Pipelined CPU

Jinyang Li

Slides are based on Patterson and Hennessy

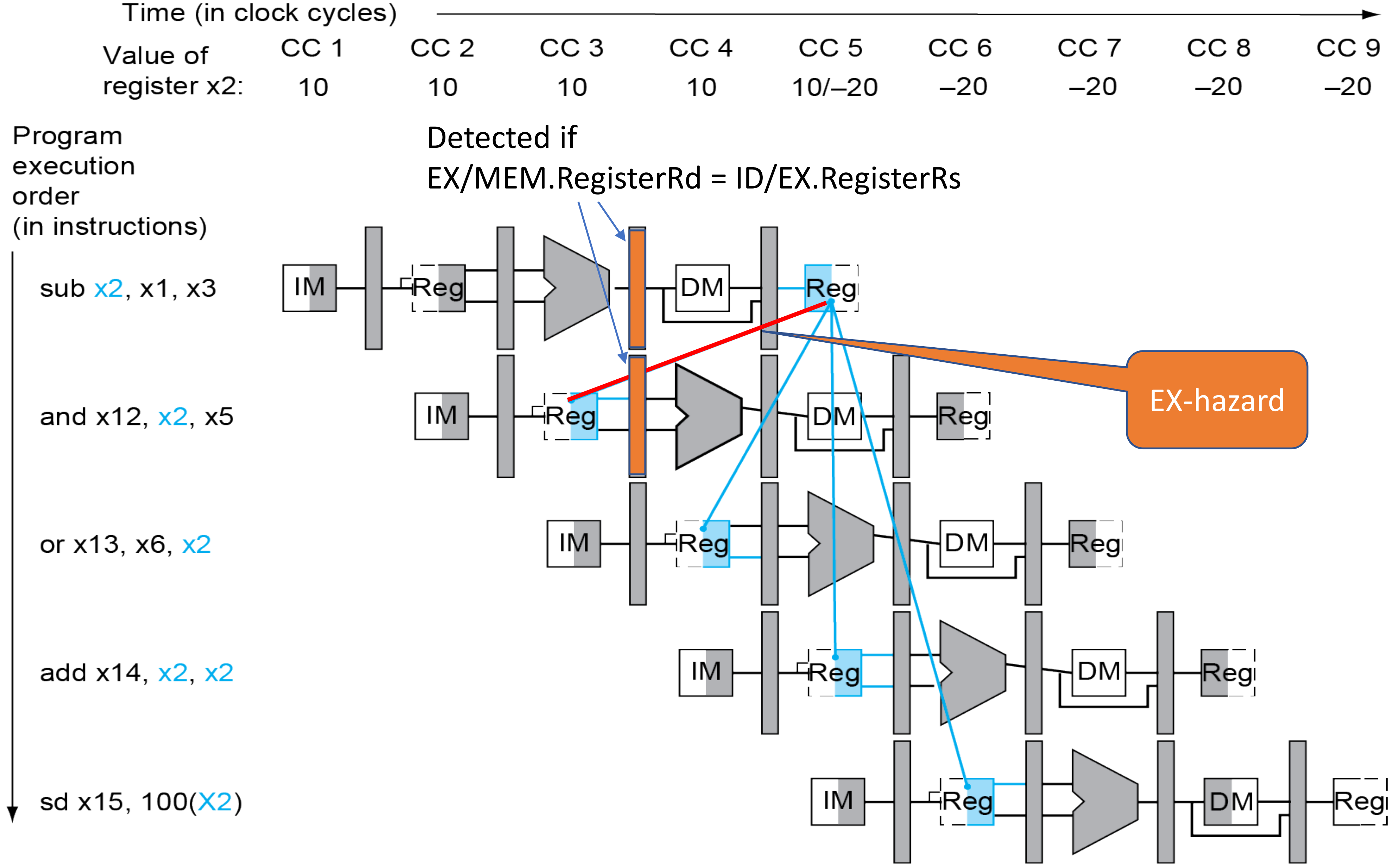
What we’ve learnt so far

- Single cycle RISC-V CPU design
- 5 stage pipelined RISC-V CPU
    - Why pipelining? Faster (ideal throughput speedup: #-of-stages)
- Pipelining challenges: hazards
    - Must stall (bubble) to ensure correctness
- 3 types of hazards:
    - Structure (To mitigate, add resources)
    - Data (To mitigate, ??)
    - Control (To mitigate, ??)

Today’s lesson

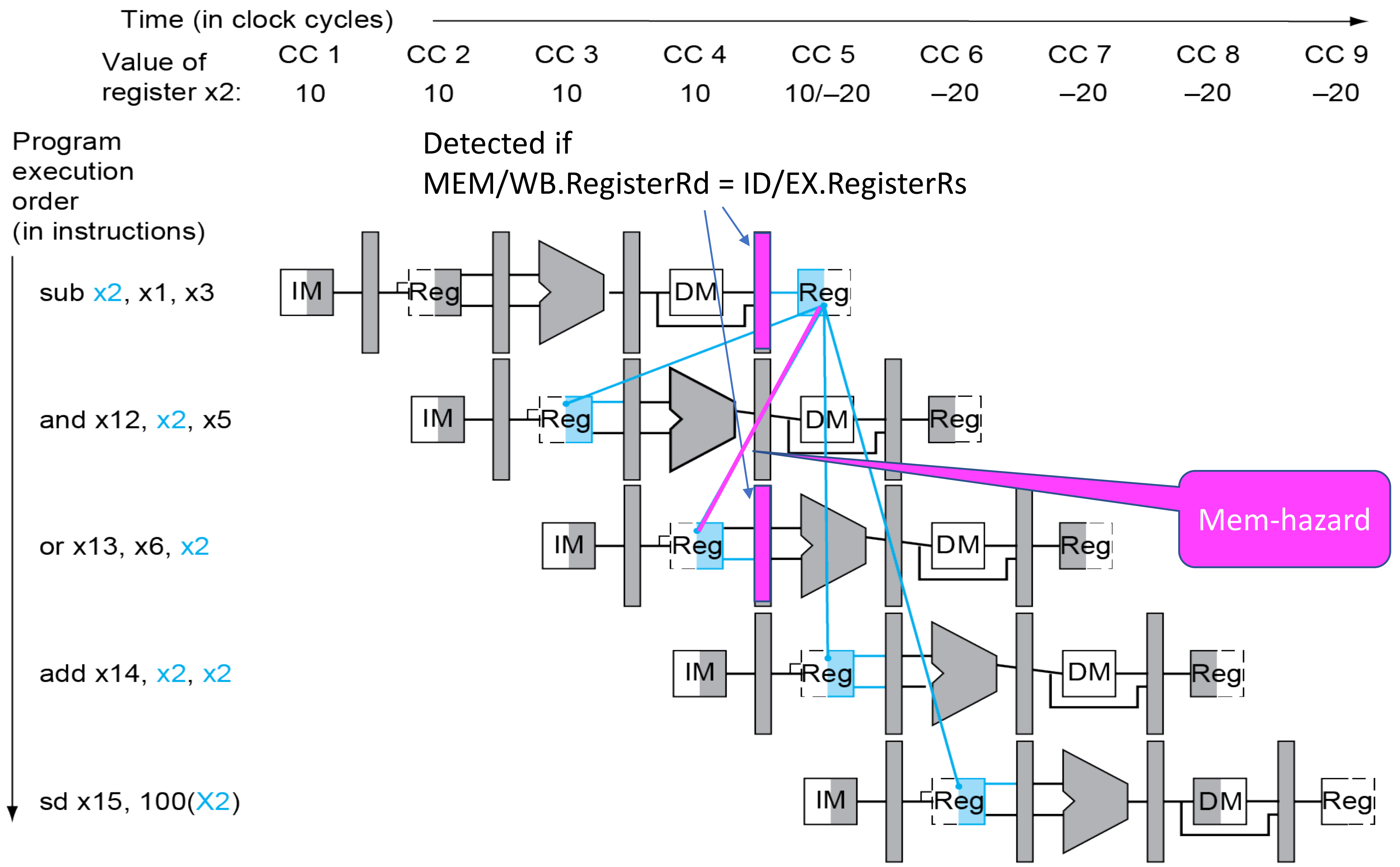
- Handling data hazard
- Handling control hazard
  - Branch hazard
  - exceptions
- Briefly, more advanced topics:
  - Multiple-issue
  - Subword parallelism (SIMD)

Data hazard



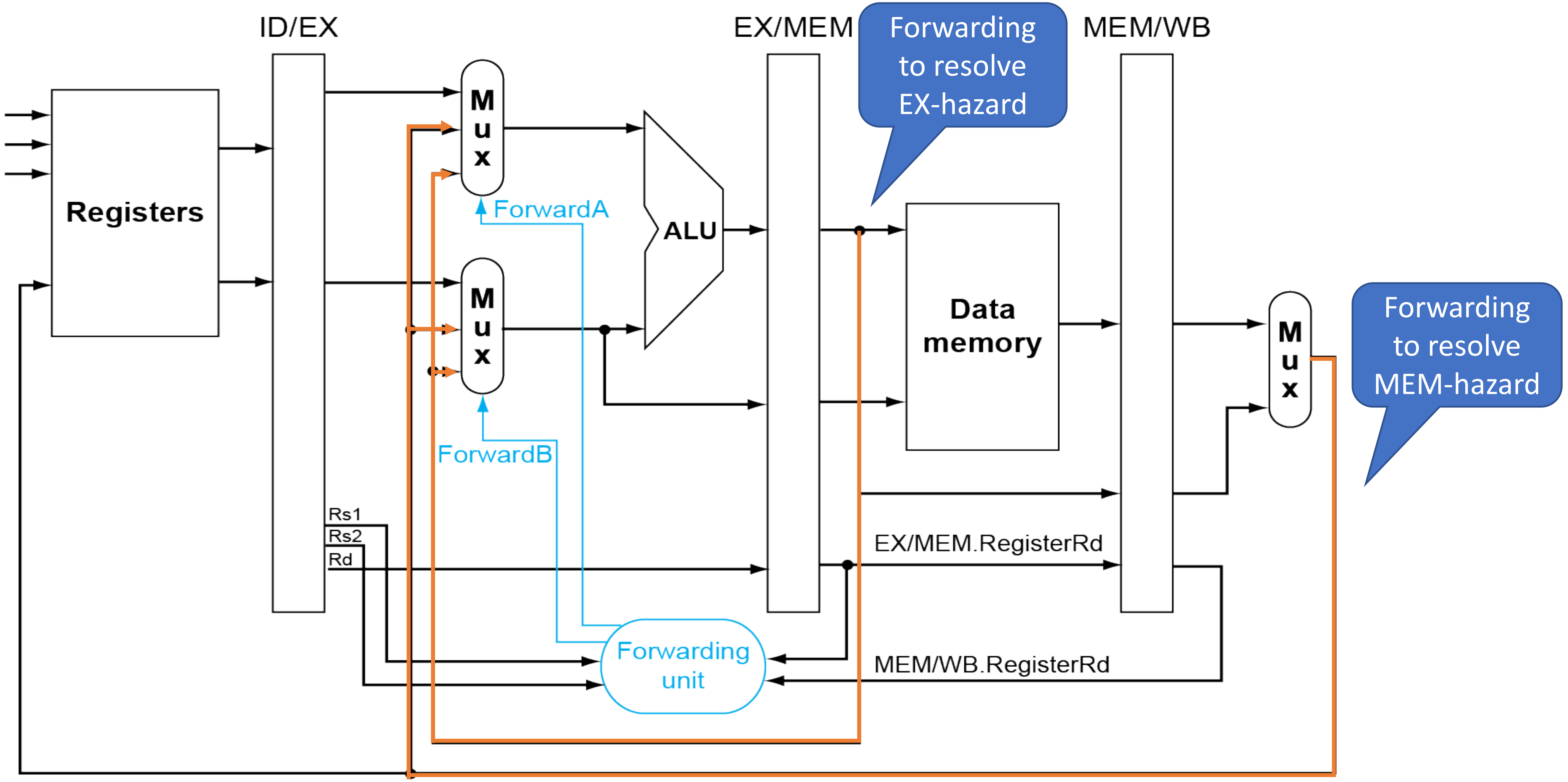
Time (in clock cycles)
Value of register x2:
CC 1 10
CC 2 10
CC 3 10
CC 4 10
CC 5 10/–20
CC 6 –20
CC 7 –20
CC 8 –20
CC 9 –20
Program execution order (in instructions)
sub x2, x1, x3
and x12, x2, x5
or x13, x6, x2
add x14, x2, x2
sd x15, 100(X2)
IM
Reg
DM
Detected if
EX/MEM.RegisterRd = ID/EX.RegisterRs
EX-hazard

Data hazard



Time (in clock cycles)
Value of register x2:
CC 1 10
CC 2 10
CC 3 10
CC 4 10
CC 5 10/–20
CC 6 –20
CC 7 –20
CC 8 –20
CC 9 –20
Program execution order (in instructions)
Detected if
MEM/WB.RegisterRd = ID/EX.RegisterRs
sub x2, x1, x3
and x12, x2, x5
or x13, x6, x2
add x14, x2, x2
sd x15, 100(X2)
IM
Reg
DM
Mem-hazard

Forwarding Paths

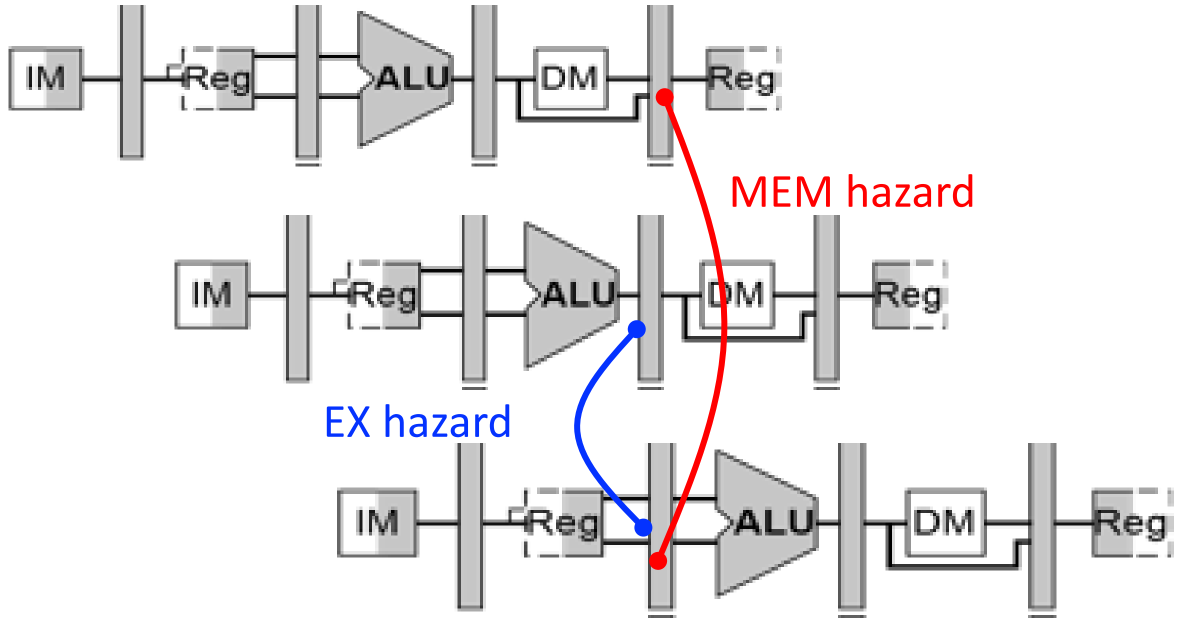


Double Data Hazard

- Consider the sequence:

```
add x1,x1,x2
add x1,x1,x3
add x1,x1,x4
```

- Both hazards occur
  - Want to use the most recent
- Revise MEM hazard condition
  - Only fwd if EX hazard condition isn't true

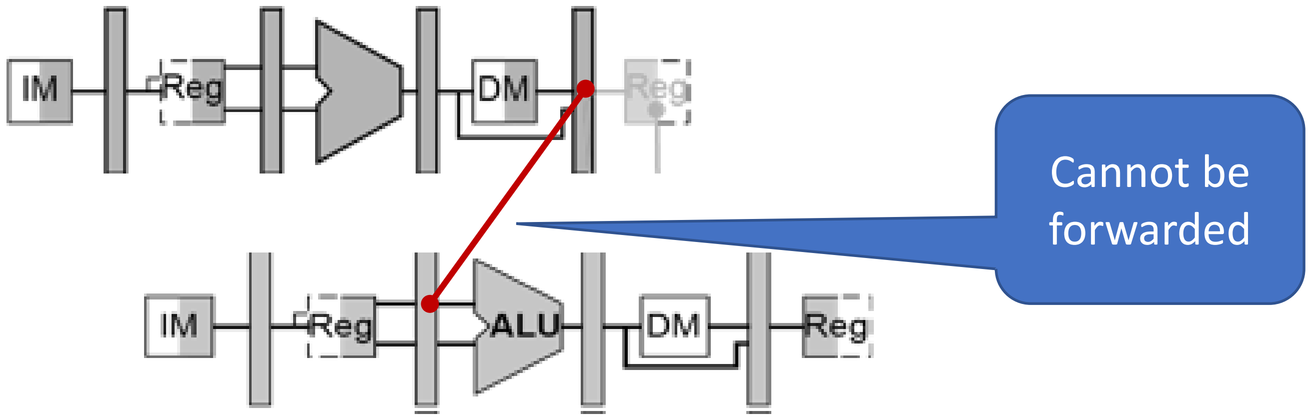


Load-Use Hazard cannot be forwarded away

- Load-use hazard cannot be resolved using forwarding alone

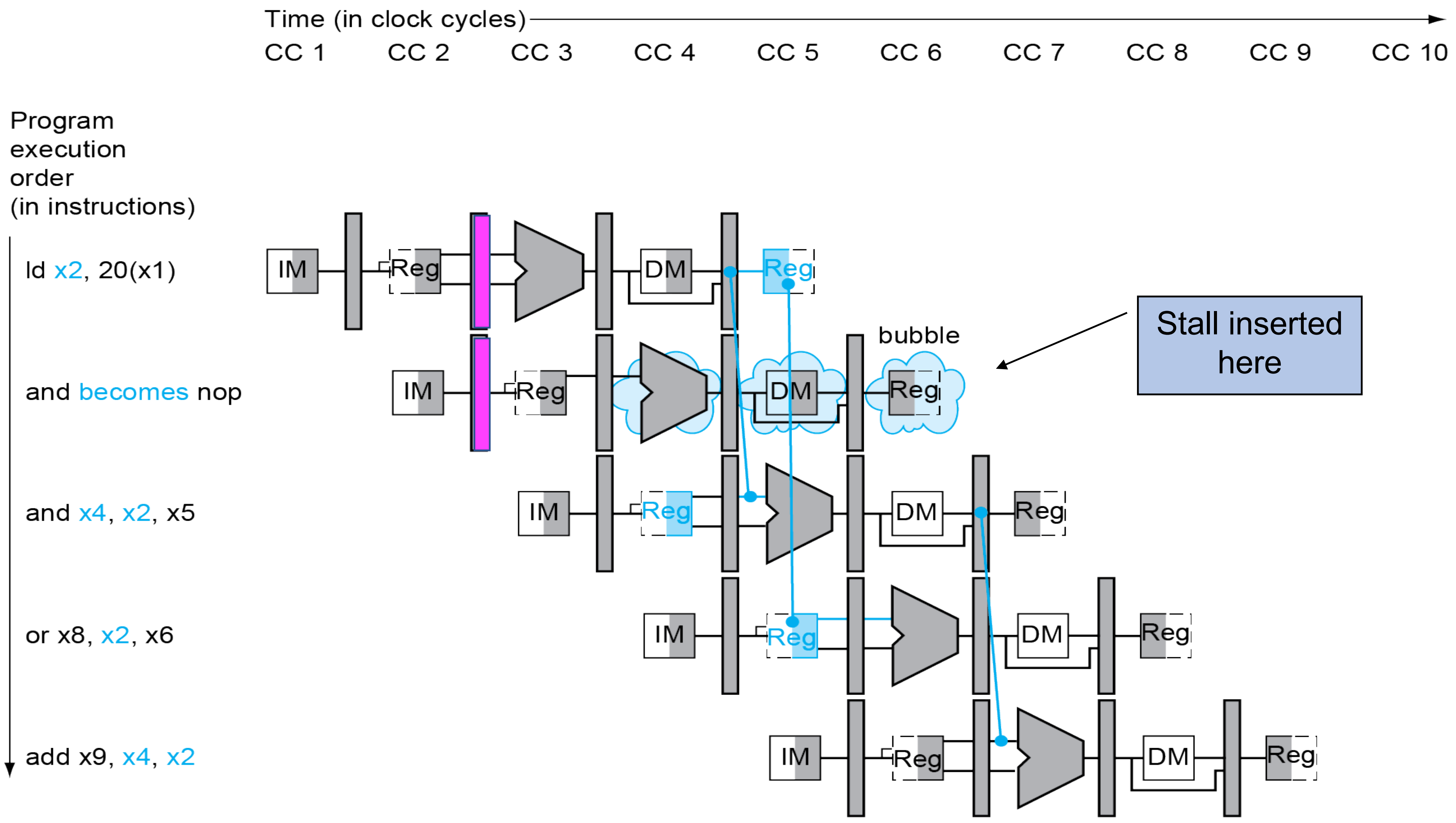
ld x2, 20(x1)

and x4, x2, x5



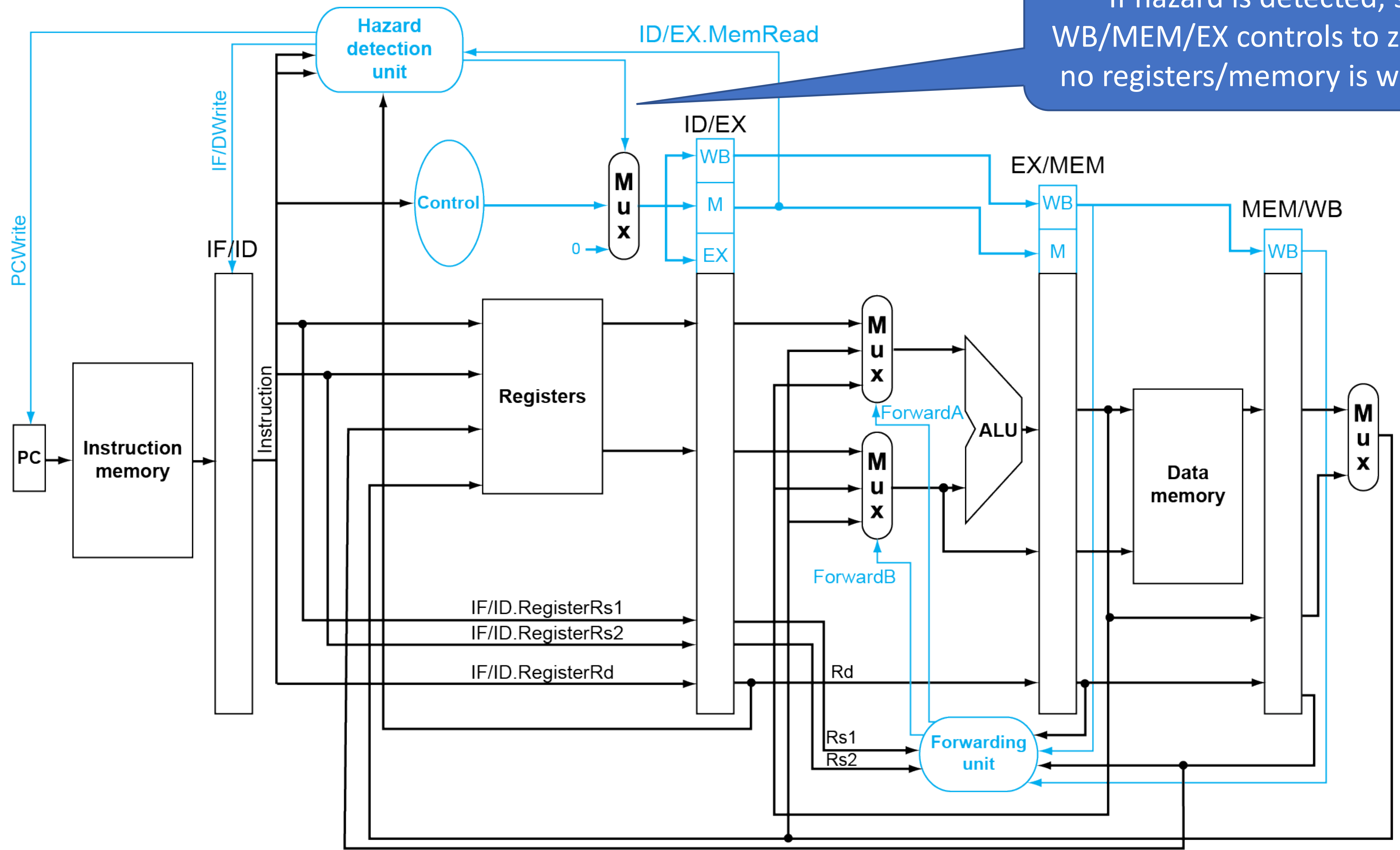
- Check load-use hazard using condition
  - ID/EX.MemRead and ((ID/EX.RegisterRd = IF/ID.RegisterRs1) or (ID/EX.RegisterRd = IF/ID.RegisterRs2))
- If detected, stall and insert bubble

Stalling upon detection of load-use data hazard

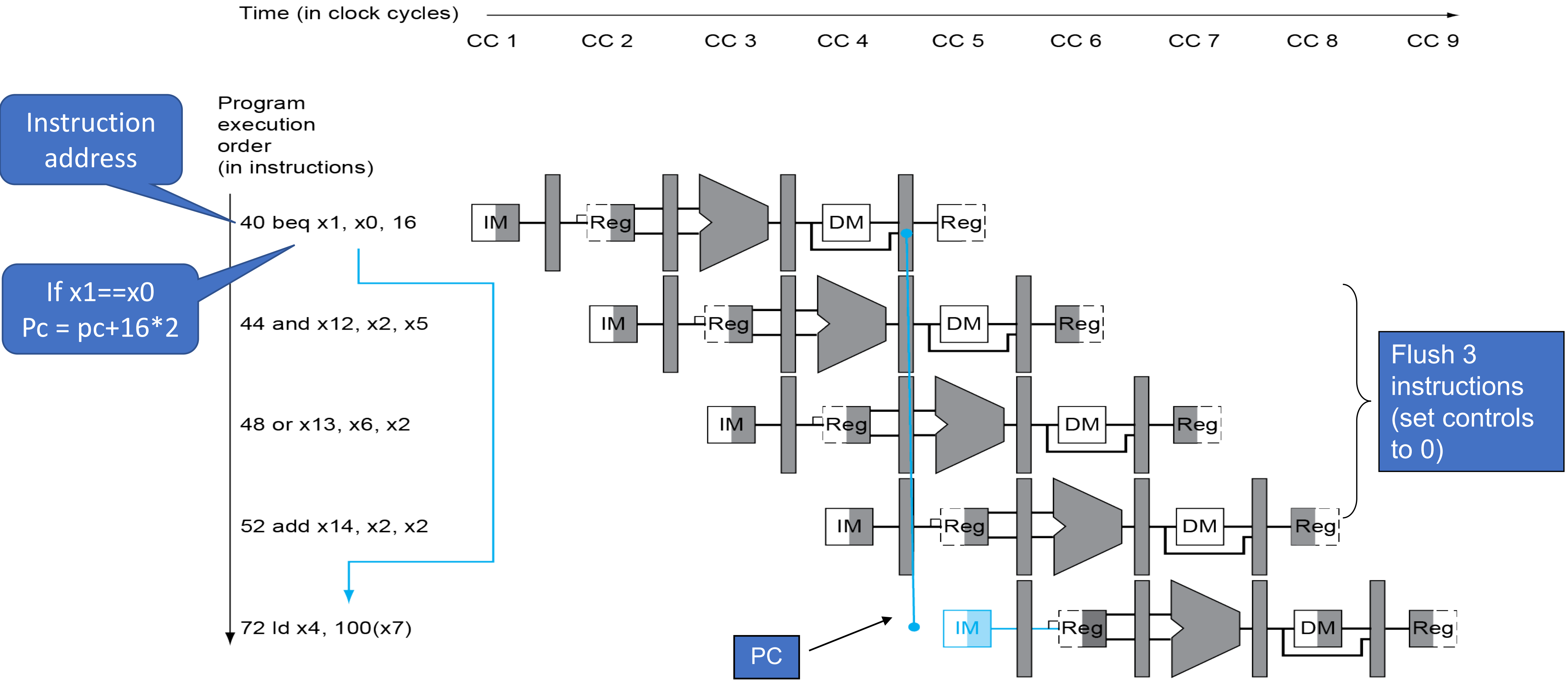


Datapath with Hazard Detection

If hazard is detected, set WB/MEM/EX controls to zero (so no registers/memory is written)



Branch Hazard: if outcome is determined in MEM…

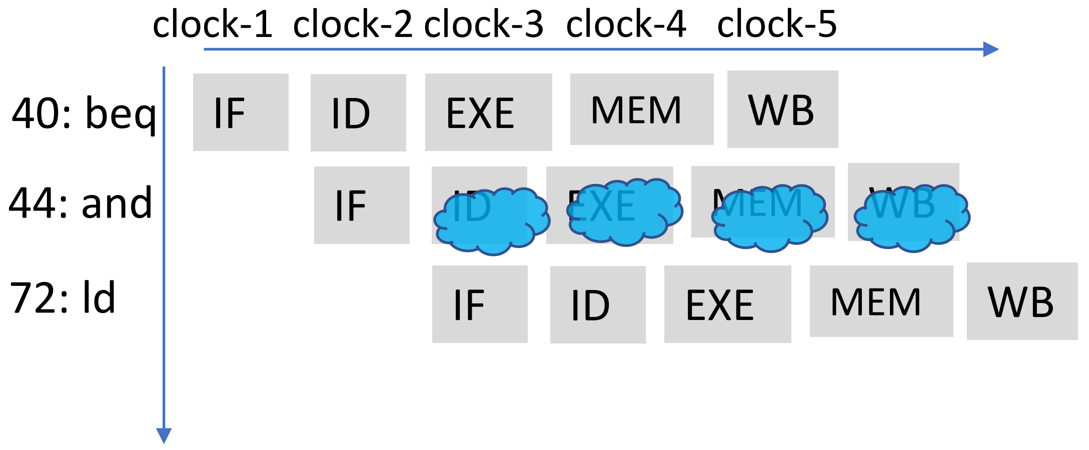


Reduce Branch Delay

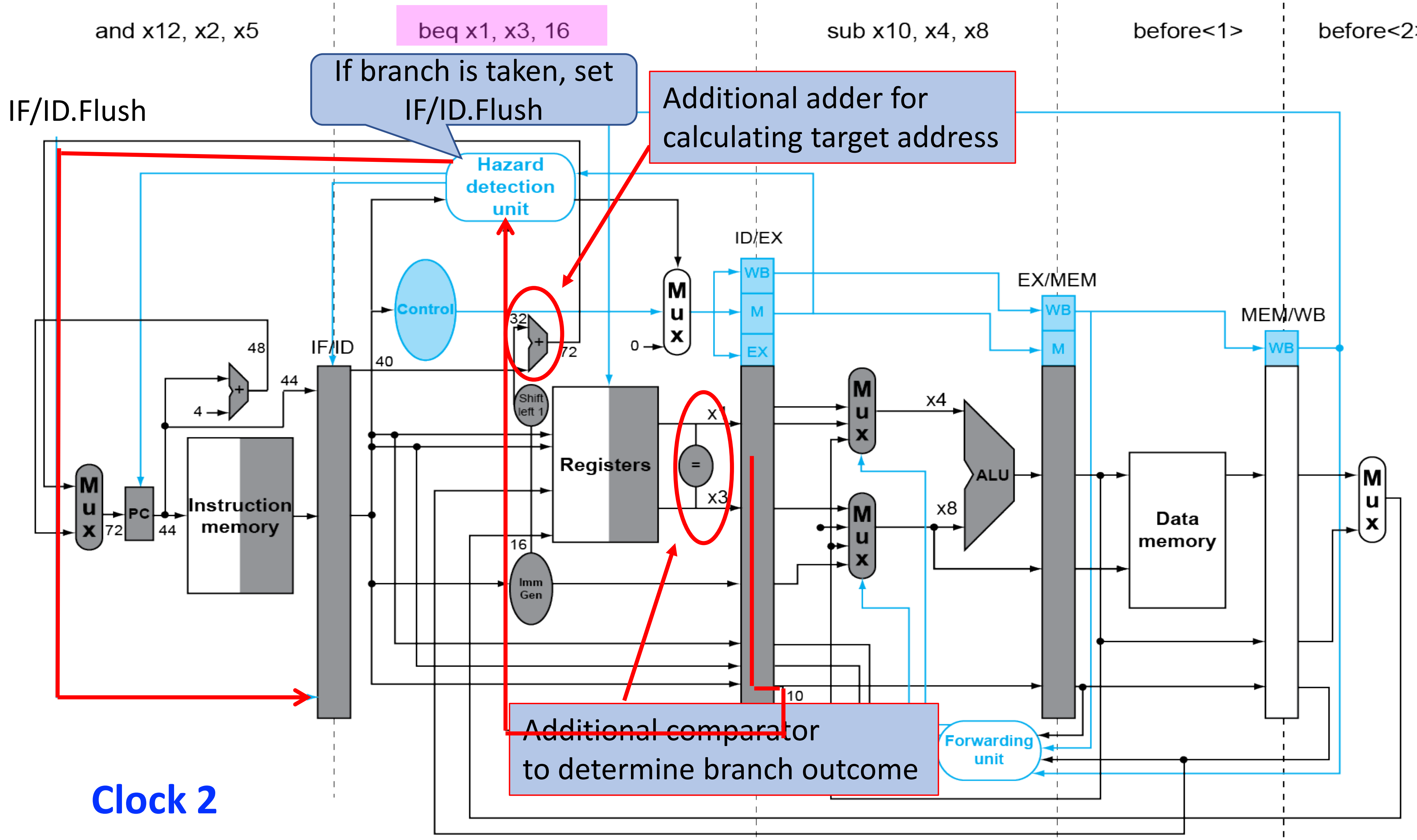
- Add hardware to determine branch outcome earlier (e.g. in ID instead of MEM) → fewer instructions to flush

```
36:   sub  x10, x4, x8
40:   beq  x1,  x3, 16  // PC-relative branch
                        // to 40+16*2=72
44:   and  x12, x2, x5
48:   orr  x13, x2, x6
52:   add  x14, x4, x2
56:   sub  x15, x6, x7
      ...
72:   ld   x4, 50(x7)
```

How many instructions to flush if branch outcome is known in ID?

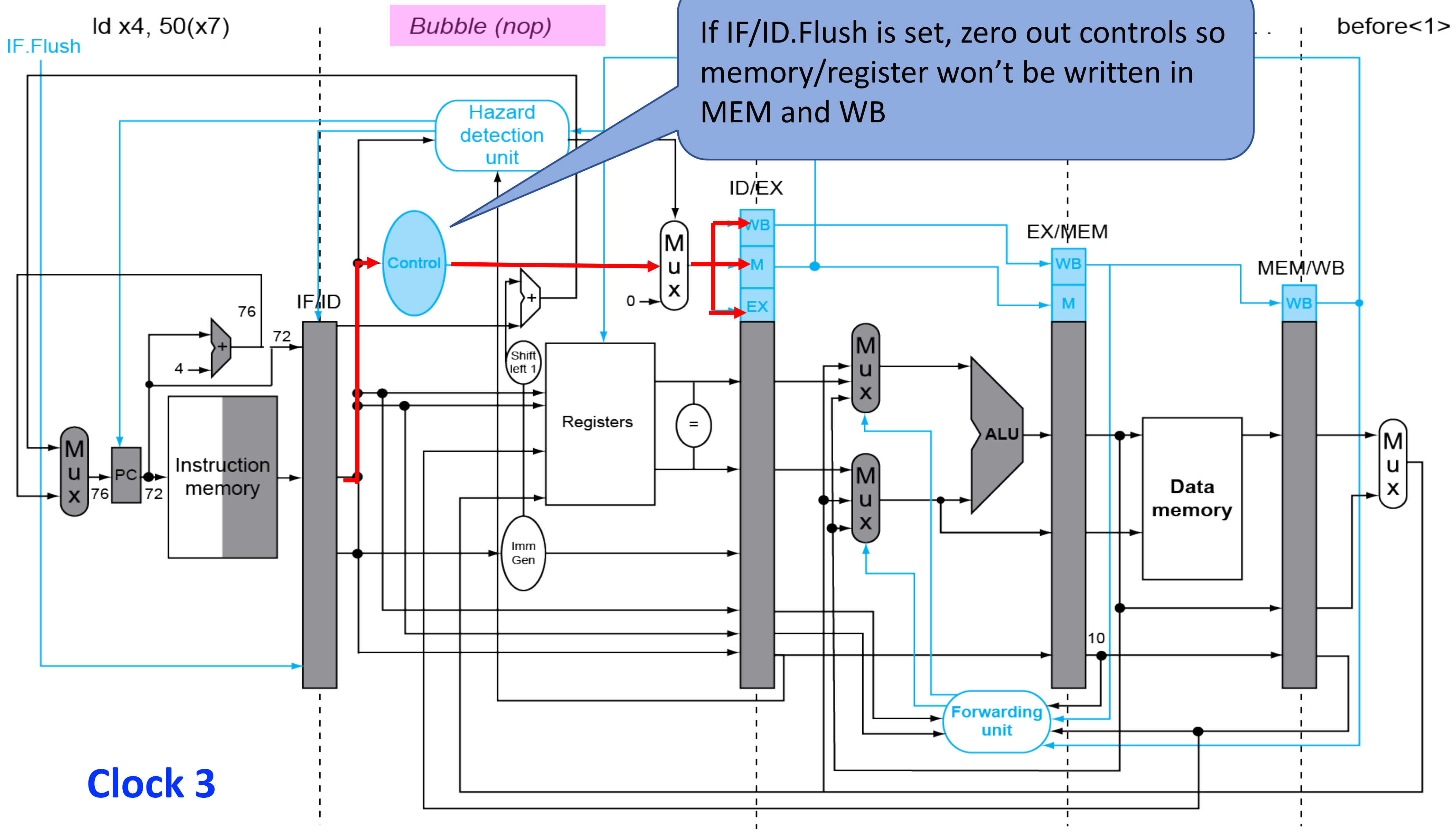


Branch determined in ID and is taken



Clock 2

Branch determined in ID and is taken



If IF/ID.Flush is set, zero out controls so memory/register won't be written in MEM and WB

Clock 3

Dynamic Branch Prediction

- Our simple 5-stage pipeline’s branch penalty is 1 bubble, but
  - In deeper pipelines, branch penalty is more significant
- Solution: dynamic prediction
  - Branch prediction buffer (aka branch history table)
    - Indexed by recent branch instruction addresses
    - Stores outcome (taken/not taken)
  - To execute a branch
    - Check table, expect the same outcome
    - Start fetching from fall-through or target
    - If wrong, flush pipeline and flip prediction

1-Bit Predictor: Shortcoming

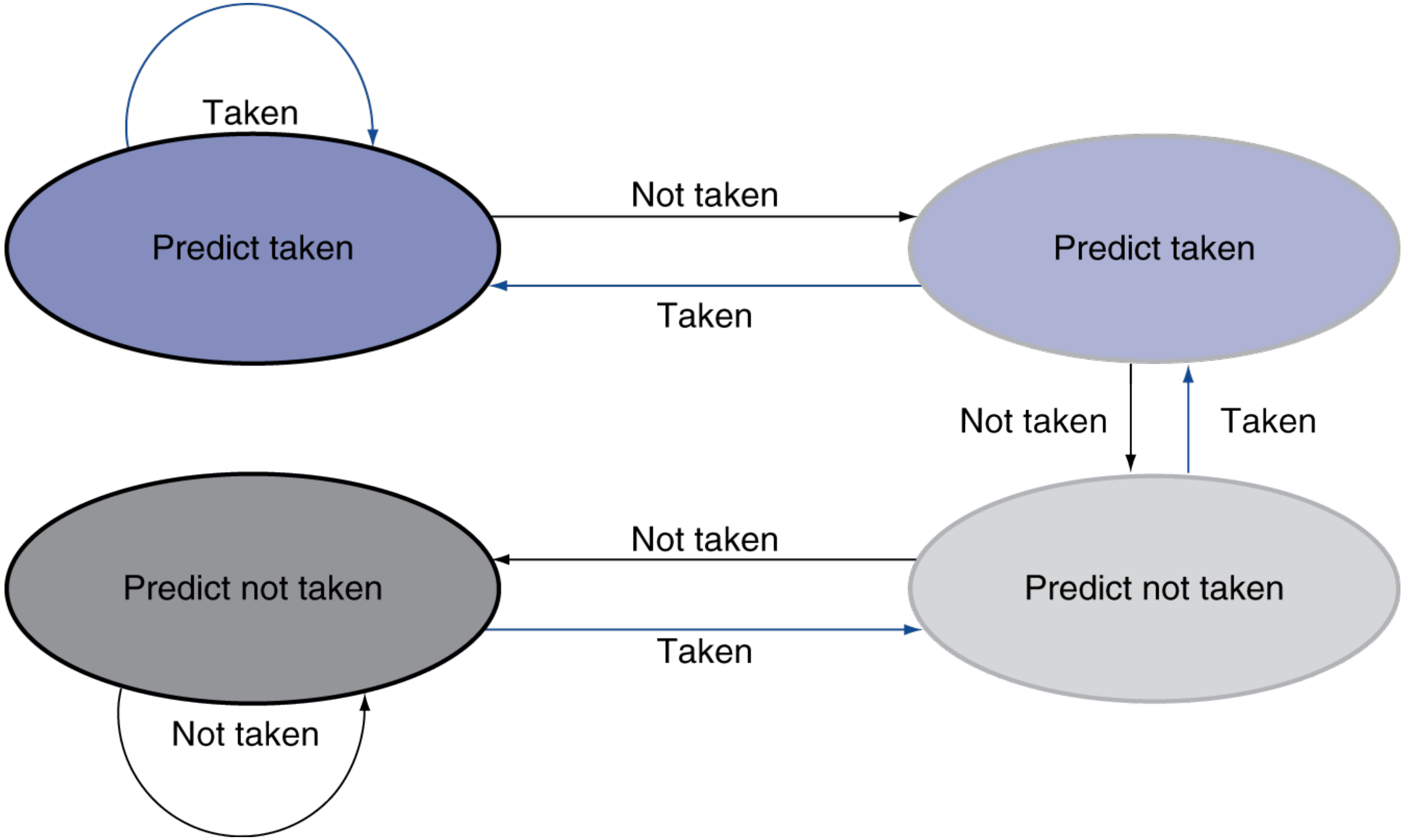
- Inner loop branches mispredicted twice!

```
outer: …
       …
inner: …
       …
       beq …, …, inner
       …
       beq …, …, outer
```

- Mispredict as taken on last iteration of inner loop
- Then mispredict as not taken on first iteration of inner loop next time around

2-Bit Predictor

- Only change prediction on two successive mispredictions



Calculating Branch Target (needed if branch is predicted taken)

- Even with predictor, still need to calculate the target address
  - 1-cycle penalty for a taken branch
- Branch target buffer
  - Cache of target addresses
  - Indexed by PC when instruction fetched
    - If hit and instruction is branch predicted taken, can fetch target immediately

Exceptions and Interrupts

- "Unexpected" events requiring change in flow of control
- Exception
  - Arises within the CPU
    - e.g., undefined opcode, syscall, memory permission error, divide by zero, …
- Interrupt
  - From an external source like I/O controllers
    - E.g. keyboard stroke, packet arrival
- Dealing with them without sacrificing performance is hard

Handling Exceptions

- Save PC of offending (or interrupted) instruction
  - In RISC-V: Supervisor Exception Program Counter (SEPC)
- Save indication of the problem
  - In RISC-V: Supervisor Exception Cause Register (SCAUSE)
- Jump to handler (aka code in OS kernel)
  - Assume at $0000\ 0000\ 1C09\ 0000_{hex}$
- OS's handler does either of the following:
  - Take corrective action; restart offending instruction
  - Terminate program; report cause

An Alternate Exception Handling Mechanism

- Vectored Exception/Interrupts (x86)
- Vector index (interrupt descriptor) specifies cause of exception
  - E.g. in x86, 0 (divide by zero), 6 (invalid opcode), …
- OS sets up the Interrupt Descriptor Table
  - IDT[i] contains handler address, where i is vector index.

Exceptions in a Pipeline

- Another form of control hazard
- Consider malfunction on add in EX stage
  `add x1, x2, x1`
- What must be done?
  - Prevent x1 from being clobbered
  - Complete previous instructions
  - Flush `add` and subsequent instructions
  - Save PC (of offending instruction) in SEPC and set SCAUSE with cause
  - Transfer control to handler
- Similar to mispredicted branch
  - Use much of the same hardware

Exception Example

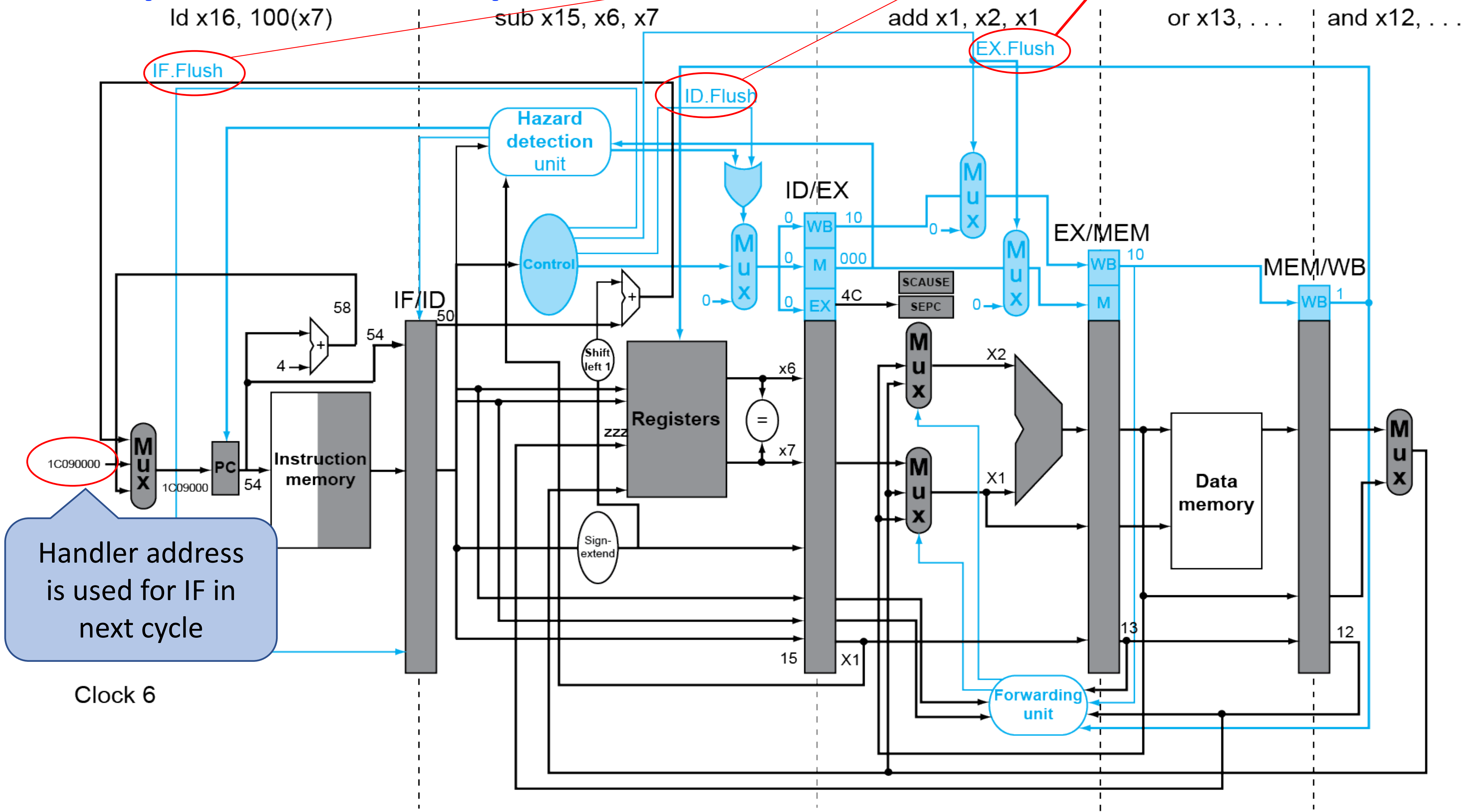
- Exception on add in

```
40     sub  x11, x2, x4
44     and  x12, x2, x5
48     orr  x13, x2, x6
4c     add  x1,  x2, x1
50     sub  x15, x6, x7
54     ld   x16, 100(x7)
…
```

- Handler

```
1C090000    sd  x26, 1000(x10)
1c090004    sd  x27, 1008(x10)
…
```

Exception Example



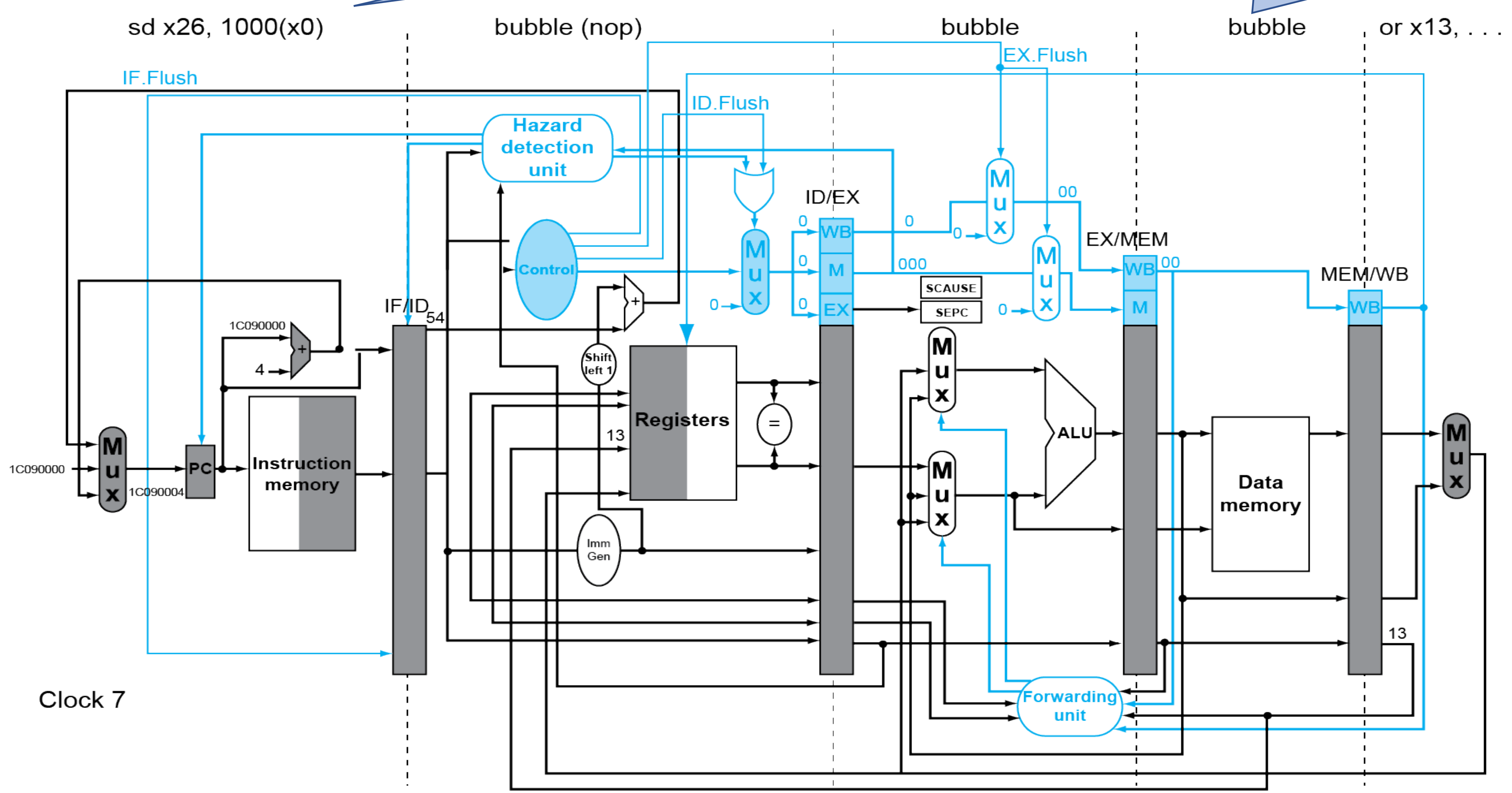
These signals are set if add malfunctioned

Handler address is used for IF in next cycle

Exception Example

Instructions in exception handler

Offending instruction (add x1, x2, x1) becomes a bubble (so it won't clobber registers)



Today's lesson

- Handling data hazard
- Handling control hazard
  - Branch hazard
  - exceptions
- Briefly, more advanced topics:
  - Multiple-issue
  - Subword parallelism (SIMD)

Instruction-Level Parallelism (ILP)

- Pipelining: execute multiple instructions in parallel
- To increase ILP
  - Deeper pipeline
    - Less work per stage $\Rightarrow$ shorter clock cycle
  - Multiple issue
    - Replicate pipeline stages $\Rightarrow$ multiple pipelines
    - Start multiple instructions per clock cycle
      - Finish multiple Instructions Per Cycle (IPC>1)
    - E.g., 4GHz 4-way multiple-issue
      - 16 billion instructions/sec, peak IPC = 4 (CPI = 1/IPC = 0.25)
    - Challenges: dependencies among multi-issued instructions
      - reduce peak IPC

Multiple Issue

- Static multiple issue (Very Long Instruction Word)
    - Compiler groups instructions to be issued together
    - Compiler detects and avoids hazards
- Dynamic multiple issue (superscalar)
    - CPU examines instruction stream and chooses instructions to issue each cycle
    - Compiler can help by reordering instructions
    - CPU resolves hazards using advanced techniques at runtime

Dynamic Pipeline Scheduling

- Allow the CPU to execute instructions out of order to avoid stalls
  - But commit result to registers in order
- Example

```
ld   x31,20(x21)
add  x1,x31,x2
sub  x23,x23,x3
andi x5,x23,20
```

  - Can start `sub` while `add` is waiting for ld

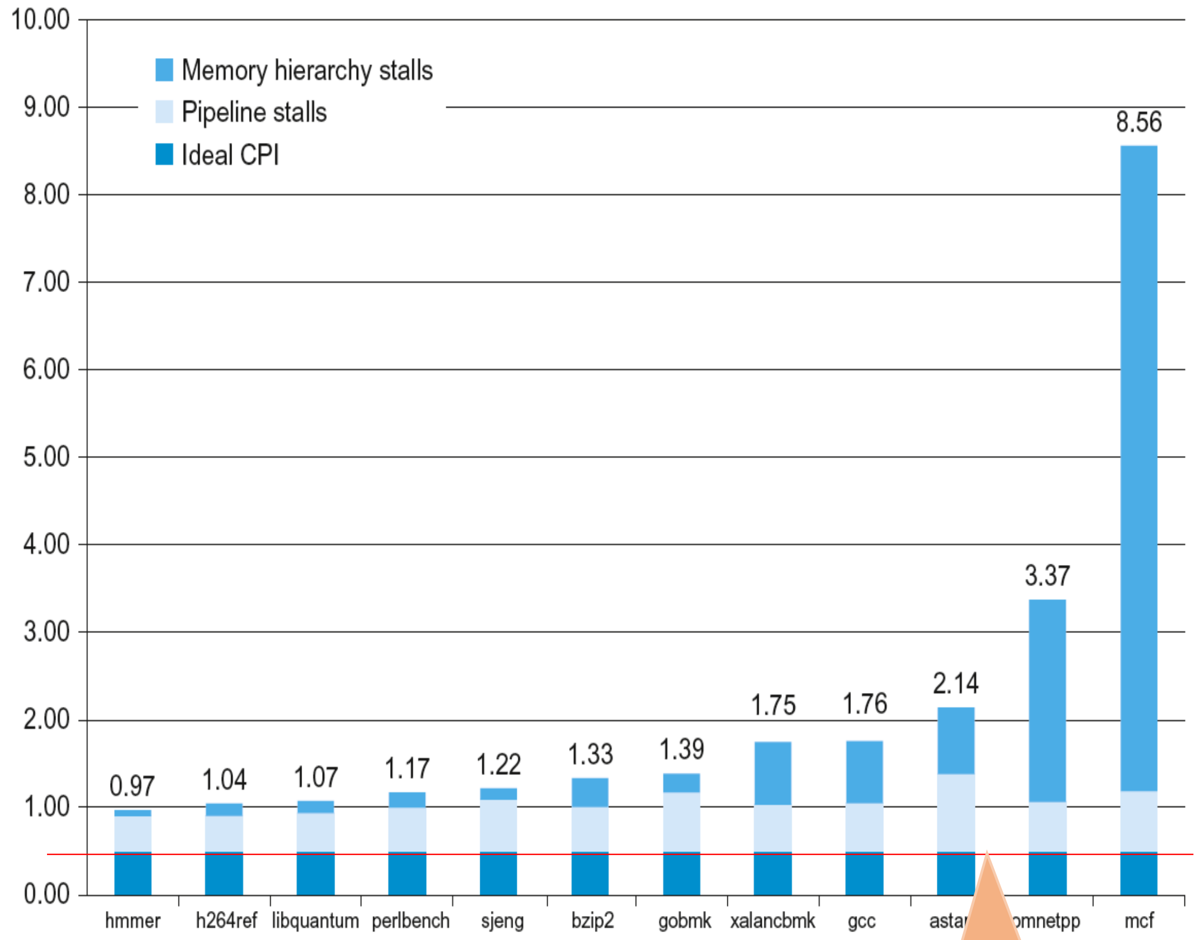
Why Do Dynamic Scheduling?

- Why not just let the compiler schedule code?
- Not all stalls are predicable
  - e.g., cache misses
- Can’t always schedule around branches
  - Branch outcome is dynamically determined
- Different implementations of an ISA have different latencies and hazards

Complexities require power: ARM Cortex A53 and Intel i7

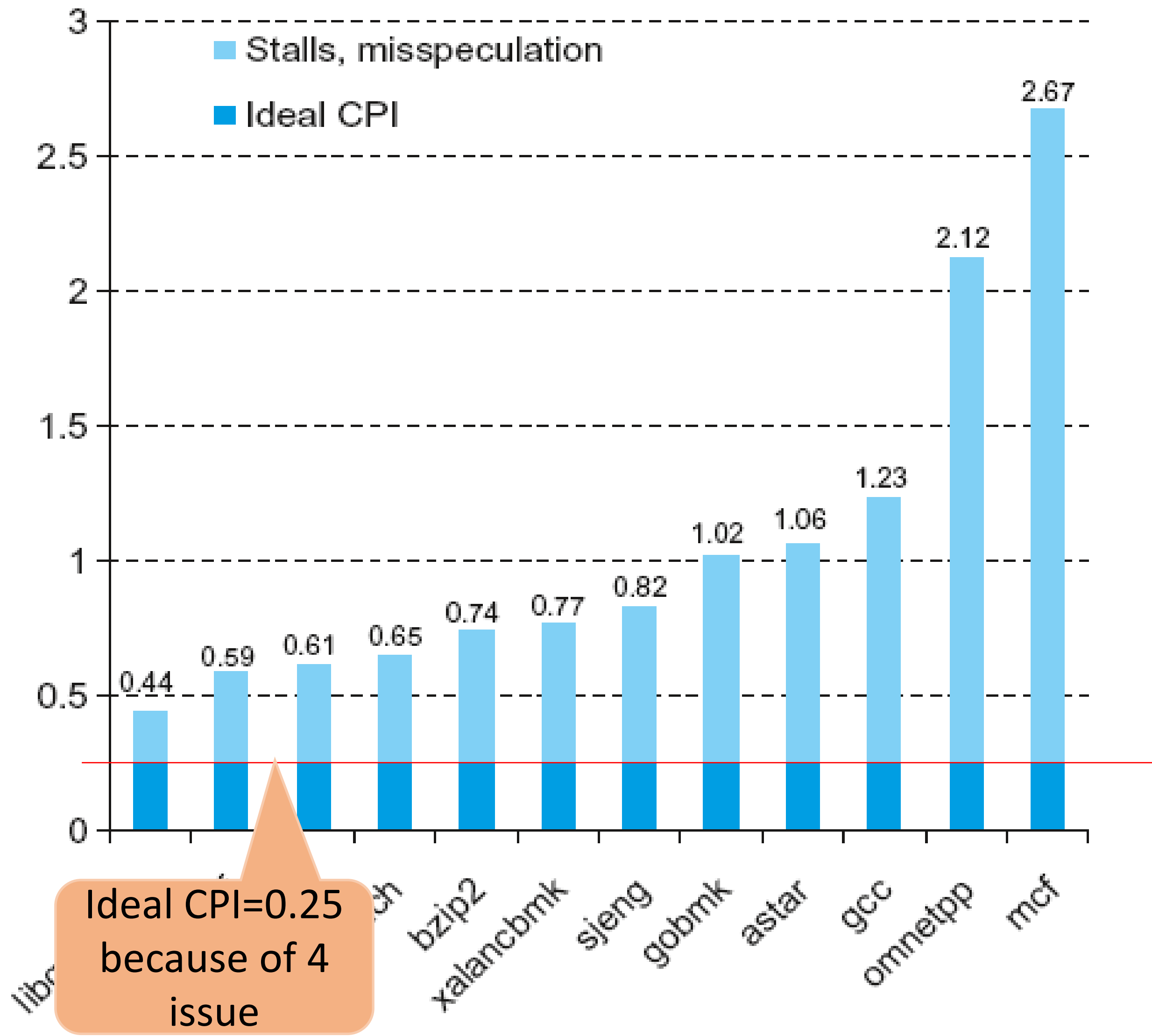
Processor	ARM A53	Intel Core i7 920
Market	Personal Mobile Device	Server, cloud
Thermal design power	100 milliWatts (1 core @ 1 GHz)	130 Watts
Clock rate	1.5 GHz	2.66 GHz
Cores/Chip	4 (configurable)	4
Floating point?	Yes	Yes
Multiple issue?	Dynamic	Dynamic
Peak instructions/clock cycle	2	4
Pipeline stages	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 caches/core	128-2048 KiB	256 KiB (per core)
3rd level caches (shared)	(platform dependent)	2-8 MB

Real world processor performance



ARM Cortex-A53

Ideal CPI=0.5 because of 2 issue



Ideal CPI=0.25 because of 4 issue

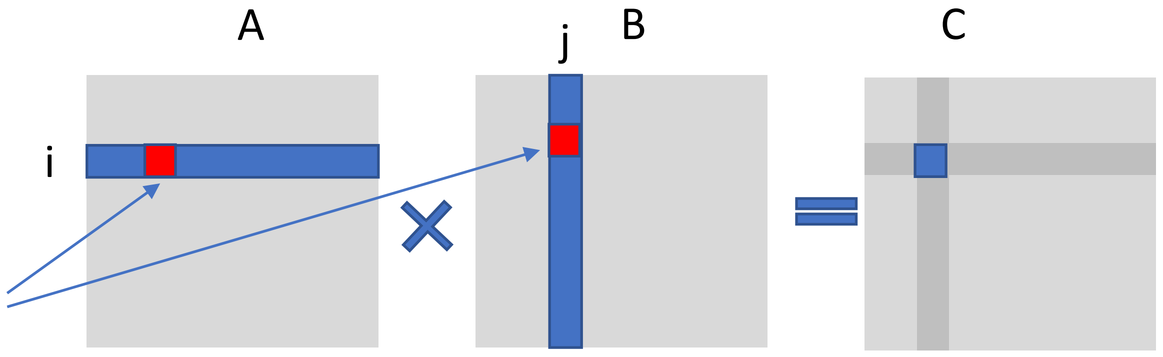
Intel core i7

Ways to improve performance: subword parallelism

- ML/graphics applications perform same operations on vectors
  - Partition a 128-bit adder into:
    - Sixteen 8-bit adds
    - Eight 16-bit adds
    - Four 32-bit adds
- Also called data-level parallelism, vector parallelism, or Single Instruction, Multiple Data (SIMD)
- Intel’s AVX introduces 256-bit floating point registers
    - 8 single-precision ops
    - 4 double-precision ops

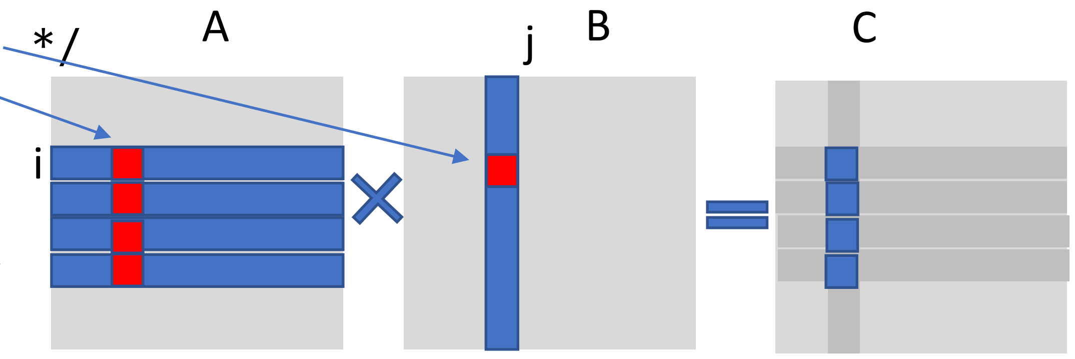
Example: unoptimized matrix multiplication

```
void dgemm (int n, double* A, double* B, double* C)
{
 for (int i = 0; i < n; ++i)
   for (int j = 0; j < n; ++j)
   {
      double cij = C[i+j*n];
      for(int k = 0; k < n; k++ )
        cij += A[i+k*n] * B[k+j*n];
      C[i+j*n] = cij;
   }
}
```

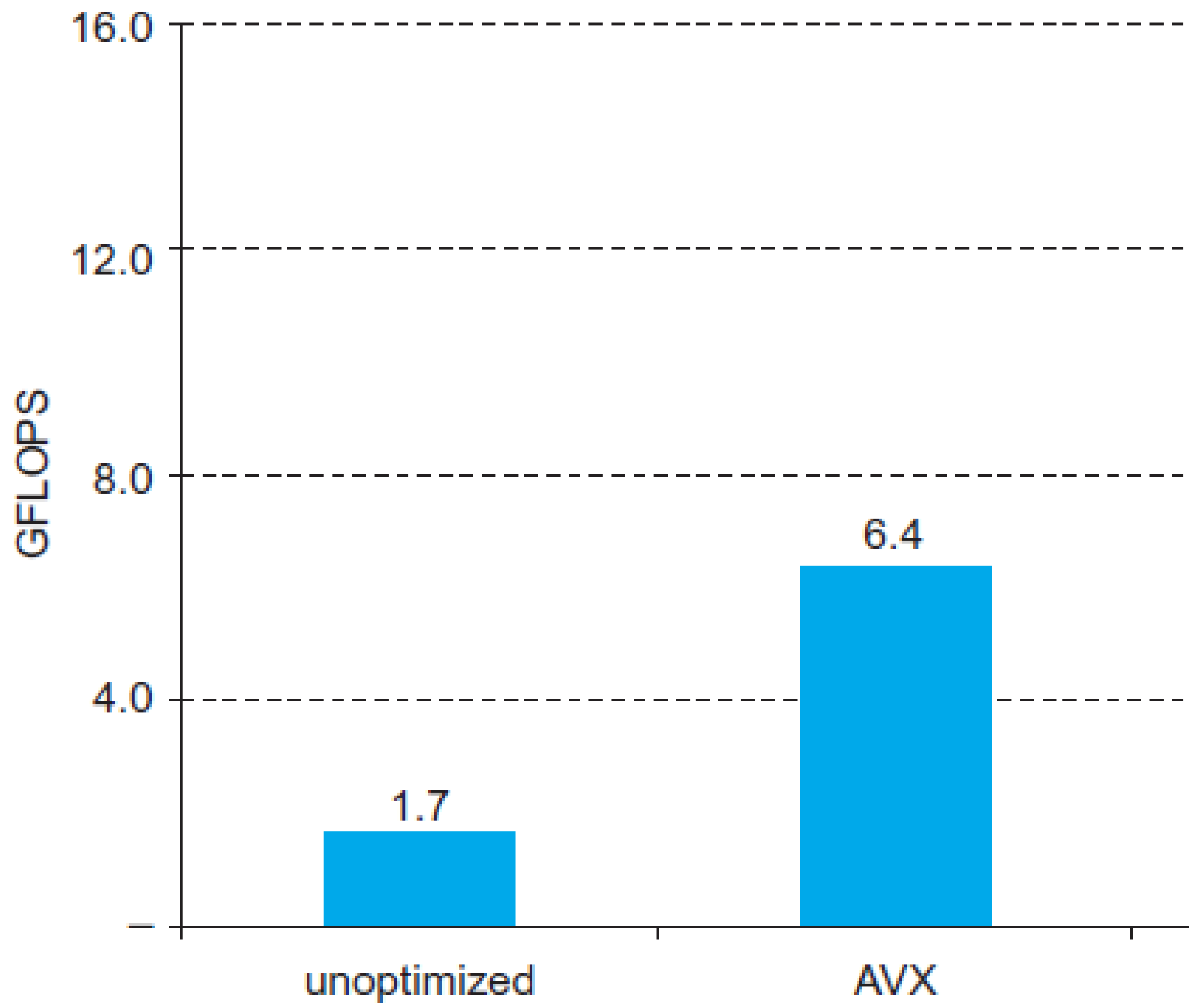


Matrix Multiply: using AVX SIMD

```
#include <x86intrin.h>
void dgemm (int n, double* A, double* B, double* C)
{
 for ( int i = 0; i < n; i+=4 )
  for ( int j = 0; j < n; j++ ) {
   __m256d c0 = _mm256_load_pd(C+i+j*n); /* c0 = C[i][j] */
   for( int k = 0; k < n; k++ )
    c0 = _mm256_add_pd(c0, /* c0 += A[i][k]*B[k][j] */
             _mm256_mul_pd(_mm256_load_pd(A+i+k*n),
             _mm256_broadcast_sd(B+k+j*n)));
   _mm256_store_pd(C+i+j*n, c0); /* C[i][j] = c0 */
  }
}
```



Performance Impact



Summary on CPU design

- ISA influences design of datapath and control
- Datapath and control influence design of ISA
- Pipelining improves instruction throughput using parallelism
  - More instructions completed per second
  - Latency for each instruction not reduced
- Hazards: structural, data, control
- Multiple issue and dynamic scheduling (ILP)
  - Dependencies limit achievable parallelism
  - Complexity leads to the power wall