CS 152 Computer Architecture and Engineering
CS252 Graduate Computer Architecture

Lecture 4 – Pipelining Part II

Krste Asanovic
Electrical Engineering and Computer Sciences
University of California at Berkeley

http://www.eecs.berkeley.edu/~krste
http://inst.eecs.berkeley.edu/~cs152

Last Time in Lecture 3

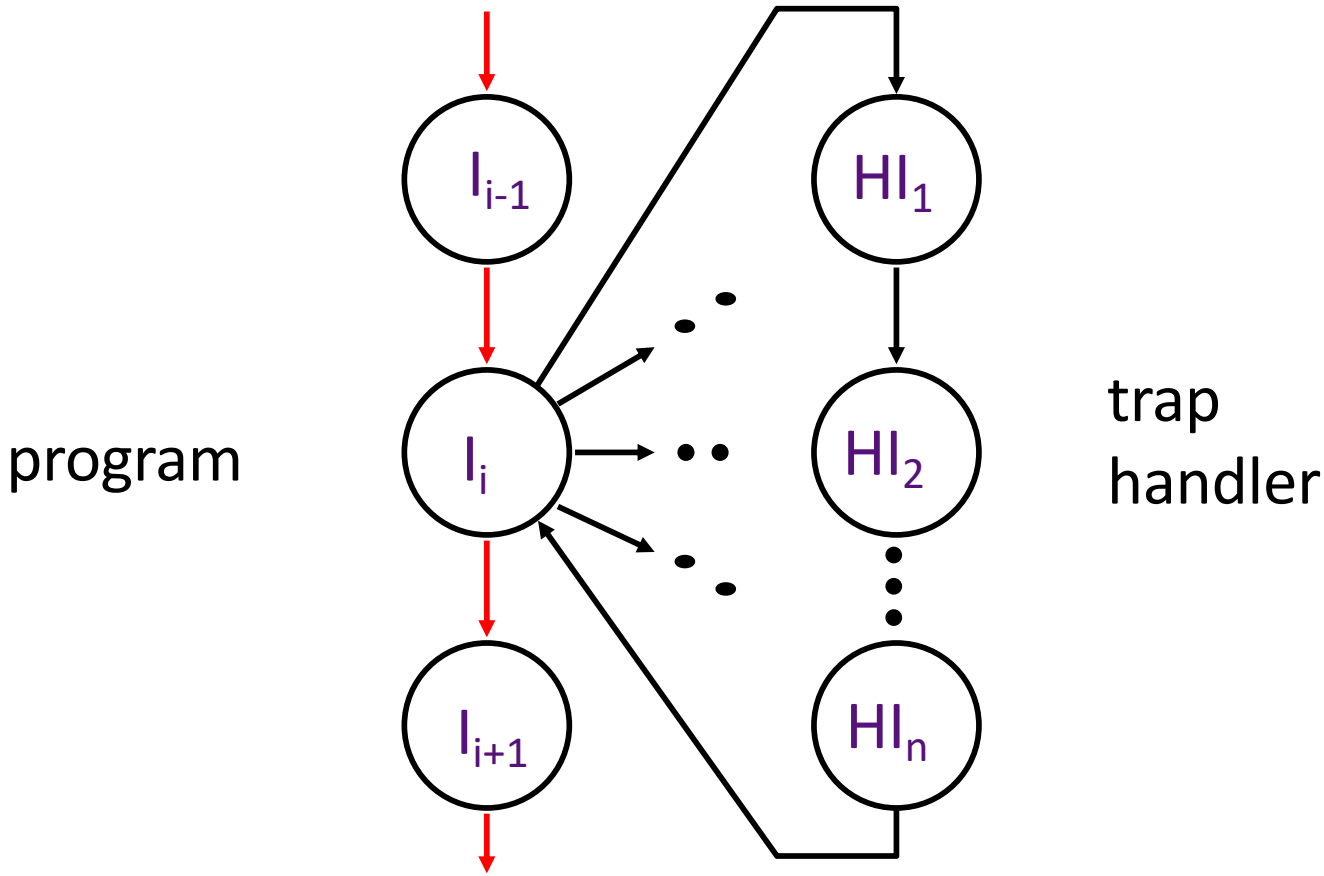
- Iron law of performance:
  - time/program = insts/program * cycles/inst * time/cycle
- Classic 5-stage RISC pipeline
- Structural, data, and control hazards
- Structural hazards handled with interlock or more hardware
- Data hazards include RAW, WAR, WAW
  - Handle data hazards with interlock, bypass, 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

Asynchronous Interrupts

- An I/O device requests attention by asserting one of the *prioritized interrupt request lines*

- When the processor decides to process the interrupt
  - It stops the current program at instruction I_i , completing all the instructions up to I_{i-1} *(precise interrupt)*
  - It saves the PC of instruction I_i in a special register (EPC)
    - *EPC = Exception Program Counter,* but also used for traps cause by interrupts
  - Saves reason for interrupt in special *Cause* register, so handler can determine what to do
  - Disables interrupts and transfers control to a designated interrupt handler running in supervisor mode

Trap:
altering the normal flow of control



An *external or internal event* that needs to be processed by another (system) program. The event is usually unexpected or rare from program's point of view.

Trap Handler

- Saves ***EPC*** before enabling interrupts to allow nested interrupts $\Rightarrow$
  - need an instruction to move EPC into GPRs
  - need a way to mask further interrupts at least until EPC can be saved
- Needs to read the ***Cause*** *register* that indicates the reason for the trap
- Uses a special indirect jump instruction **ERET** (*return-from-environment*) which
  - enables interrupts
  - restores the processor to the user mode
  - restores hardware status and control state
  - sets PC to the EPC value, and resumes execution

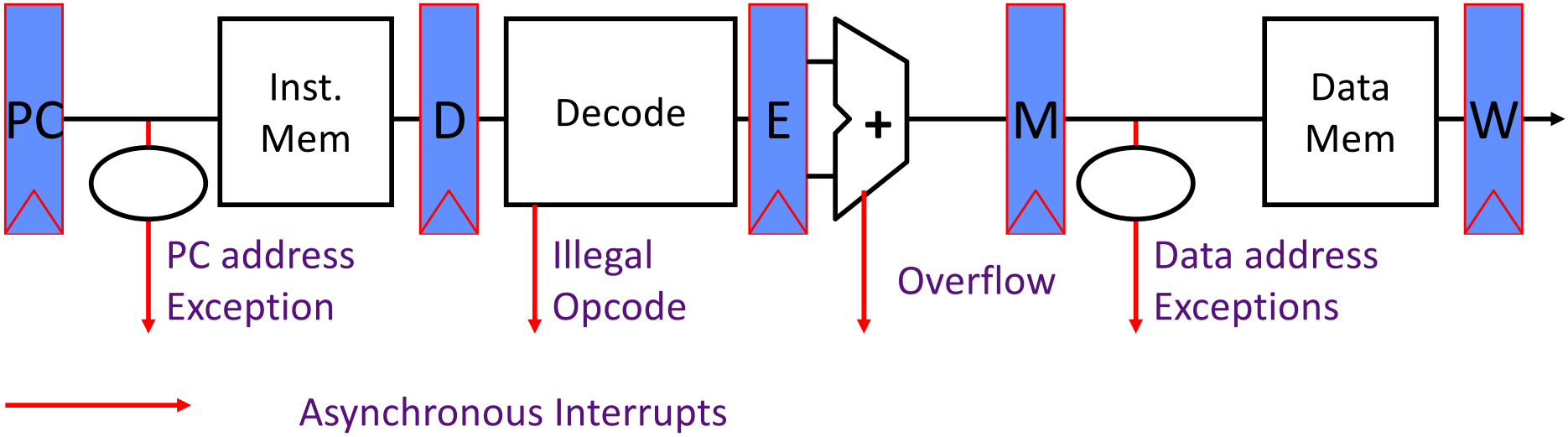
Synchronous Trap

- A synchronous trap is caused by an exception on a *particular instruction*

- In general, the instruction cannot be completed and needs to be *restarted* after the exception has been handled
  - May require undoing the effect of one or more partially executed instructions in microarchitecture

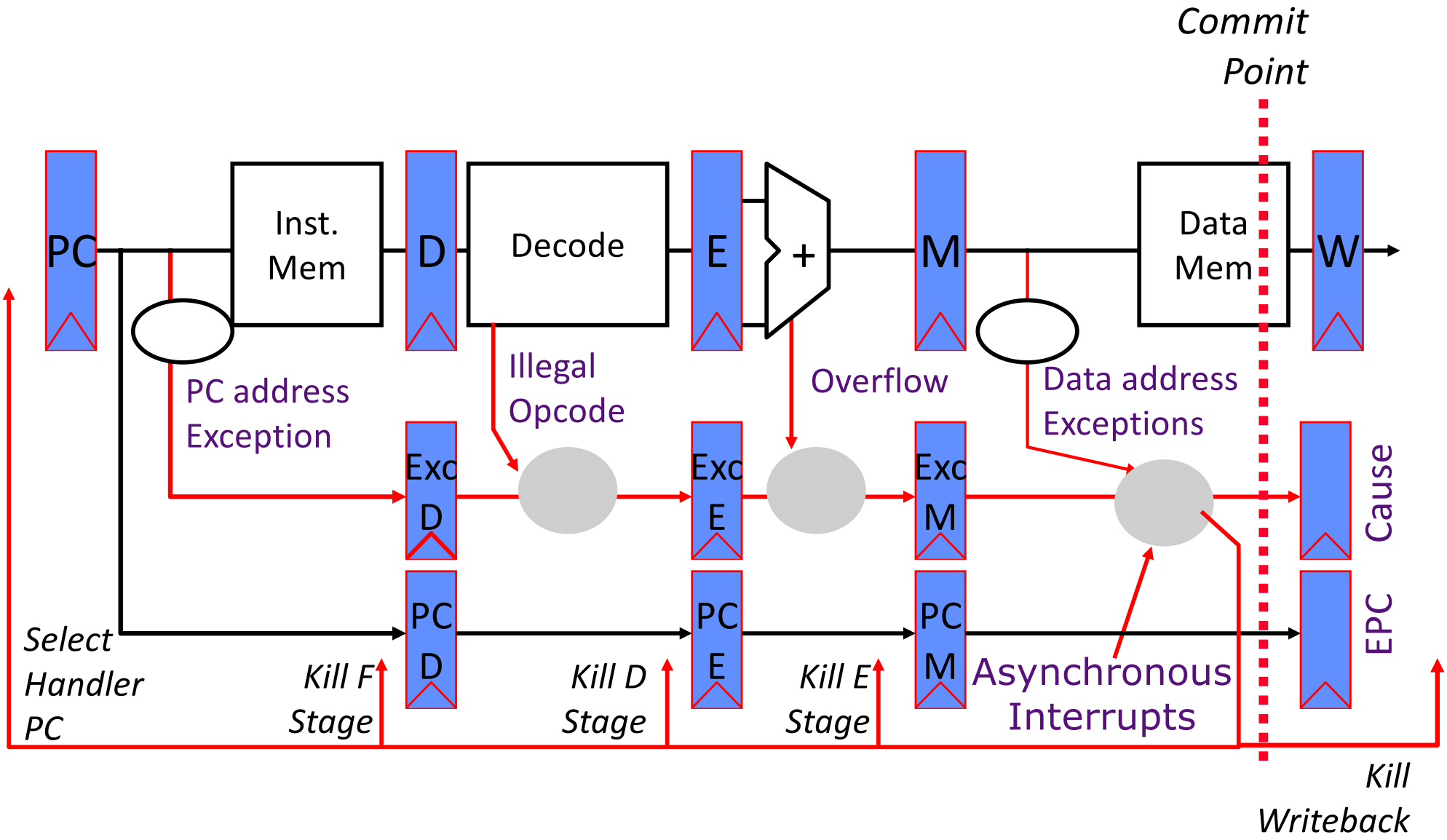
- In the case of a system call trap, the instruction is considered to have been completed
  - In RISC-V, a special ECALL instruction causes a trap into a higher-privilege mode

Exception Handling 5-Stage Pipeline



- How to handle multiple simultaneous exceptions in different pipeline stages?
- How and where to handle external asynchronous interrupts?

Exception Handling 5-Stage Pipeline



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*
- 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

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

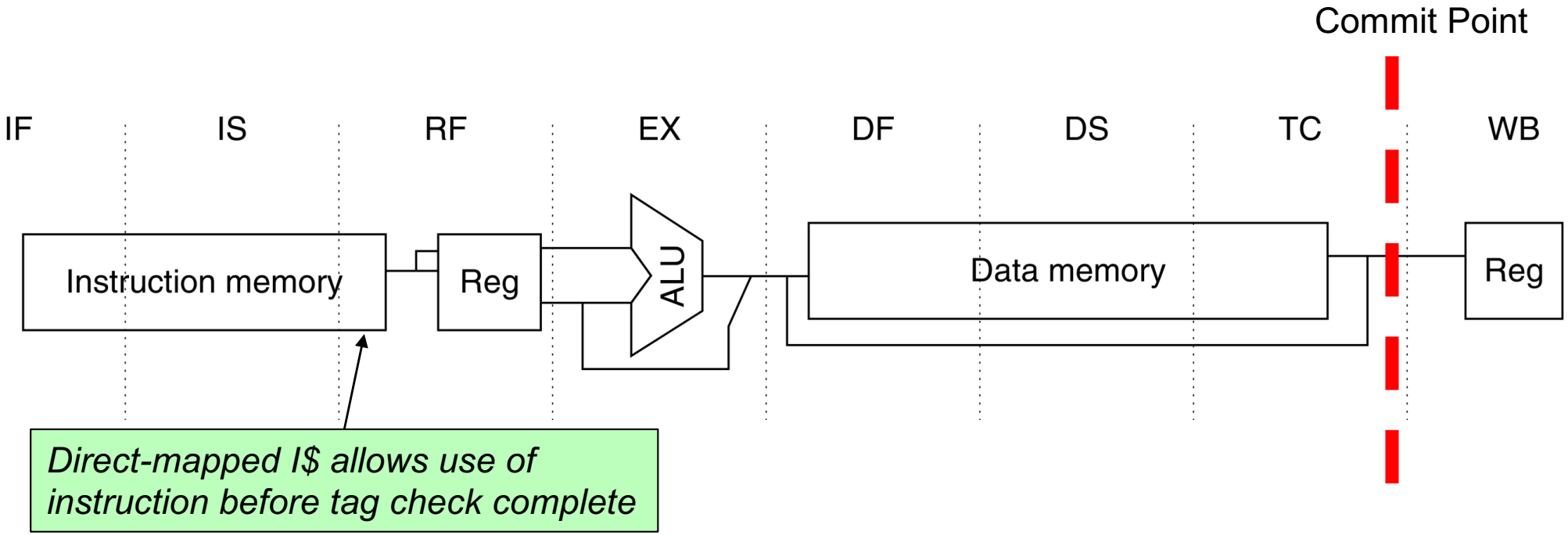


Figure C.36 The eight-stage pipeline structure of the R4000 uses pipelined instruction and data caches. The pipe stages are labeled and their detailed function is described in the text. The vertical dashed lines represent the stage boundaries as well as the location of pipeline latches. The instruction 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

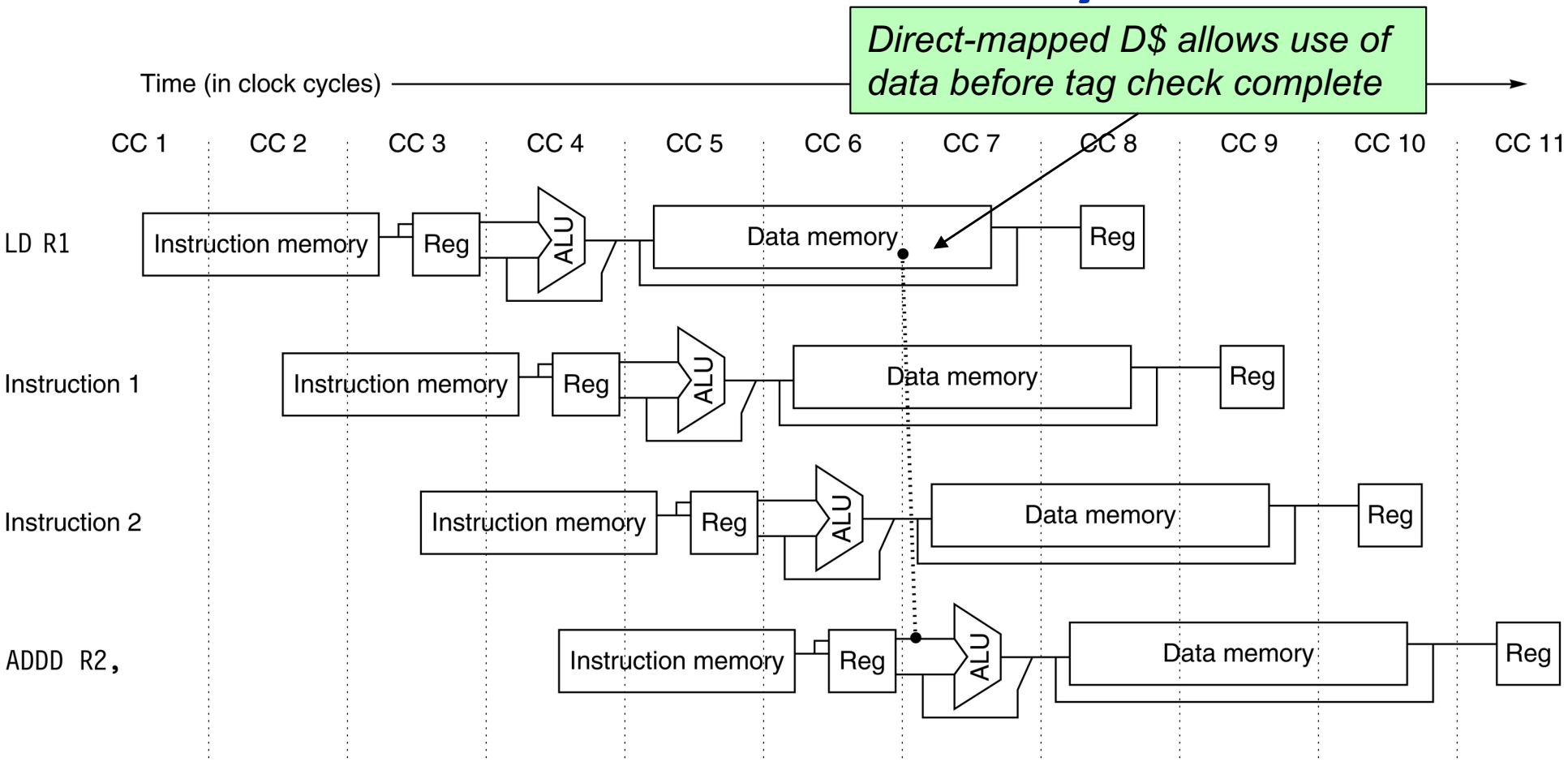


Figure C.37 The structure of the R4000 integer pipeline leads to a x1 load delay. A x1 delay is possible because the data value is available at the end of DS and can be bypassed. If the tag check in TC indicates a miss, the pipeline is backed up a cycle, when the correct data are available.

R4000 Branches

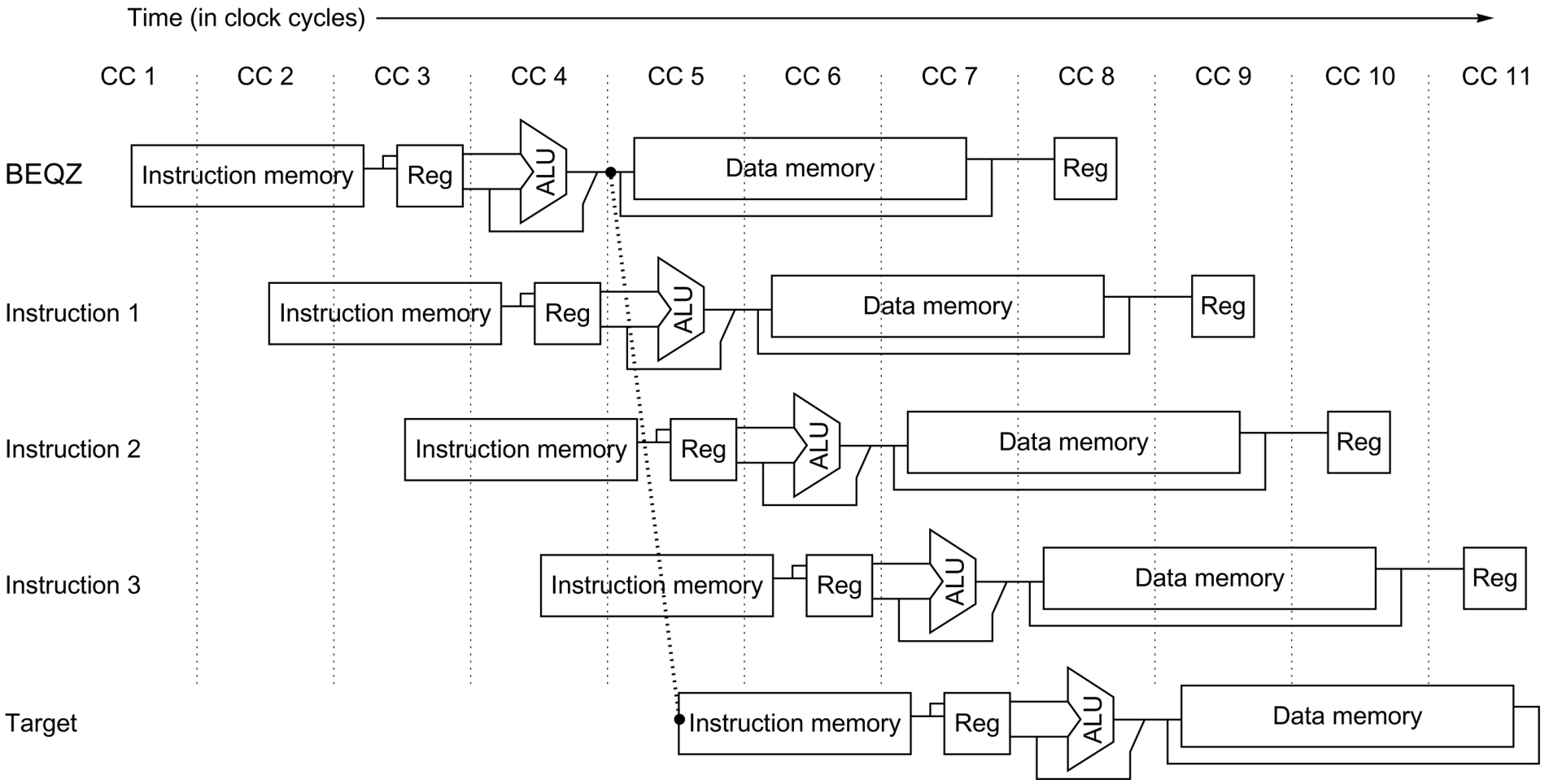


Figure C.39 The basic branch delay is three cycles, because the condition evaluation is performed during EX.

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

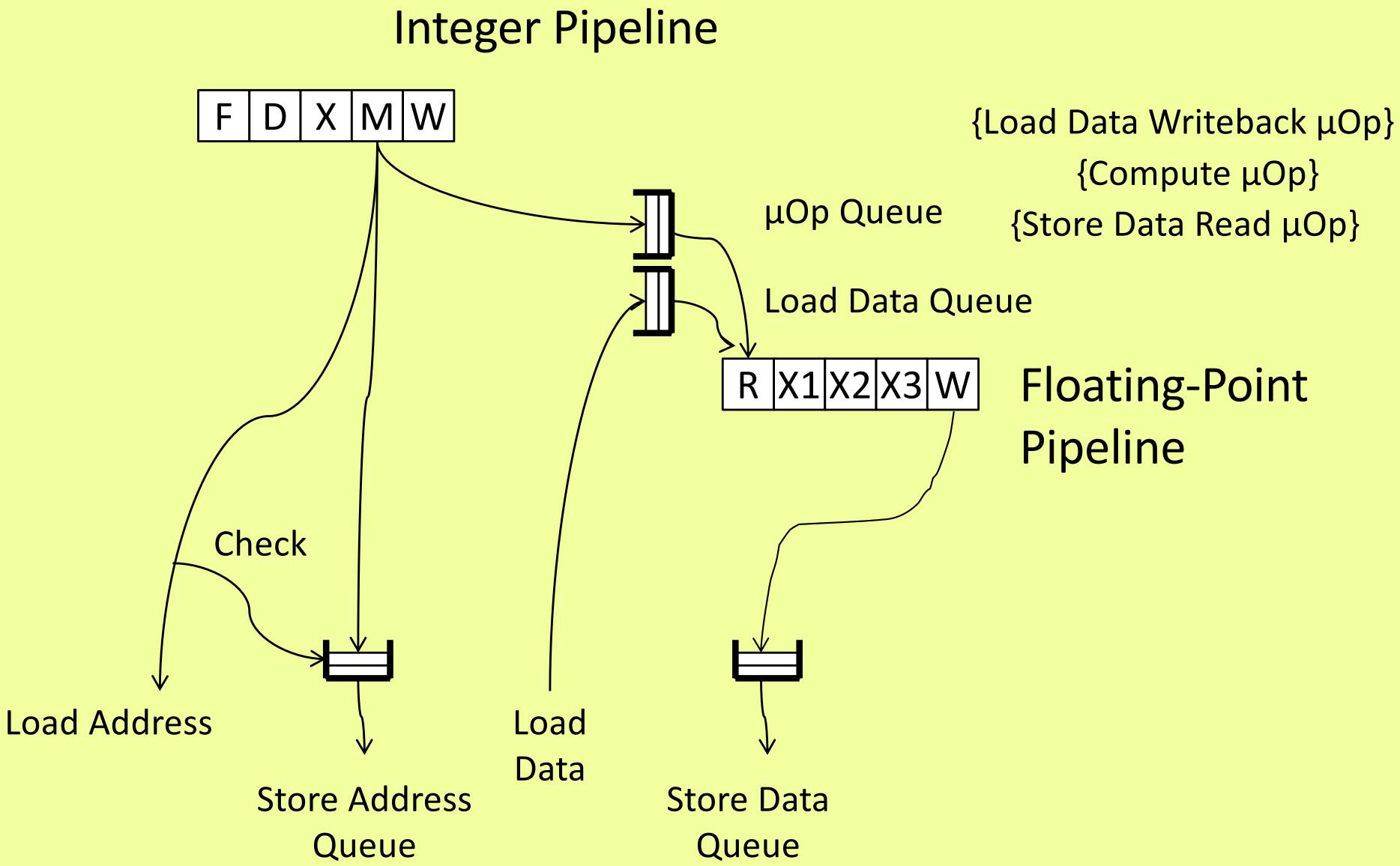
Alternative Approach: Decoupling *(lookahead, runahead)* in µarchitecture

Can separate control and memory address operations from data computations:

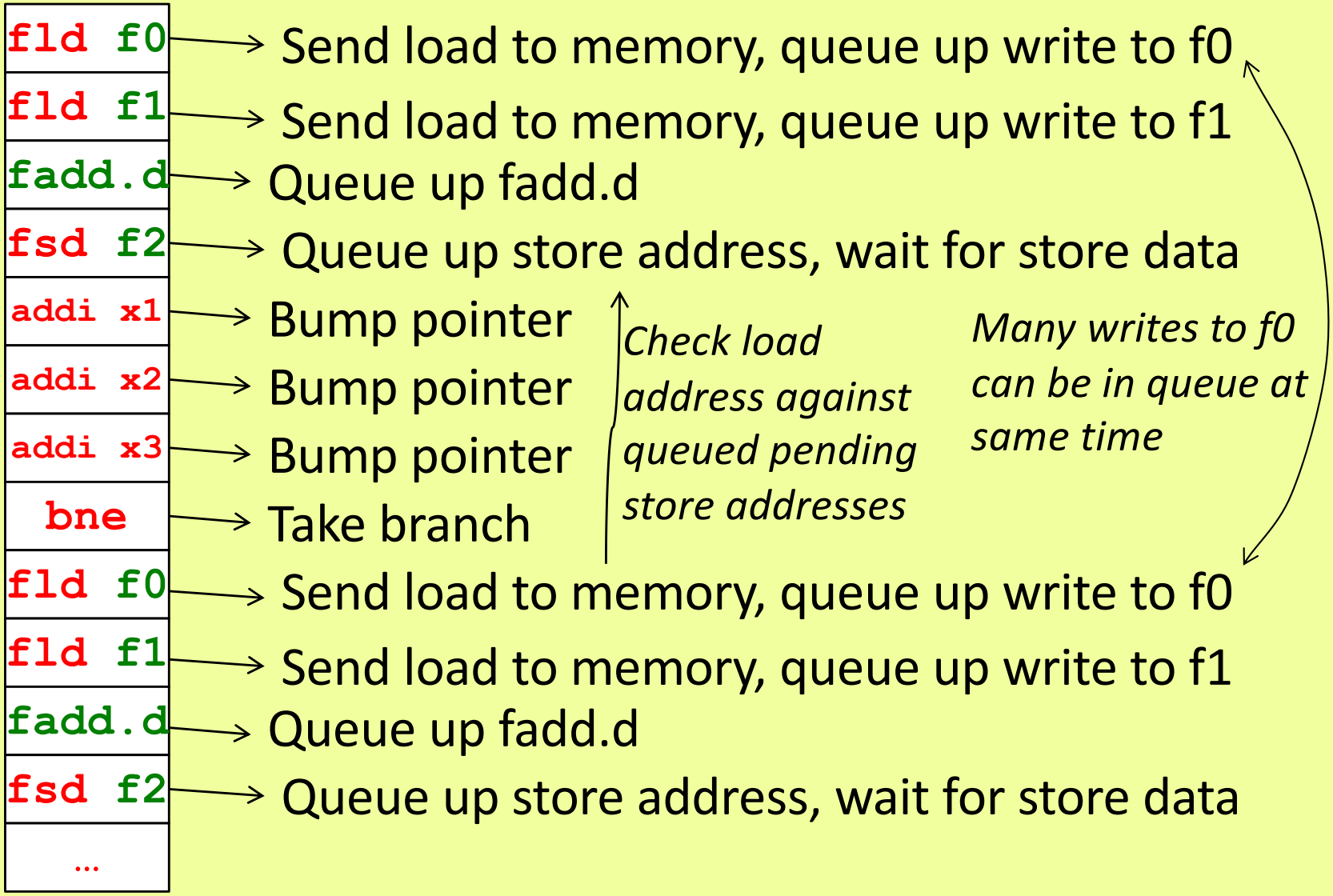
```
loop: fld f0, 0(x2) // x2 points to B
      fld f1, 0(x3) // x3 points to C
      fadd.d f2, f0, f1
      fsd f2, 0(x1) // x1 points to A
      addi x1,x1,8  // Bump pointer
      addi x2,x2,8  // Bump pointer
      addi x3,x3,8  // Bump pointer
      bne x1, x4, loop // x4 holds end
```

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.

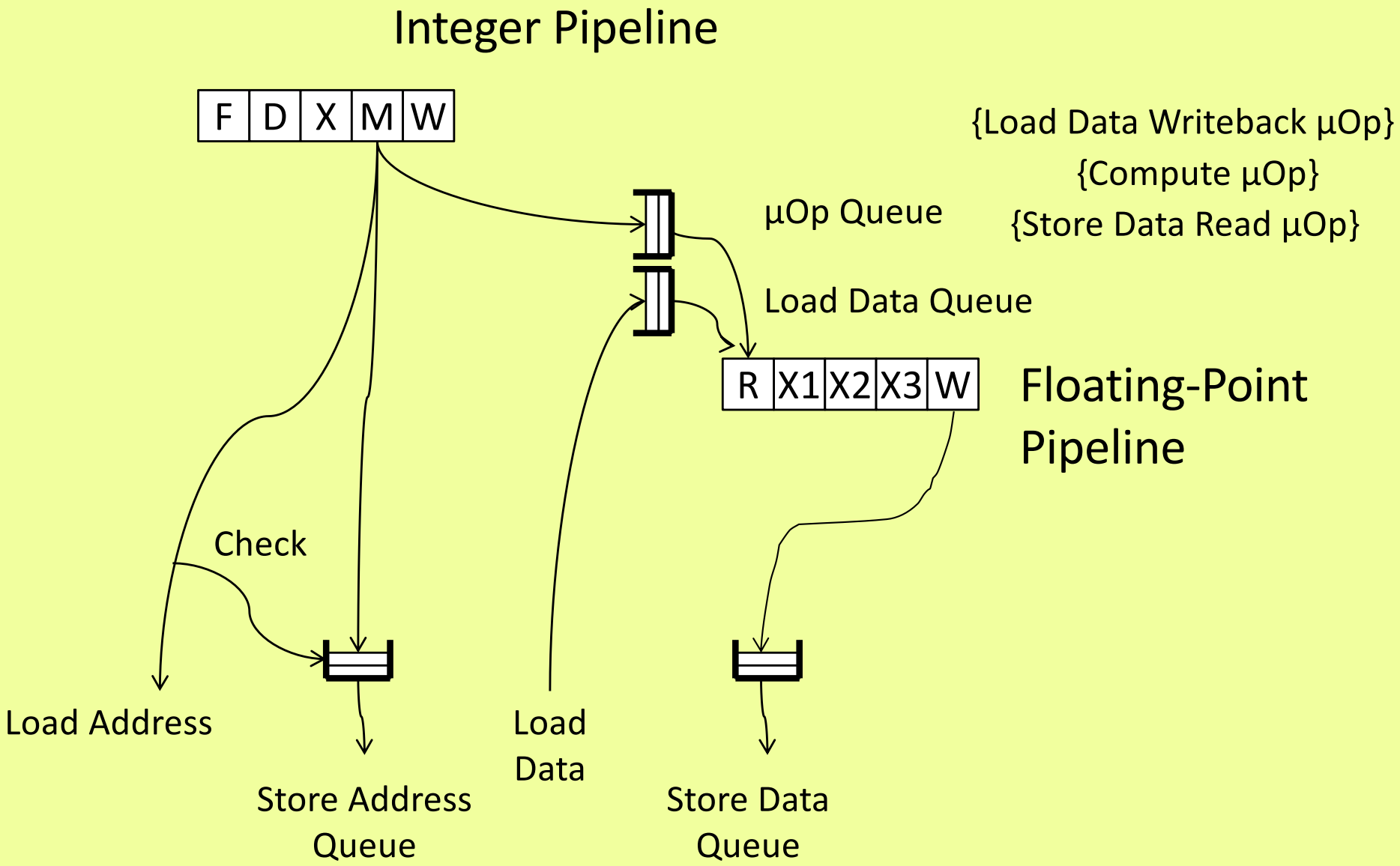
Simple Decoupled Machine



Decoupled Execution



Simple Decoupled Machine



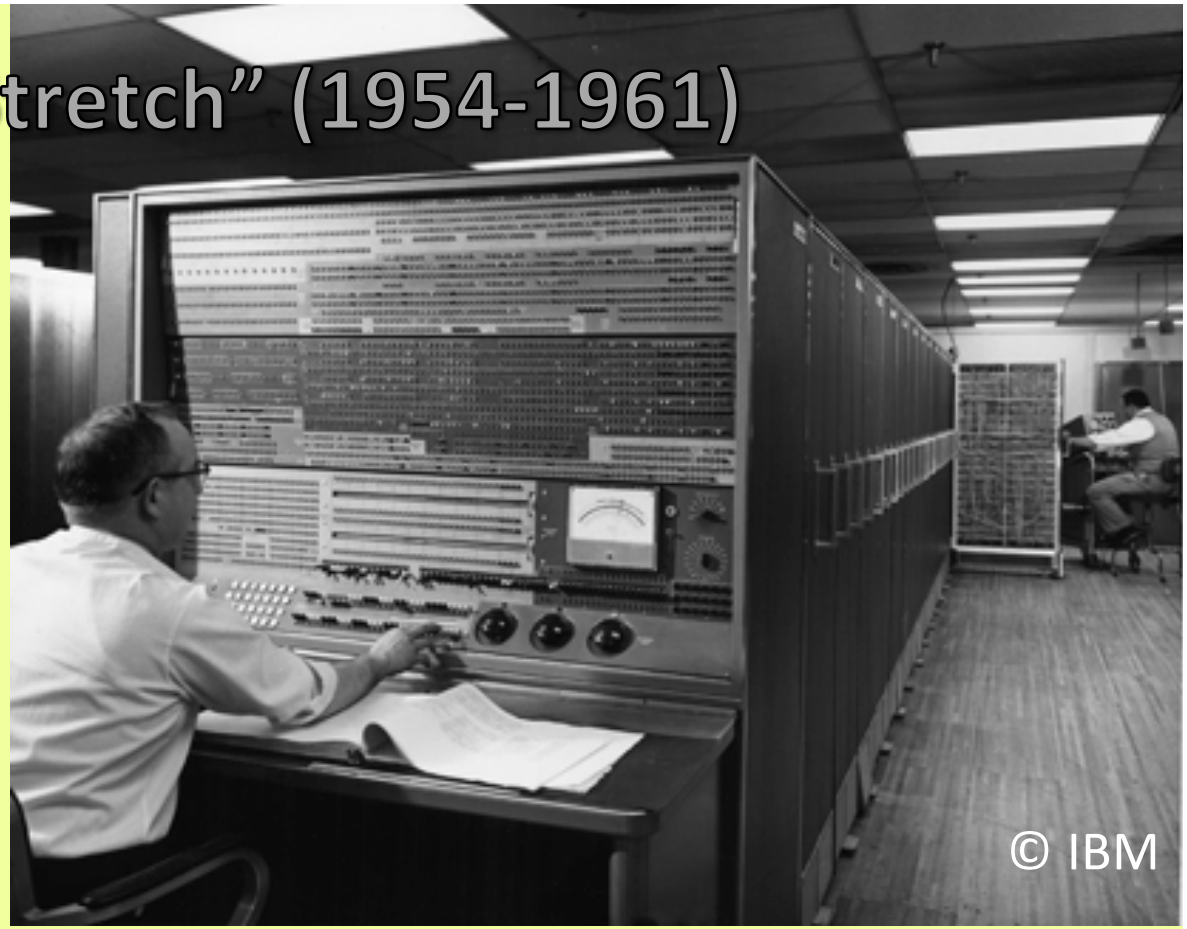
CS152 Administrivia

- PS 1 due 11:59PM on Monday Feb 8
- Lab 1 due 11:59PM Wed Feb 17

CS252 Administrivia

- Project proposals due 11:59PM Wed Feb 26th
- Use Krste's office hours Tue 10-11am to get feedback on ideas
  - email for link
- Readings discussion will be Thursdays 5-6pm
  - zoom link on Piazza
  - Questions on Piazza

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

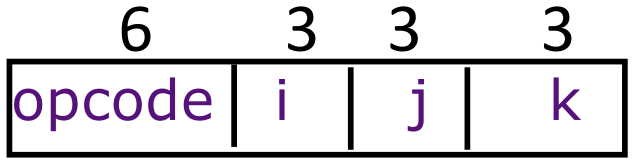
CDC 6600 *Seymour Cray, 1964*

- 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

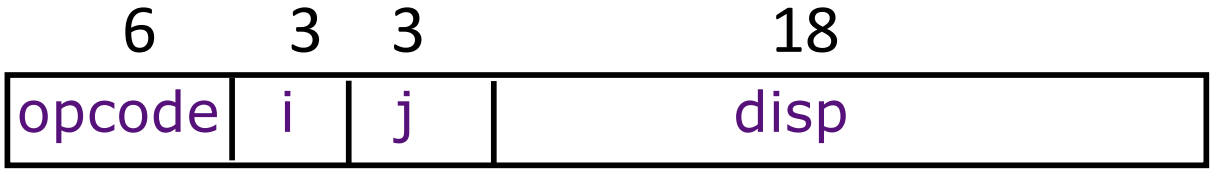
- Separate instructions to manipulate three types of reg.
  - 8x60-bit data registers (X)
  - 8x18-bit address registers (A)
  - 8x18-bit index registers (B)

- All arithmetic and logic instructions are register-to-register



Ri ← Rj op Rk

- Only Load and Store instructions refer to memory!



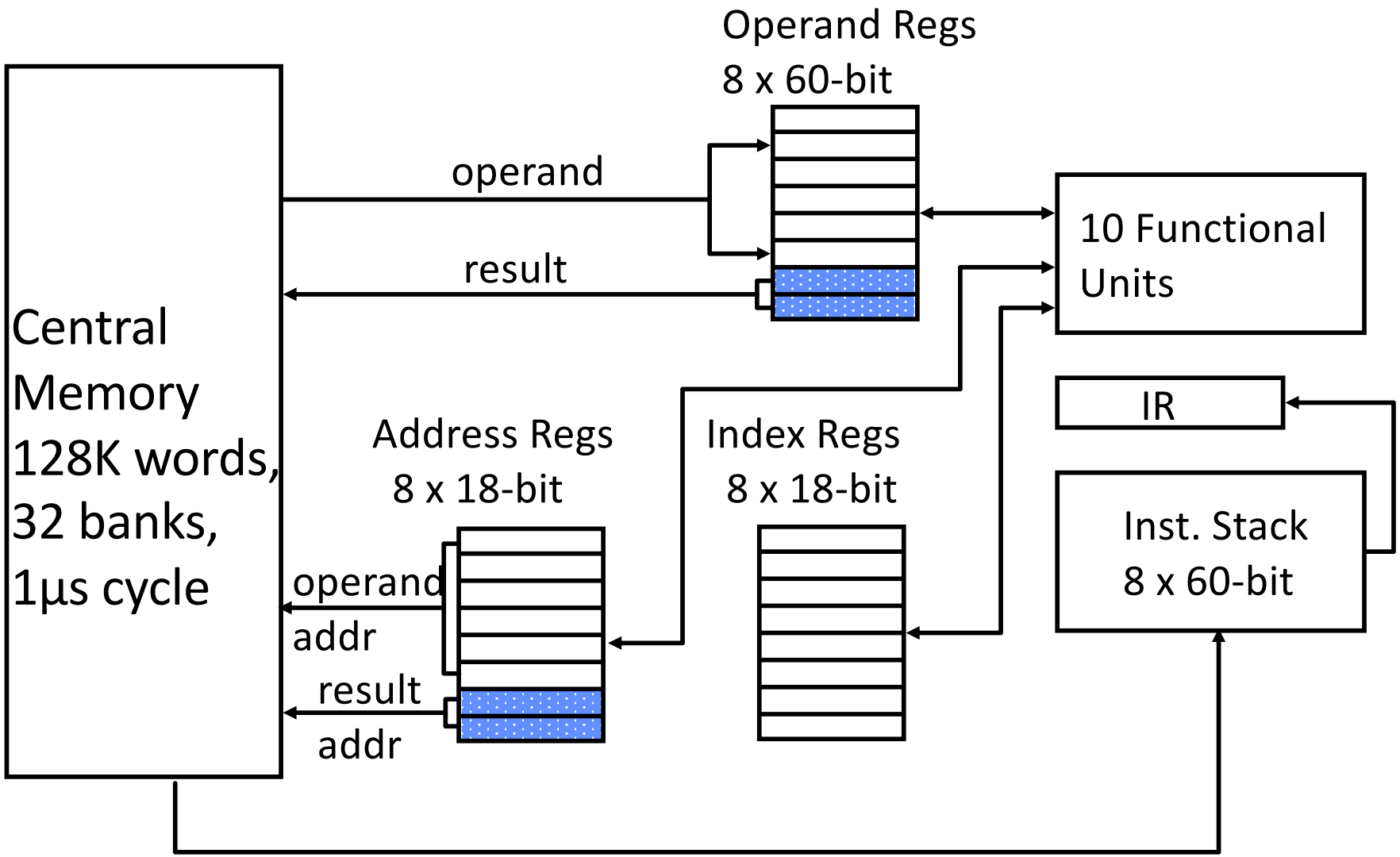
Ri ← M[Rj + disp]

Touching address registers 1 to 5 initiates a load

6 to 7 initiates a store

- very useful for vector operations

CDC 6600: Datapath



CDC6600: 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
```

Ai = address register

Bi = index register

Xi = data register

CDC6600 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 inbetween
- 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 Memo on CDC6600

Thomas Watson Jr., IBM CEO, August 1963:

> *"Last week, Control Data ... 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."*

Computer Architecture Terminology

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

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

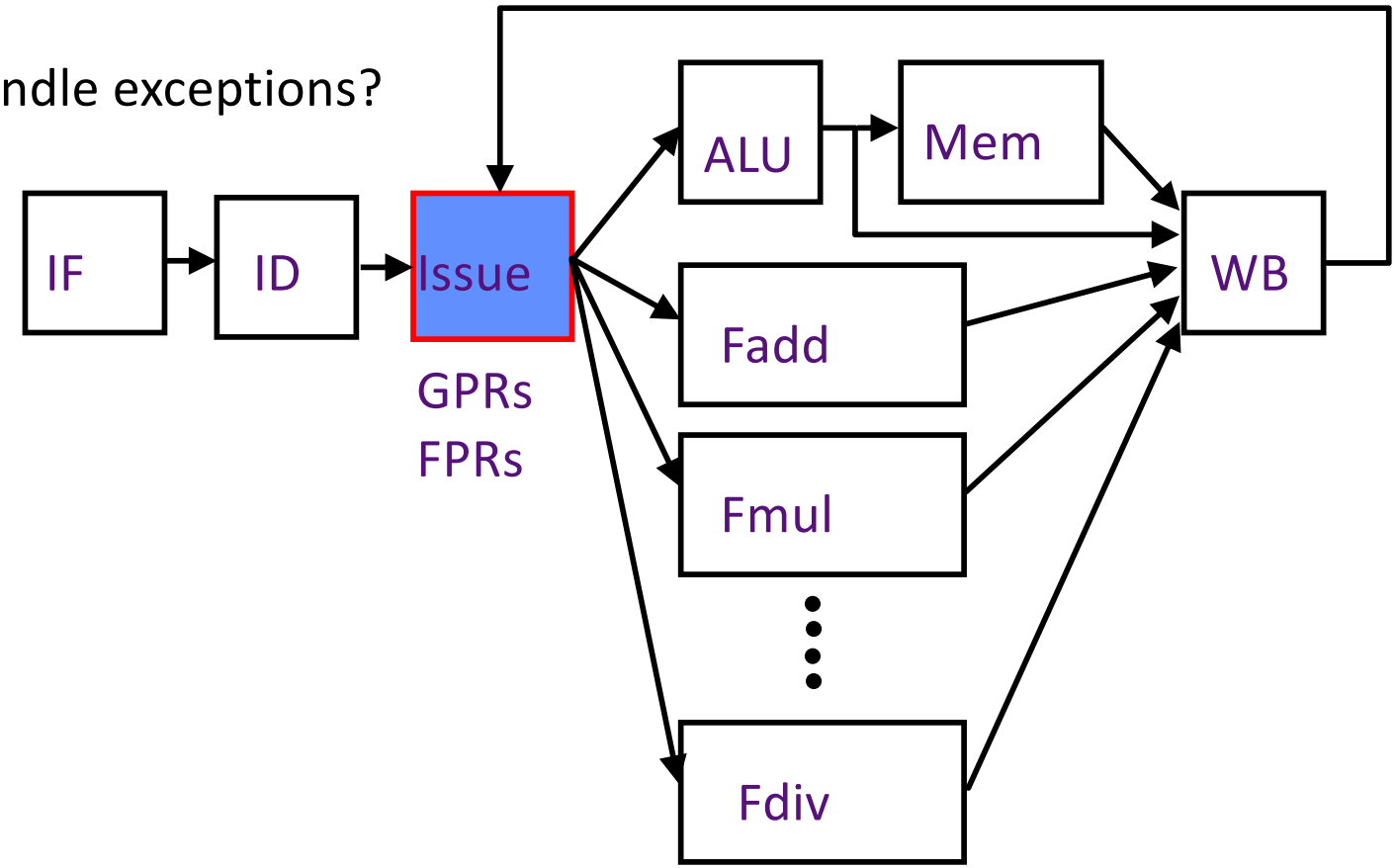
Occupancy (in seconds or cycles): Time during which the unit is blocked on an operation (structural hazard)

Note, for a single functional unit:

- Occupancy can be much less than latency (how?)
- Occupancy can be greater than latency (how?)
- Bandwidth can be greater than 1/latency (how?)
- Bandwidth can be less than 1/latency (how?)

Issues in Complex Pipeline Control

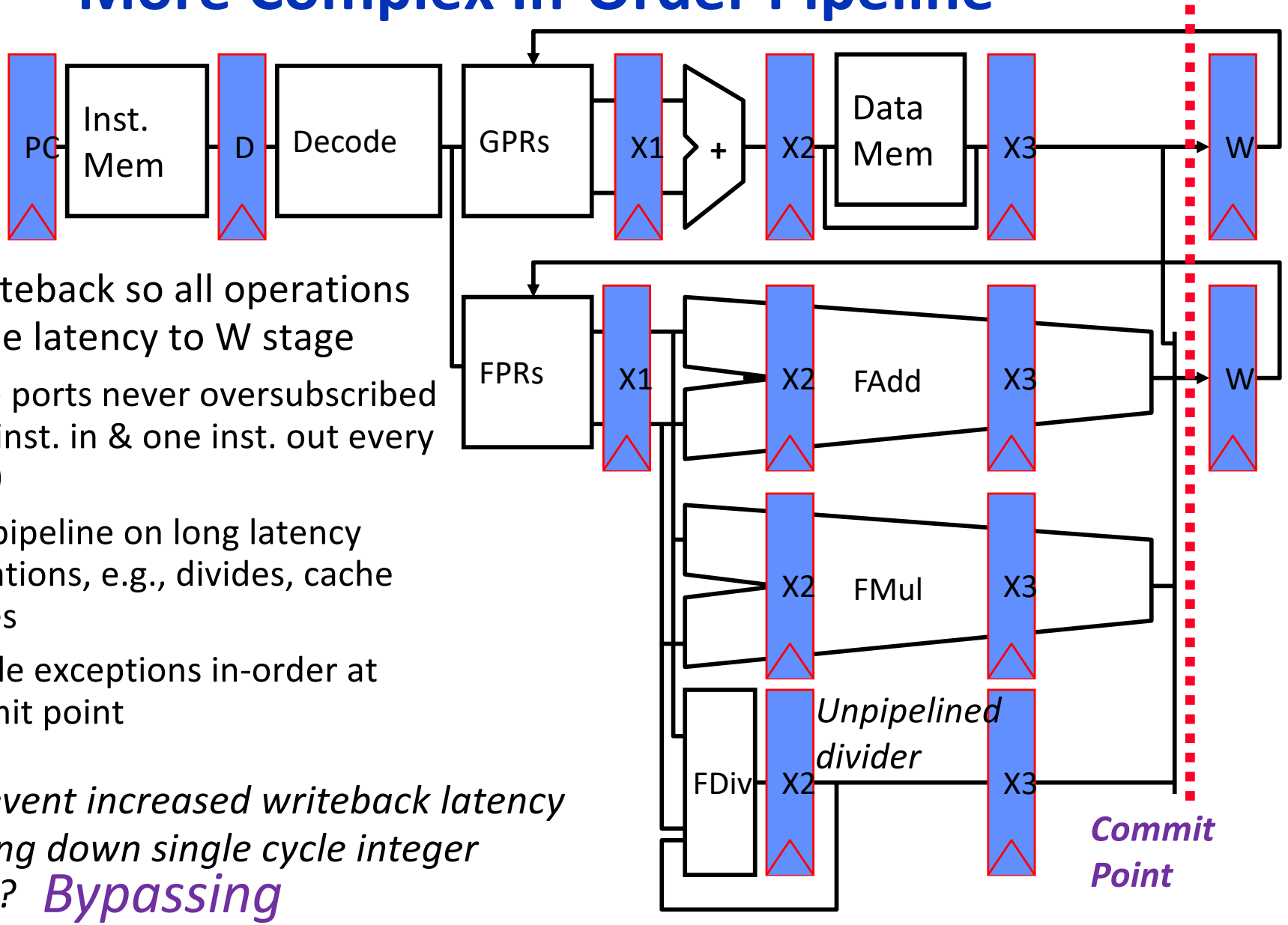
- Structural conflicts at the execution stage if some FPU or memory unit is not pipelined and takes more than one cycle
- Structural conflicts at the write-back stage due to variable latencies of different functional units
- Out-of-order write hazards due to variable latencies of different functional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



CDC6600 Scoreboard

- 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 until register free

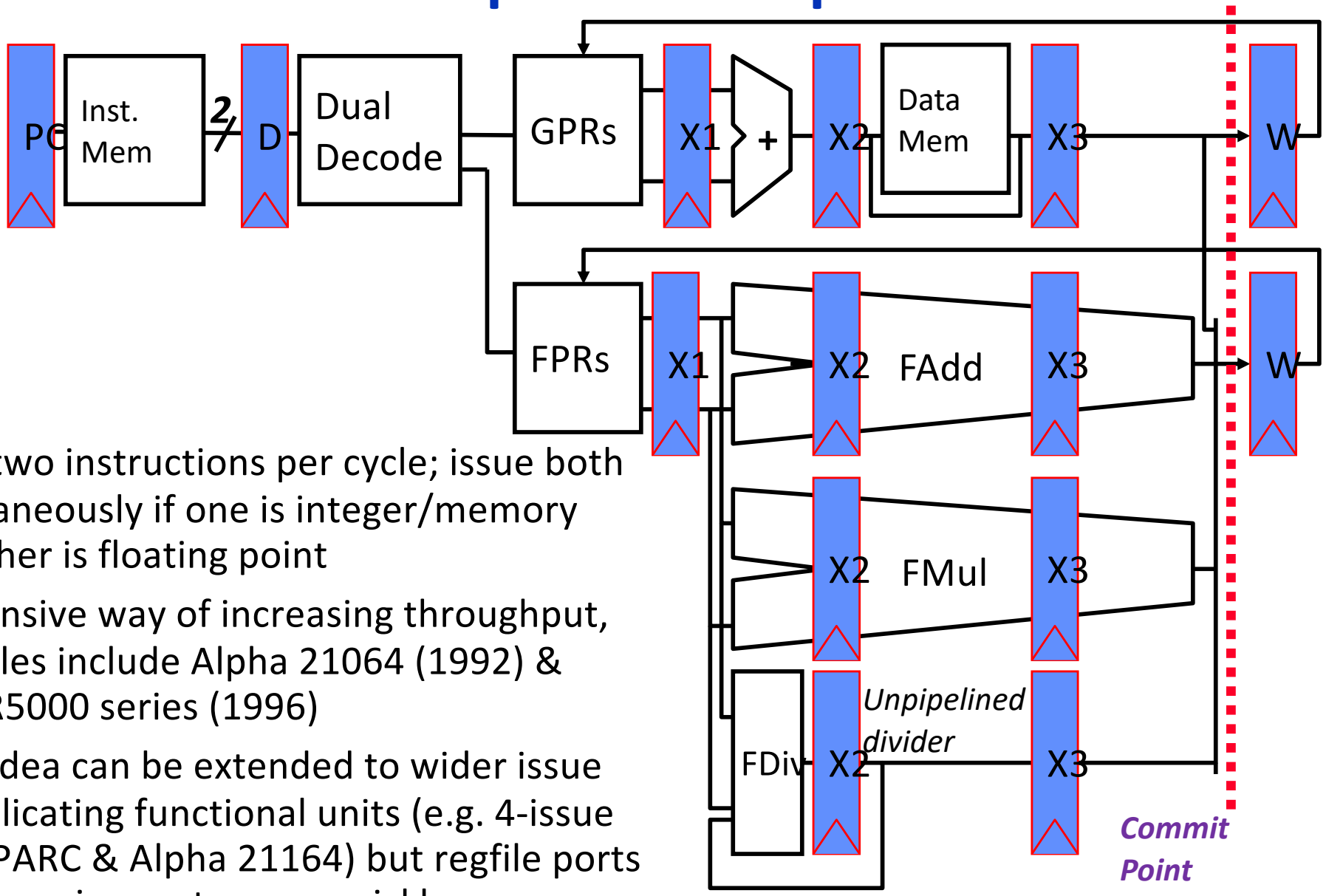
More Complex In-Order Pipeline



- 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? *Bypassing*

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

In-Order Pipeline with two ALU stages

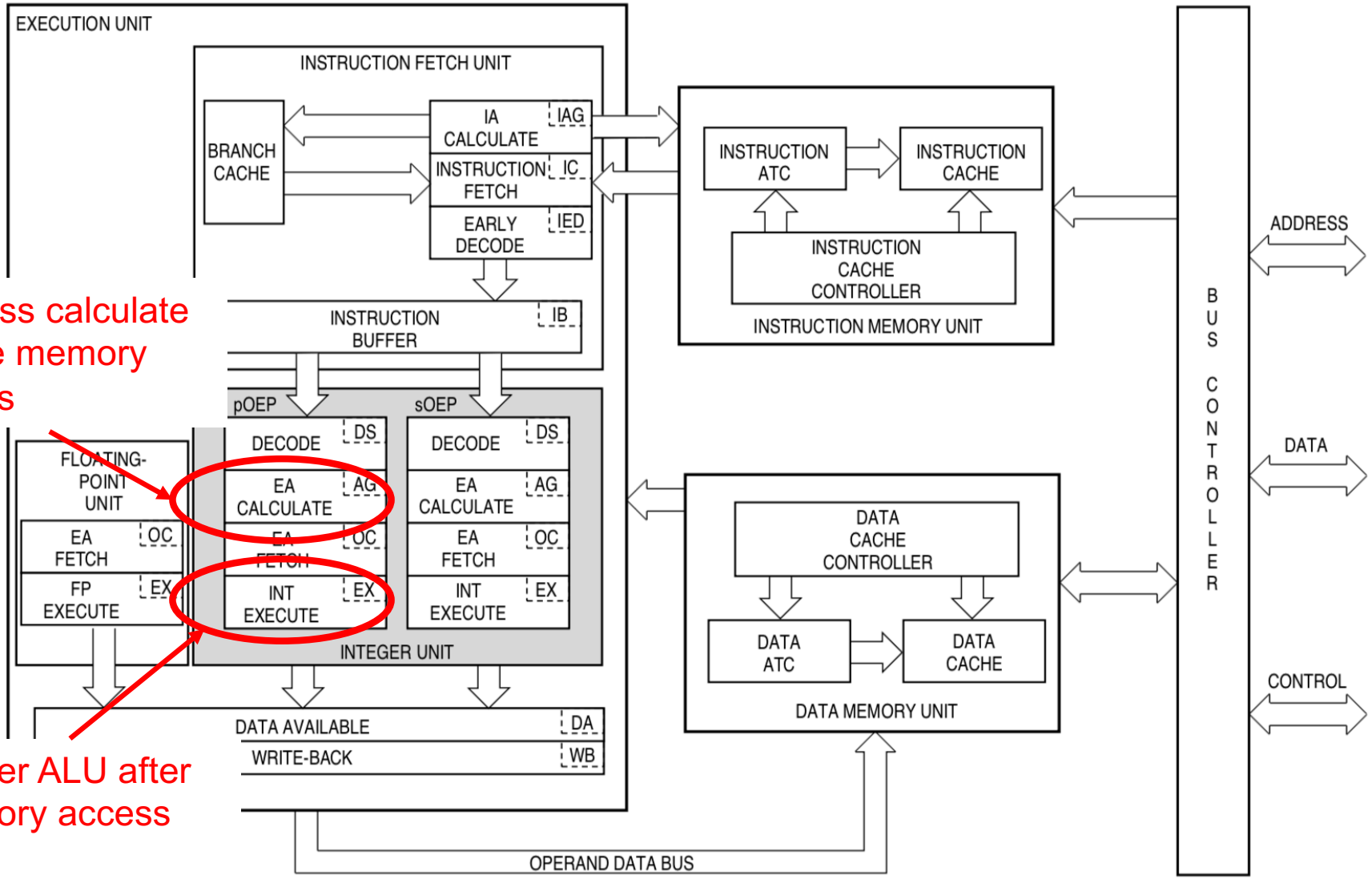
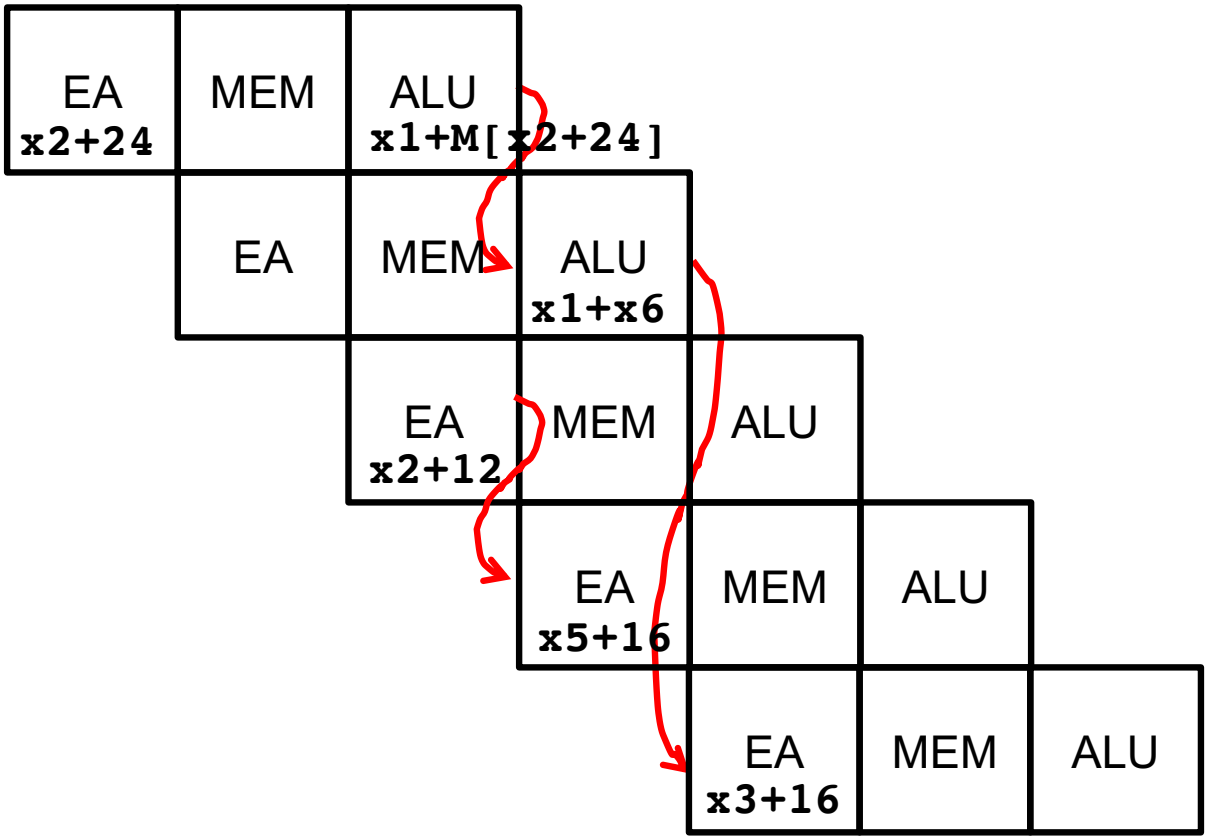


Figure 3-1. MC68060 Integer Unit Pipeline

[© Motorola 1994]

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

Acknowledgements

- This course is partly inspired by previous MIT 6.823 and Berkeley CS252 computer architecture courses created by my collaborators and colleagues:
  - Arvind (MIT)
  - Joel Emer (Intel/MIT)
  - James Hoe (CMU)
  - John Kubiatowicz (UCB)
  - David Patterson (UCB)