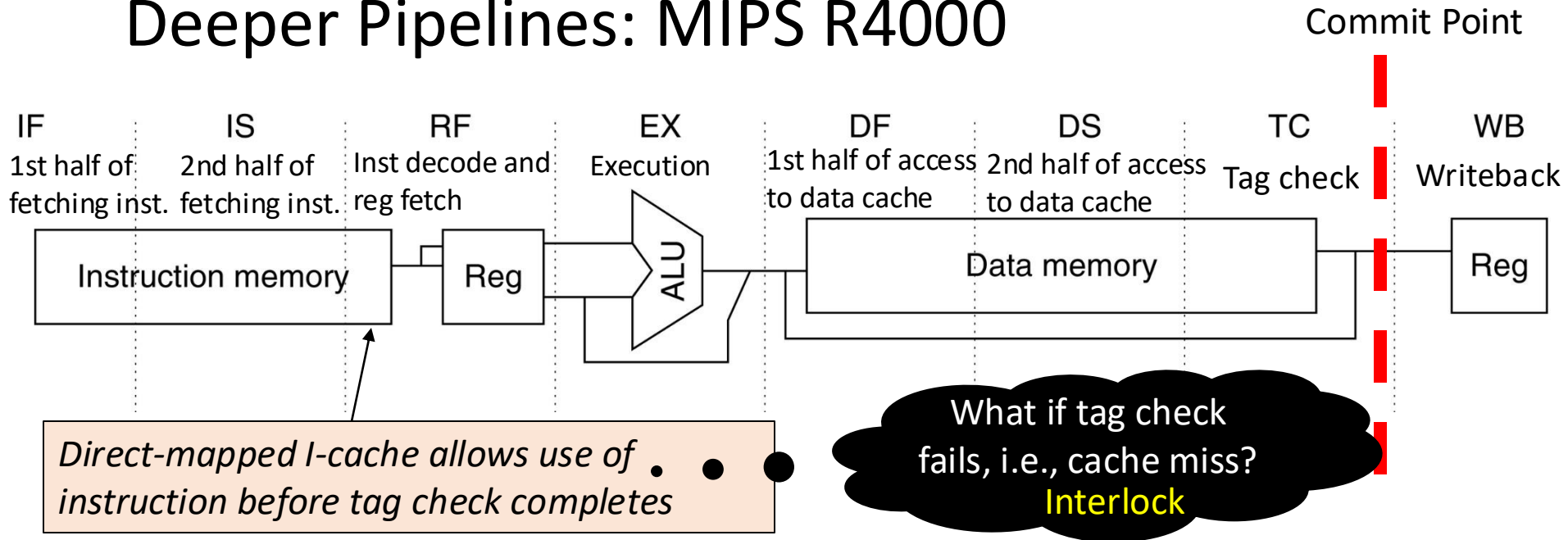


Figure C.36 The eight-stage pipeline structure of the R4000 uses pipelined instruction and data caches. The pipe stages are labeled and their detailed function is described in the textbook. The vertical dashed lines represent the stage boundaries as well as the location of pipeline latches. The instruction is actually available at the end of IS, but the tag check is done in RF, while the registers are fetched. Thus, we show the instruction memory as operating through RF. The TC stage is needed for data memory access, because we cannot write the data into the register until we know whether the cache access was a hit or not.

R4000 Load-Use Delay

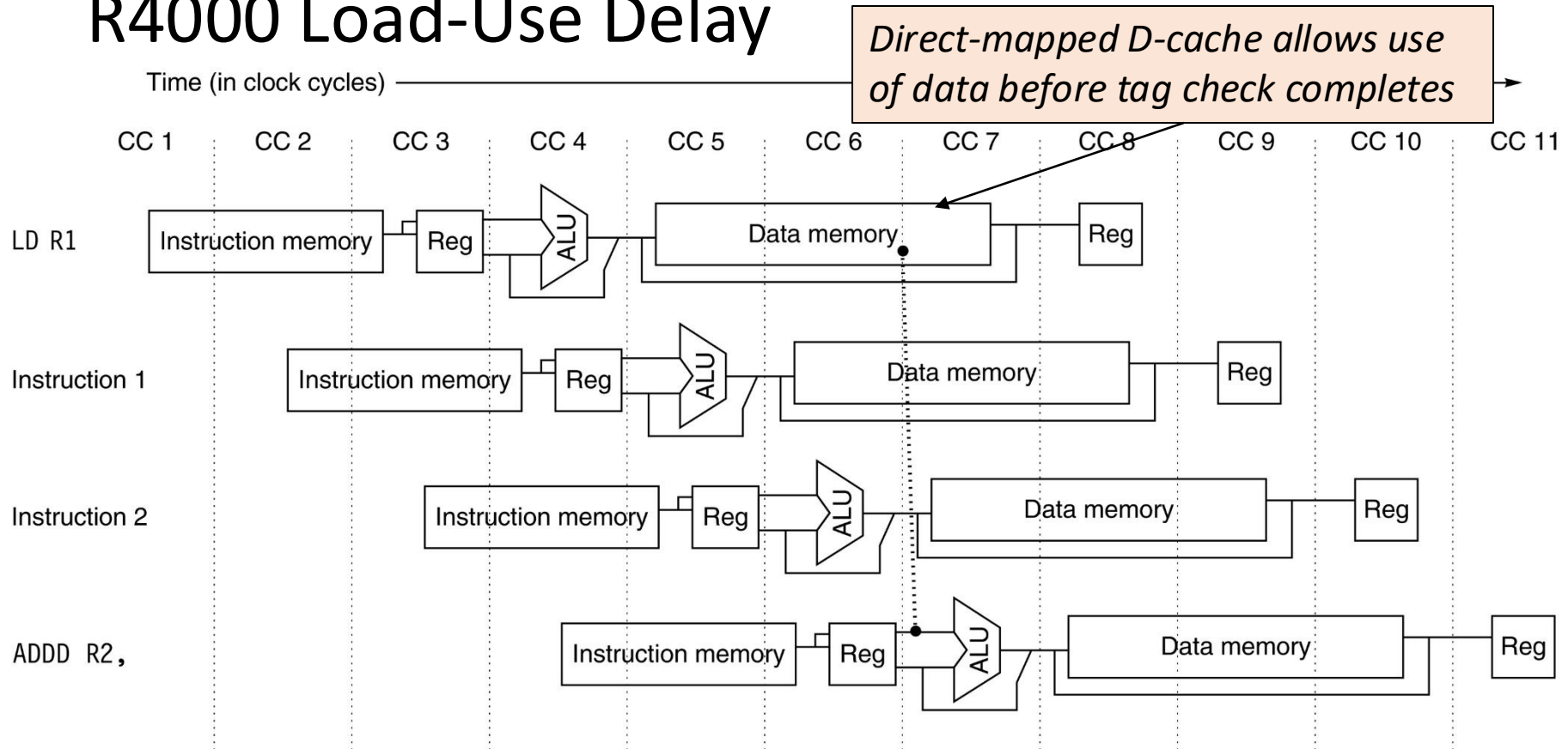


Figure C.37 The structure of the R4000 integer pipeline leads to a **x2** load delay. A x1 delay is possible because the data value is available at the end of DS and can be bypassed. If the tag check in TC indicates a miss, the pipeline is backed up a cycle, when the correct data are available.

R4000 Branches

The CPU pipeline has a branch delay of three cycles and a load delay of two cycles. (Chapter 3.3)

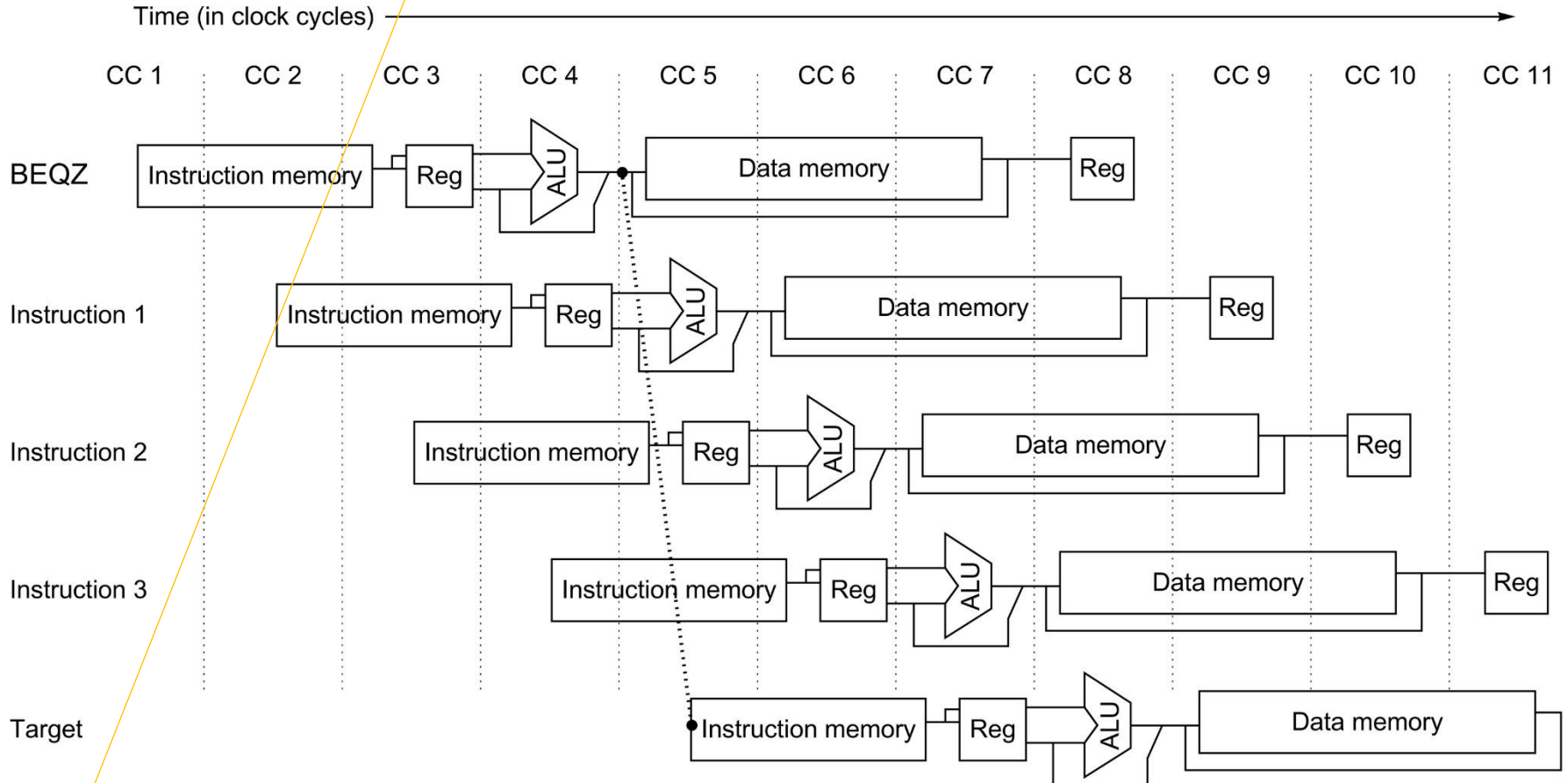


Figure C.39 The basic branch delay is three cycles, because the condition evaluation is performed during EX.

https://groups.csail.mit.edu/cag/raw/documents/R4400_Uman_book_Ed2.pdf (Manual)



Simple vector-vector add code example

```
#      for(i=0; i<N; i++)
#          A[i] = B[i]+C[i];

loop: fld f0, 0(x2) // x2 points to B
      fld f1, 0(x3) // x3 points to C
      fadd.d f2, f0, f1
      fsd f2, 0(x1) // x1 points to A
      addi x1, x1, 8 // Bump pointer
      addi x2, x2, 8 // Bump pointer
      addi x3, x3, 8 // Bump pointer
      bne x1, x4, loop // x4 holds end
```



Simple Pipeline Scheduling

Can **reschedule** code to try to reduce pipeline hazards

```
loop: fld f0, 0(x2) // x2 points to B
      fld f1, 0(x3) // x3 points to C
      addi x3, x3, 8 // Bump pointer
      addi x2, x2, 8 // Bump pointer
      fadd.d f2, f0, f1
      addi x1, x1, 8 // Bump pointer
      fsd f2, -8(x1) // x1 points to A
      bne x1, x4, loop // x4 holds end
```

Long latency loads and floating-point operations limit parallelism within a single loop iteration



One way to reduce hazards: Loop Unrolling

Can unroll to expose more parallelism, reduce dynamic instruction count

```
loop:  fld f0, 0(x2) // x2 points to B
      fld f1, 0(x3) // x3 points to C
      fld f10, 8(x2)
      fld f11, 8(x3)
      addi x3,x3,16    // Bump pointer
      addi x2,x2,16    // Bump pointer
      fadd.d f2, f0, f1
      fadd.d f12, f10, f11
      addi x1,x1,16    // Bump pointer
      fsd f2, -16(x1) // x1 points to A
      fsd f12, -8(x1)
      bne x1, x4, loop // x4 holds end
```

- Unrolling limited by number of architectural registers
- Unrolling increases instruction cache footprint
- More complex code generation for compiler, has to understand pointers
- Can also software pipeline, but has similar concerns



Alternative Approach: Decoupling (*lookahead*, *runahead*) in microarchitecture

Can separate **control and memory address** operations from **data computations**:

```
loop: fld f0, 0(x2) // x2 points to B
      fld f1, 0(x3) // x3 points to C
      fadd.d f2, f0, f1
      fsd f2, 0(x1) // x1 points to A
      addi x1, x1, 8 // Bump pointer
      addi x2, x2, 8 // Bump pointer
      addi x3, x3, 8 // Bump pointer
      bne x1, x4, loop // x4 holds end
```

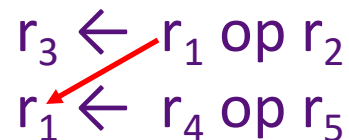
The control and address operations do not depend on the data computations, so can be computed early relative to the data computations, which can be delayed until later.

Decoupled Access/Execute

- Motivation: Tomasulo's algorithm too complex to implement
 - 1980s before HPS, Pentium Pro
- Tomasulo's algorithm
 - Register renaming for WAR and WAW

Anti-dependence

$r_3 \leftarrow r_1 \text{ op } r_2$
 $r_1 \leftarrow r_4 \text{ op } r_5$



Write-after-Read
(WAR) hazard

Anti-dependence is not data dependence. We can use an idle register, say r8, to replace r1.

No dependence

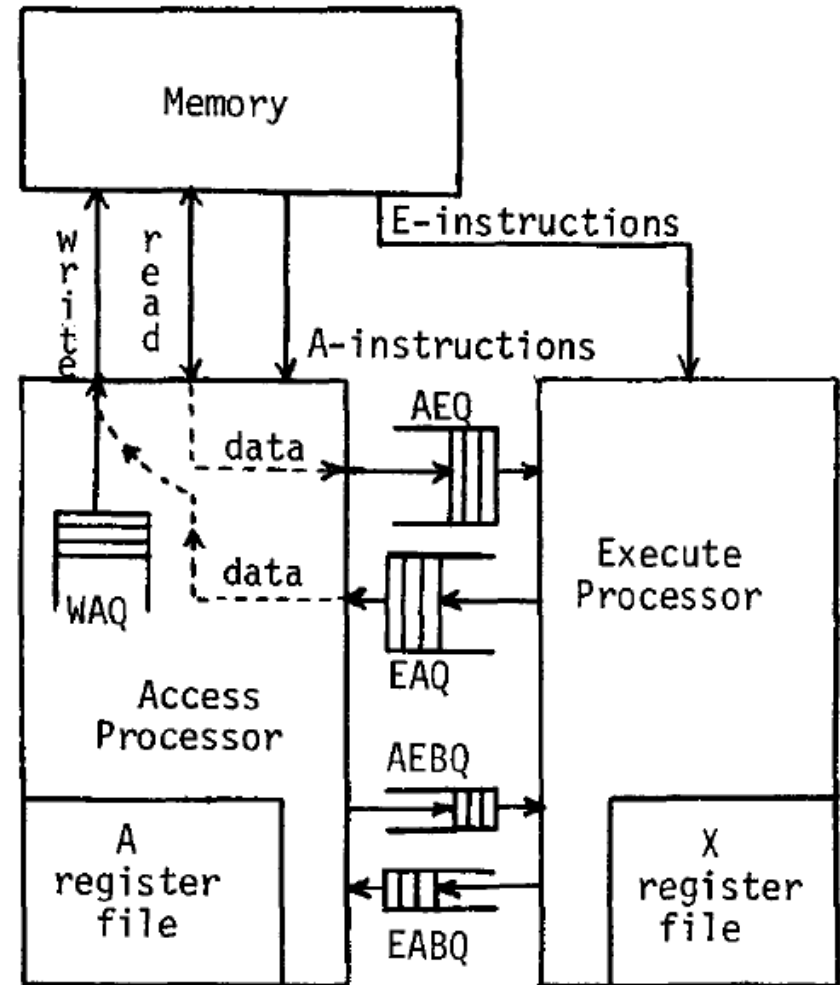
$r_3 \leftarrow r_1 \text{ op } r_2$
 $r_8 \leftarrow r_4 \text{ op } r_5$



Suppose there are *extra* registers (**reservation stations**) that even ISA does not know, but can be used for such renaming.

Decoupled Access/Execute

- Motivation: Tomasulo's algorithm too complex to implement
 - 1980s before HPS, Pentium Pro
- Idea: **Decouple operand access and execution via two separate instruction streams that communicate via ISA-visible queues.**
- J. Smith, “Decoupled Access/Execute Computer Architectures,” ISCA 1982, ACM TOCS 1984.



Decoupled Access/Execute (II)

- Compiler generates two instruction streams (A and E)
 - Synchronizes the two upon control flow instructions (using branch queues)

```

q = 0.0
Do 1 k = 1, 400
1  x(k) = q + y(k) * (r * z(k+10) + t * z(k+11))
    
```

Fig. 2a. Lawrence Livermore Loop 1 (HYDRO EXCERPT)

	A7 ← -400	. negative loop count
	A2 ← 0	. initialize index
	A3 ← 1	. index increment
	X2 ← r	. load loop invariants
	X5 ← t	. into registers
loop:	X3 ← z + 10, A2	. load z(k+10)
	X7 ← z + 11, A2	. load z(k+11)
	X4 ← X2 *f X3	. r*z(k+10)-flt. mult.
	X3 ← X5 *f X7	. t * z(k+11)
	X7 ← y, A2	. load y(k)
	X6 ← X3 +f X4	. r*z(x+10)+t*z(k+11))
	X4 ← X7 *f X6	. y(k) * (above)
	A7 ← A7 + 1	. increment loop counter
	x, A2 ← X4	. store into x(k)
	A2 ← A2 + A3	. increment index
	JAM loop	. Branch if A7 < 0

Fig. 2b. Compilation onto CRAY-1-like architecture

<u>Access</u>	<u>Execute</u>
.	
.	
.	
AEQ ← z + 10, A2	X4 ← X2 *f AEQ
AEQ ← z + 11, A2	X3 ← X5 *f AEQ
AEQ ← y, A2	X6 ← X3 +f X4
A7 ← A7 + 1	EAQ ← AEQ *f X6
x, A2 ← EAQ	.
A2 ← A2+ A3	.
.	.
.	.
.	.

Fig. 2c. Access and execute programs for straight-line section of loop



Decoupled Access/Execute (III)

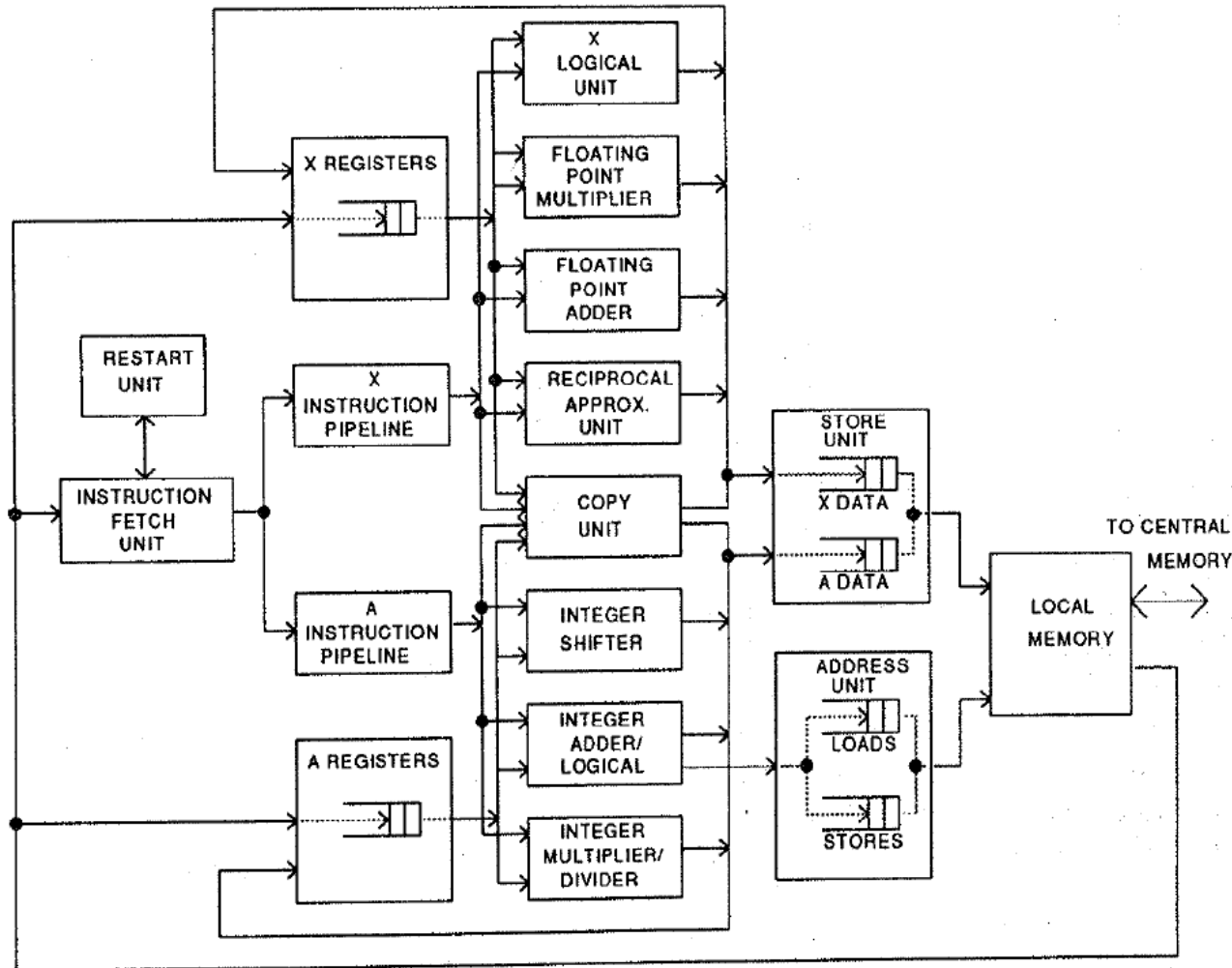
- Advantages:

- + Execute stream can run ahead of the access stream and vice versa
- + If A takes a cache miss, E can perform useful work
- + If A hits in cache, it supplies data to lagging E
- + Queues reduce the number of required registers
- + Limited out-of-order execution without wakeup/select complexity

- Disadvantages:

- Compiler support to partition the program and manage queues
 - Determines the amount of decoupling
- Branch instructions require synchronization between A and E
- Multiple instruction streams (can be done with a single one, though)

Astronautics ZS-1



- Single stream steered into A and X pipelines
Each pipeline in-order

Smith et al., “**The ZS-1 central processor**,”
ASPLOS 1987.

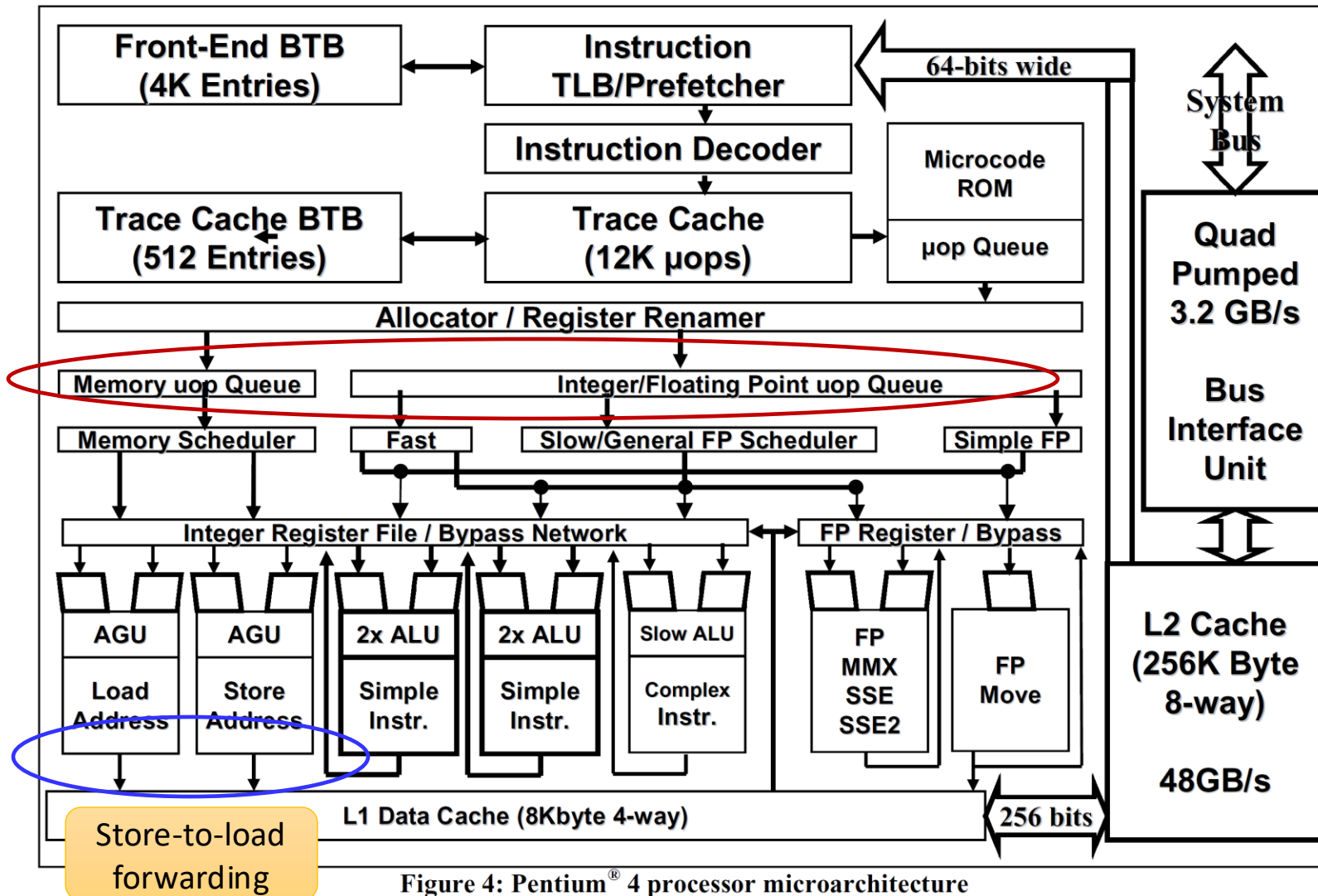
Smith, “**Dynamic Instruction Scheduling and the Astronautics ZS-1**,” IEEE Computer 1989.



Astronautics ZS-1 Instruction Scheduling

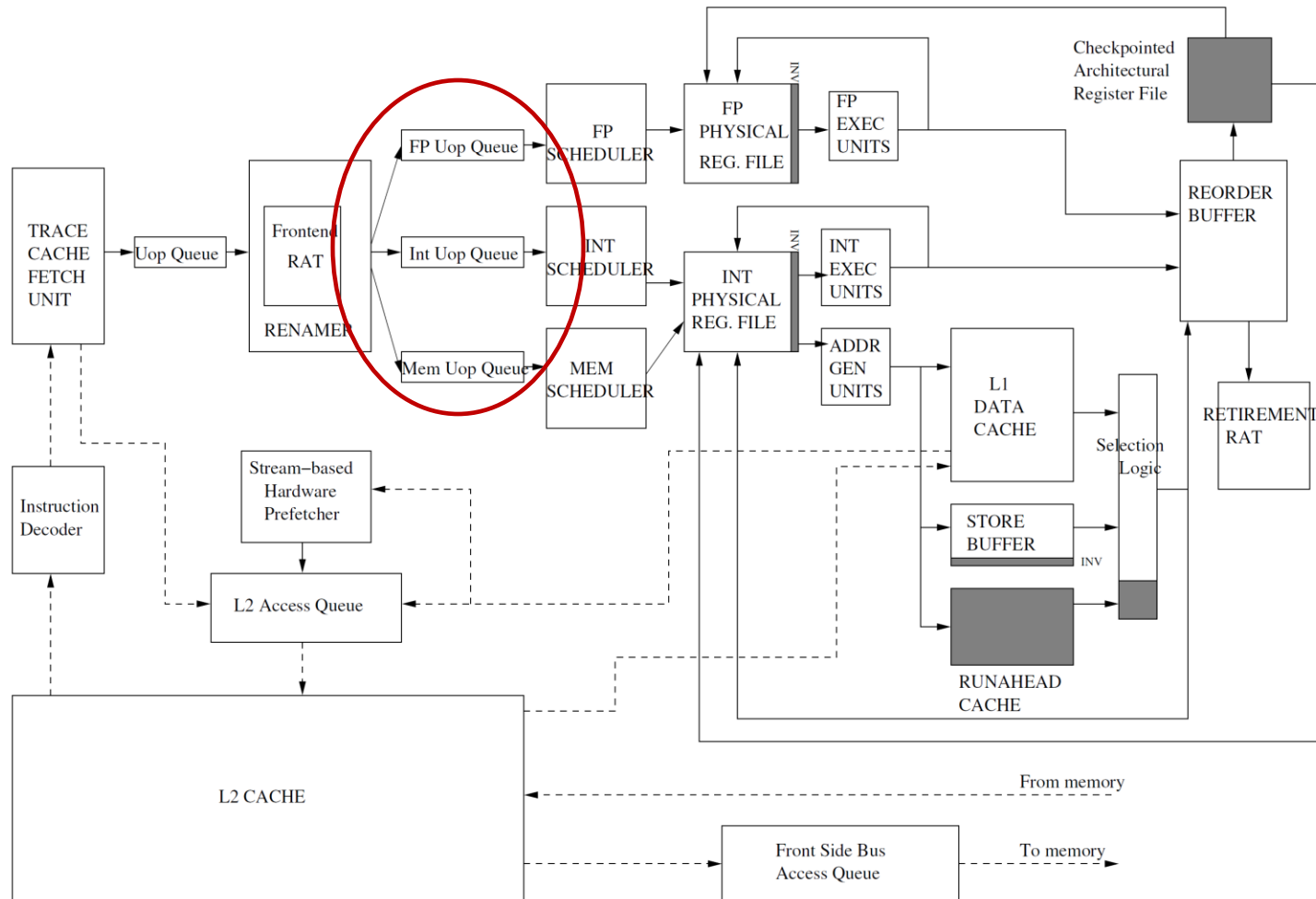
- Dynamic scheduling (**hardware, runtime**)
 - A and X streams are issued/executed independently
 - Loads can bypass stores in the memory unit (if no conflict)
 - Branches executed early in the pipeline
 - To reduce synchronization penalty of A/X streams
 - Works only if the register that a branch sources is available
- Static scheduling (**compiler**)
 - Move compare instructions as early as possible before a branch
 - So that branch source register is available when branch is decoded
 - Reorder code to expose parallelism in each stream
 - Loop unrolling:
 - Reduces branch count + exposes code reordering opportunities

A Modern DAE Example: Pentium 4



Intel Pentium 4 Simplified

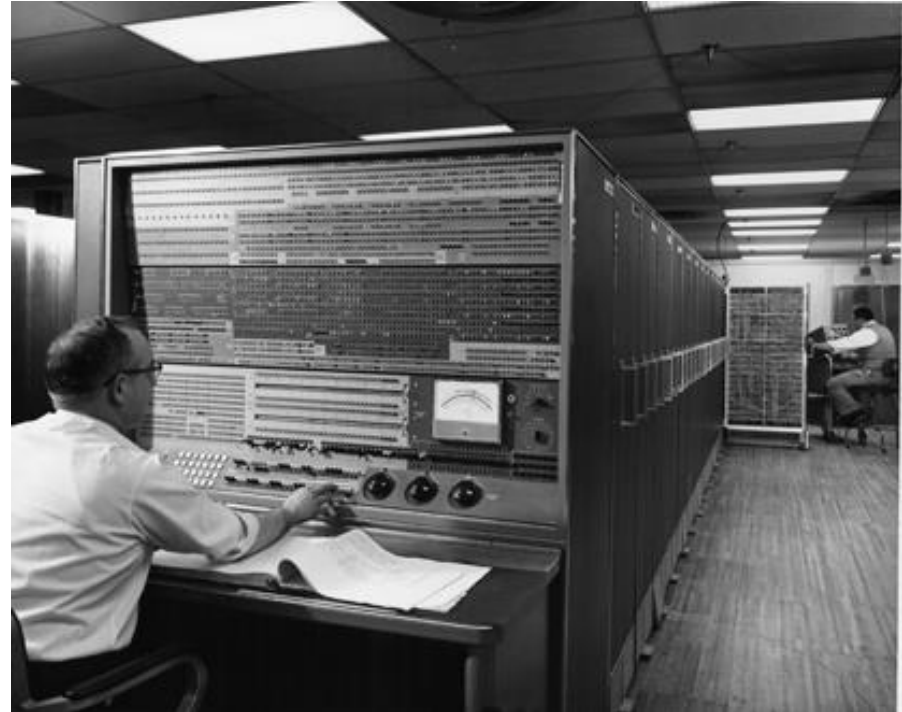
Onur Mutlu et al. “Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-order Processors”, in Proceedings of HPCA 2003.



IBM 7030 "Stretch" (1954-1961)



- Original goal was to use new transistor technology to give **100x** performance of tube-based IBM 704.
- Design based around 4 stages of "lookahead" pipelining
- More than just pipelining, a simple form of decoupled execution with indexing and branch operations performed speculatively ahead of data operations
- Also had a simple store buffer
 - Very complex design for the time, difficult to explain to users performance of pipelined machine
 - When finally delivered in 1961, was benchmarked at only **30x** 704 and embarrassed IBM, causing price to drop from \$13.5M to \$7.8M, and withdrawal after initial deliveries
 - But technologies lived on in later IBM computers, 360 and POWER





Supercomputers

Definitions of a supercomputer:

- Fastest machine in world at given task
- A device to turn a compute-bound problem into an I/O bound problem
- Any machine costing \$30M+
- Any machine designed by Seymour Cray
- CDC6600 (Cray, 1964) regarded as first supercomputer

CDC 6600 *Seymour Cray, 1964*



Scoreboarding is a technique for allowing instructions to execute out of order when there are sufficient resources and no data dependences

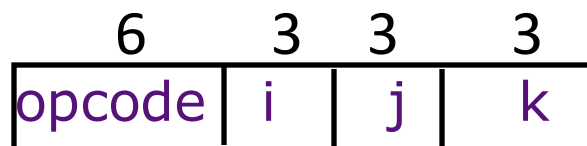
- A fast pipelined machine with 60-bit words
 - 128 Kword main memory capacity, 32 banks
- Ten functional units (parallel, unpipelined)
 - Floating Point: adder, 2 multipliers, divider
 - Integer: adder, 2 incrementers, ...
- Hardwired control (no microcoding)
- *Scoreboard* for dynamic scheduling of instructions
- Ten Peripheral Processors for Input/Output
 - a fast multi-threaded 12-bit integer ALU
- Very fast clock, 10 MHz (FP add in 4 clocks)
- >400,000 transistors, 750 sq. ft., 5 tons, 150 kW, novel freon-based technology for cooling
- Fastest machine in world for 5 years (until 7600)
 - over 100 sold (\$7-10M each)



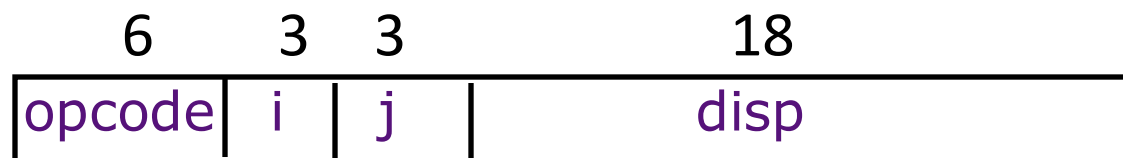
CDC 6600:

A Load/Store Architecture

- Separate instructions to manipulate three types of reg.
 - 8x60-bit data registers (X)
 - 8x18-bit address registers (A)
 - 8x18-bit index registers (B)
- All arithmetic and logic instructions are register-to-register


$$R_i \leftarrow R_j \text{ op } R_k$$

- Only Load and Store instructions refer to memory!

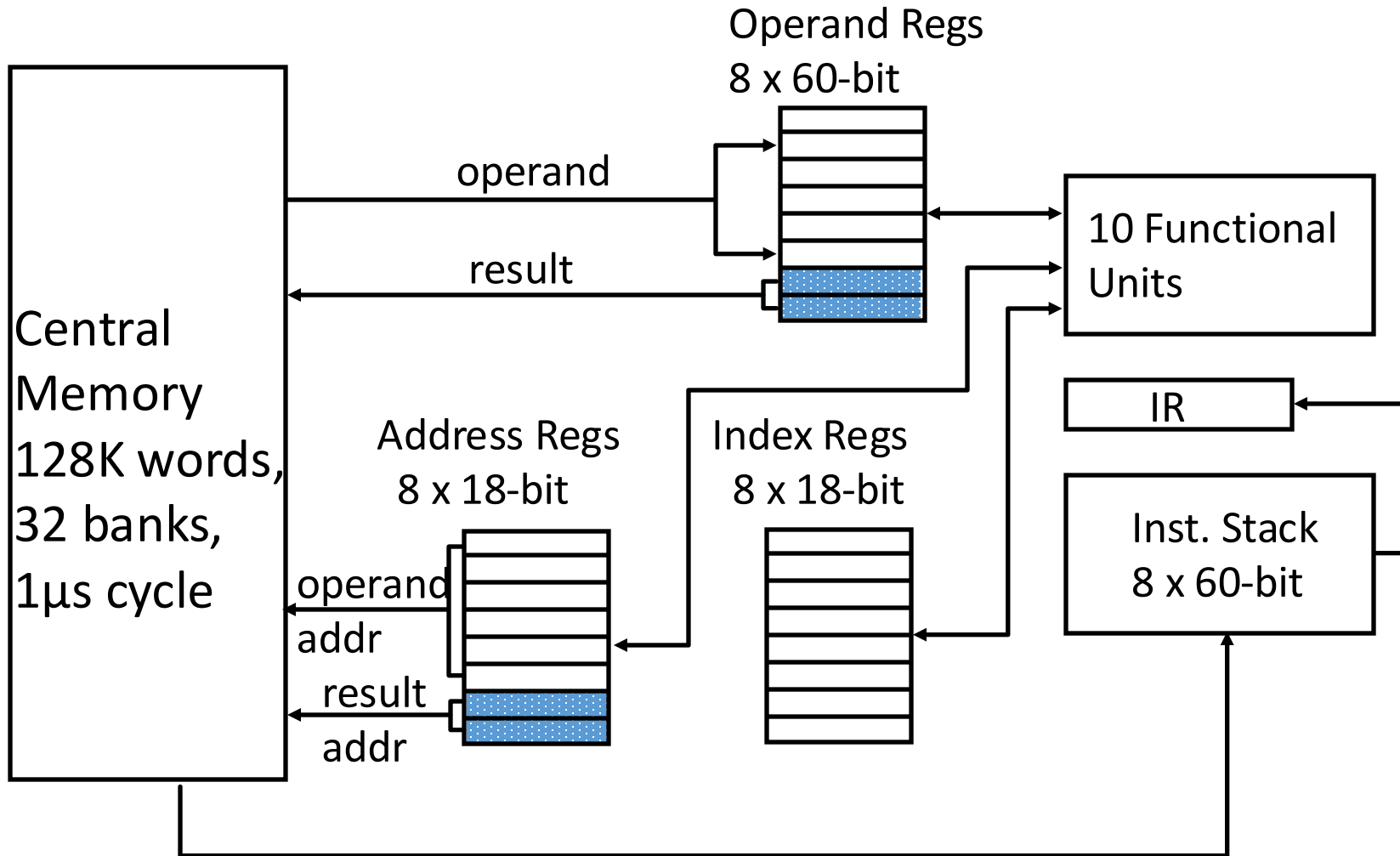

$$R_i \leftarrow M[R_j + \text{disp}]$$

Touching address registers 1 to 5 initiates a load

6 to 7 initiates a store

- *very useful for vector operations*

CDC 6600: Datapath





CDC 6600: Vector Addition

```
      B0 ← - n
loop: JZE  B0, exit
      A0 ← B0 + a0      load X0
      A1 ← B0 + b0      load X1
      X6 ← X0 + X1
      A6 ← B0 + c0      store X6
      B0 ← B0 + 1
      jump loop
```

A_i = address register

B_i = index register

X_i = data register



CDC 6600 ISA designed to simplify high-performance implementation

- Use of three-address, register-register ALU instructions simplifies pipelined implementation
 - Only 3-bit register-specifier fields checked for dependencies
 - No implicit dependencies between inputs and outputs
- Decoupling setting of address register (Ar) from retrieving value from data register (Xr) simplifies providing multiple outstanding memory accesses
 - Address update instruction also issues implicit memory operation
 - Software can schedule load of address register before use of value
 - Can interleave independent instructions in between
- CDC6600 has multiple parallel *unpipelined* functional units
 - E.g., 2 separate multipliers
- Follow-on machine CDC7600 used pipelined functional units
 - Foreshadows later RISC designs

CDC 6600 ISA designed to simplify high-performance implementation

- Use of three-address, register-register ALU instructions simplifies pipelined implementation
 - Only 3-bit register-specifier fields checked for dependencies
 - No implicit dependencies between inputs and outputs
- Decoupling setting of address register (Ar) from retrieving value from data register (Xr) simplifies providing multiple outstanding memory accesses

opcode	i	j	k
--------	---	---	---

$R_i \leftarrow R_j \text{ op } R_k$

 - Address update instruction also issues implicit memory operation
 - Software can schedule load of address register before use of value
 - Can interleave independent instructions in between
- CDC6600 has multiple parallel *unpipelined* functional units
 - E.g., 2 separate multipliers
- Follow-on machine CDC7600 used pipelined functional units
 - Foreshadows later RISC designs



MEMORANDUM

August 28, 1963

Memorandum To: Messrs. A. L. Williams
T. V. Learson
H. W. Miller, Jr.
E. R. Piore
O. M. Scott
M. B. Smith
A. K. Watson

Last week CDC had a press conference during which they officially announced their 6600 system. I understand that in the laboratory developing this system there are only 34 people, "including the janitor." Of these, 14 are engineers and 4 are programmers, and only one person has a Ph.D., a relatively junior programmer. To the outsider, the laboratory appeared to be cost conscious, hard working and highly motivated.

Contrasting this modest effort with our own vast development activities, I fail to understand why we have lost our industry leadership position by letting someone else offer the world's most powerful computer. At Jenny Lake, I think top priority should be given to a discussion as to what we are doing wrong and how we should go about changing it immediately.

TJW, Jr:jmc

T. J. Watson, Jr.

cc: Mr. W. B. McWhirter

[© IBM]

IBM Memo on CDC 6600

Thomas Watson Jr., IBM CEO, August 1963:

“Last week, CDC ... announced the 6600 system. I understand that in the laboratory developing the system there are only 34 people including the janitor. Of these, 14 are engineers and 4 are programmers... Contrasting this modest effort with our vast development activities, I fail to understand why we have lost our industry leadership position by letting someone else offer the world's most powerful computer.”

To which Cray replied: *“It seems like Mr. Watson has answered his own question.”*

The interaction between pipelining and instruction set design was understood, and the instruction set was kept simple to promote pipelining.

https://www.elsevier.com/_data/assets/pdf_file/0010/297496/Section-4-16_Hist-Persp.pdf



Computer Architecture Terminology

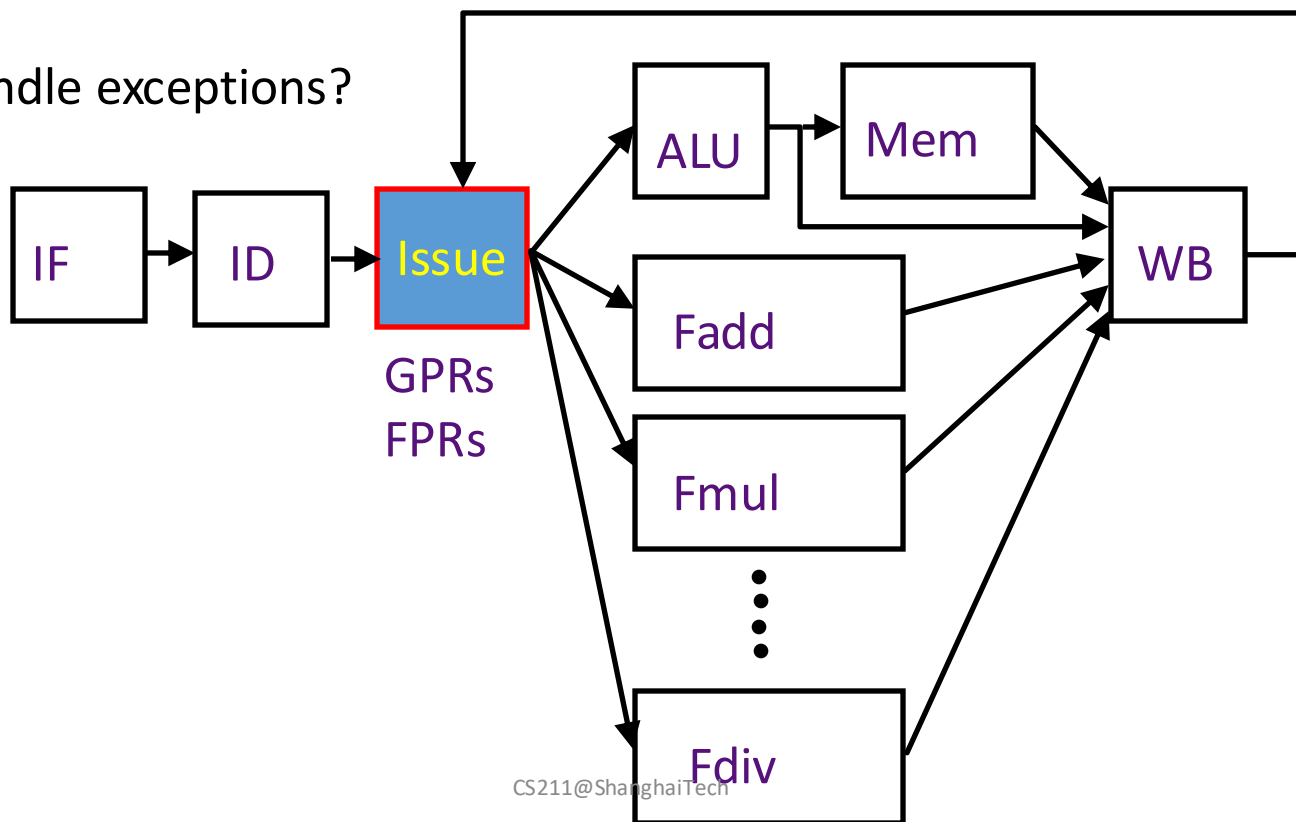
Latency (in seconds or cycles): Time taken for a single operation from start to finish (initiation to usable result)

Bandwidth (in operations/second or operations/cycle): Rate of which operations can be performed

Occupancy (in seconds or cycles): Time during which the unit is blocked on an operation (structural hazard)

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
- How to handle exceptions?





CDC6600 Scoreboard

If a functional unit for the instruction is free and no other active instruction has the same destination register (WAW), the scoreboard issues the instruction to the functional unit and updates its internal data structure. If a structural or WAW hazard exists, then the instruction issue stalls, and no further instructions will issue until these hazards are cleared.

- Scoreboard keeps track of

- Instruction status
- Functional unit status
- Register result status

When the source operands are available, the scoreboard tells the functional unit to proceed to read the operands from the registers and begin execution. The scoreboard resolves RAW hazards dynamically in this step, and instructions may be sent into execution out of order.

- Instructions dispatched in-order to functional units provided no structural hazard or WAW

- Stall on structural hazard, no functional units available
- Only one pending write to any register

- Instructions wait for input operands (RAW hazards) before execution

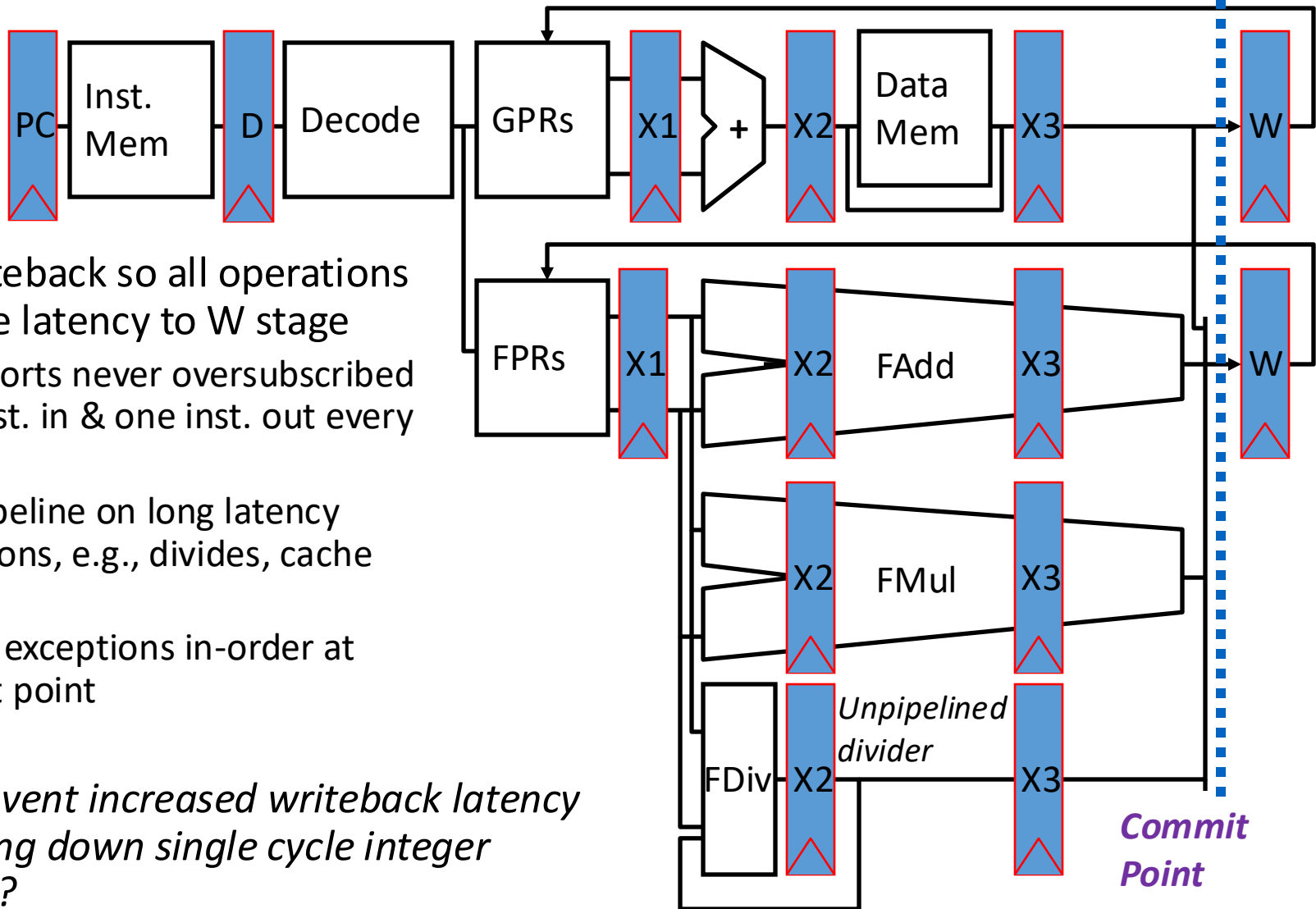
- Can execute out-of-order

- Instructions wait for output register to be read by preceding instructions (WAR)

- Result held in functional unit register free

Once the scoreboard is aware that the functional unit has completed execution, the scoreboard checks for WAR hazards. If none, it writes results. If WAR, then it stalls the instruction

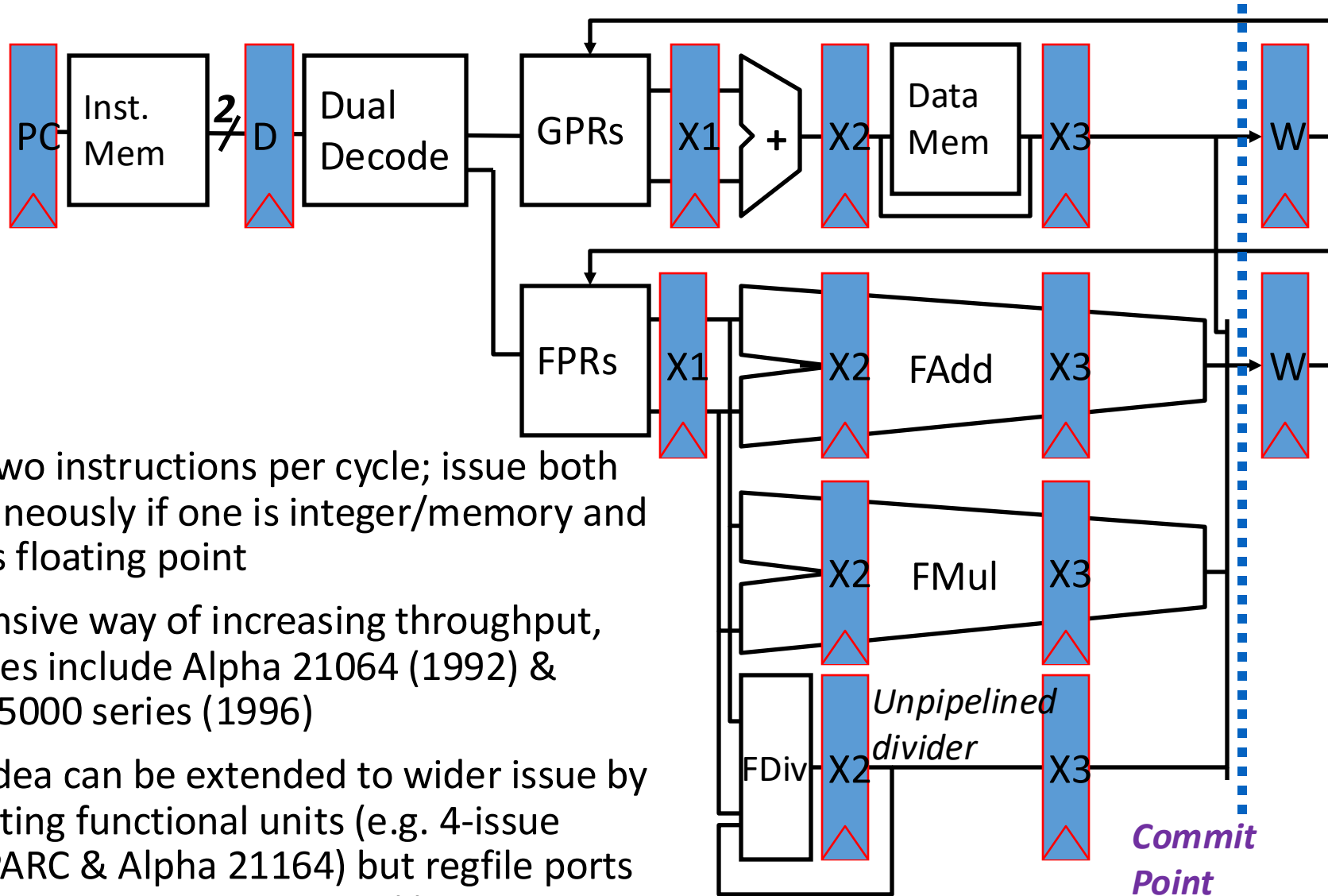
More Complex In-Order Pipeline



- 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
 - Handle exceptions in-order at commit point

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

In-Order Superscalar Pipeline



*Commit
Point*

- 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

In-Order Pipeline with two ALU stages

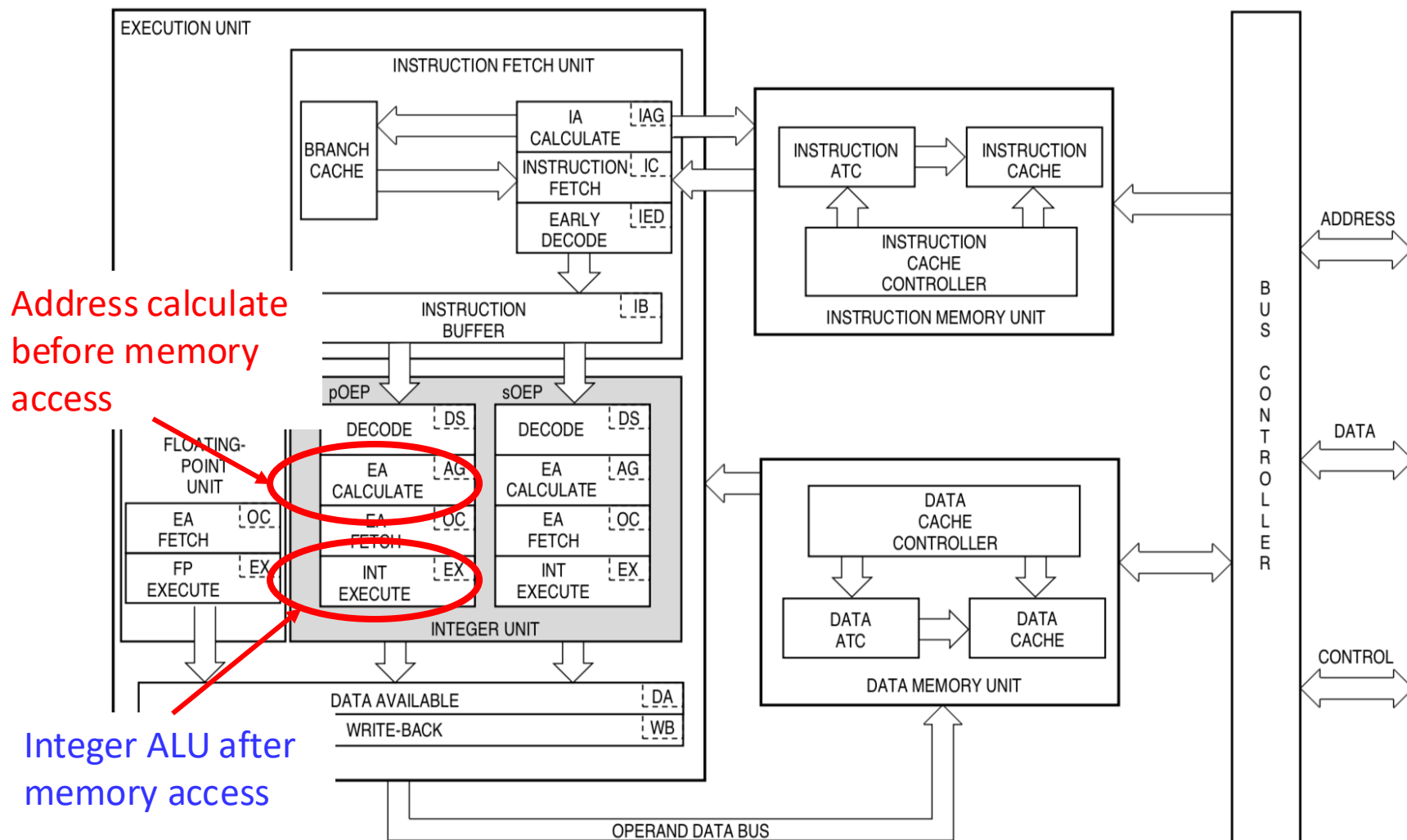
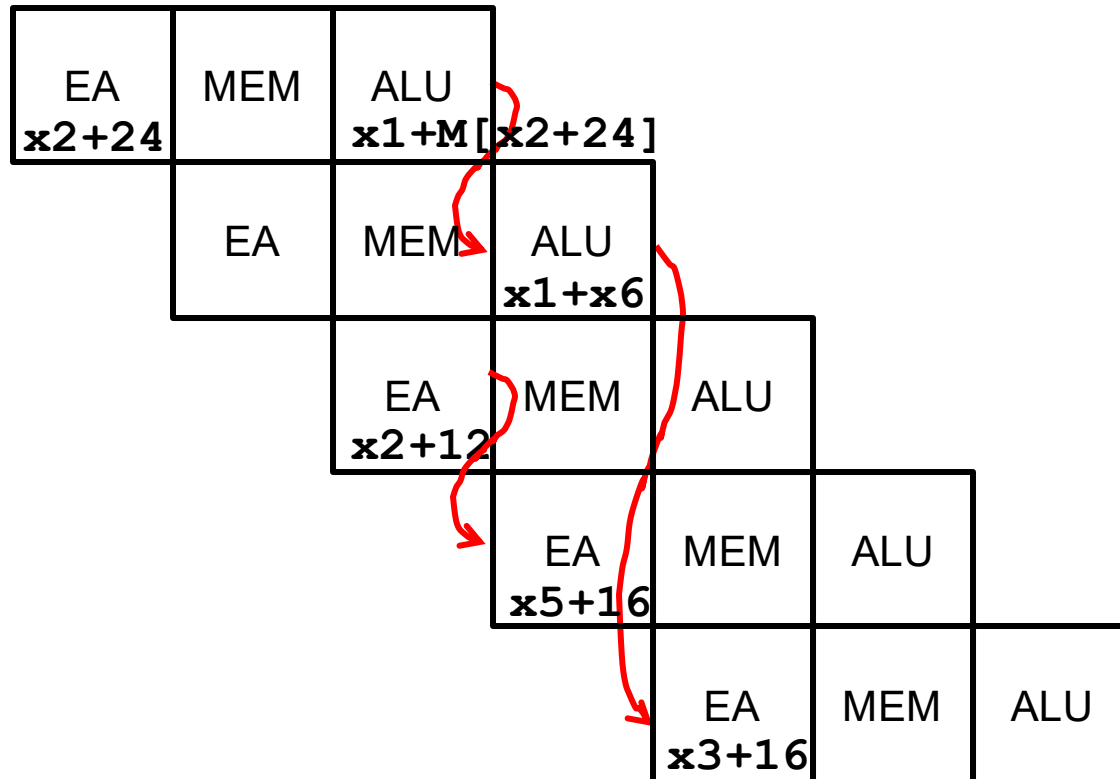


Figure 3-1. MC68060 Integer Unit Pipeline

MC68060 Dynamic ALU Scheduling

Using RISC-V style assembly code for MC68060



`add x1,x1,24(x2)`

`add x3,x1,x6`

`addi x5,x2,12`

`lw x4, 16(x5)`

`lw x8, 16(x3)`

Not a real RISC-V instruction!

Common trick used in modern in-order RISC pipeline designs, even without reg-mem operations



Conclusion

- Advanced topics in Pipelining
- With several classic examples



Acknowledgements

- These slides contain materials developed and copyright by:
 - Prof. Krste Asanovic (UC Berkeley)
 - Prof. Onur Mutlu (ETH Zurich)
 - Prof. Daniel J. Sorin (Duke)
 - Prof. Andreas Moshovos (U. of Toronto)