Pipelined CPU

Jinyang Li

Slides are based on Patterson and Hennessy

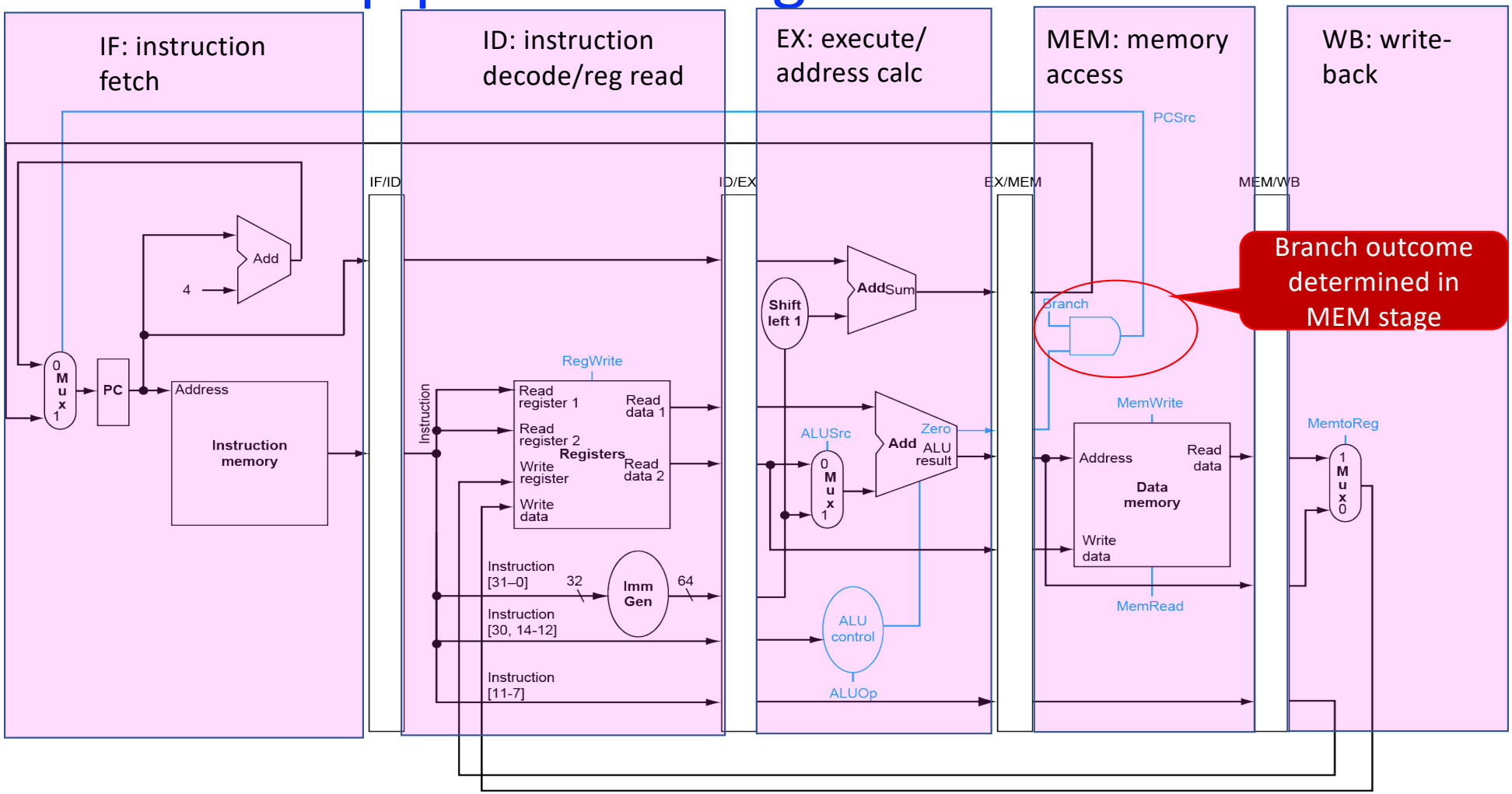
What we've learnt so far

- Single cycle RISC-V CPU design
- 5 stage pipelined RISC-V CPU
  - Why pipelining? Faster (ideal throughput speedup: #-of-stages)
- Pipelining challenges: hazards
  - Structure (how to avoid stall? add resources)
  - Data (how to minimize stall? forwarding)
  - Control

Today's lesson

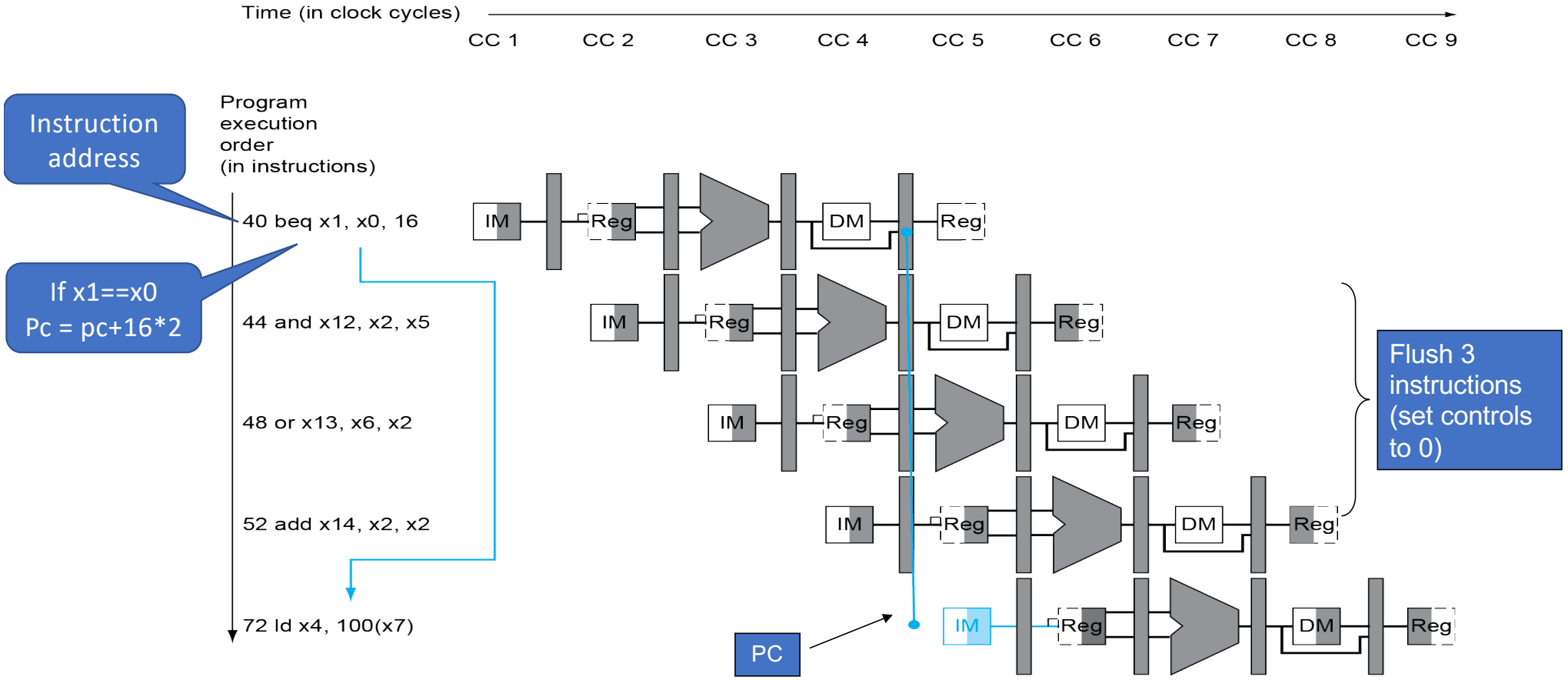
- Handling control (branch) hazard
- Handling exceptions
- Briefly, more advanced topics:
  - Multiple-issue
  - Subword parallelism (SIMD)

Recall our pipelined design so far



Branch Hazard

- If branch outcome is determined in MEM

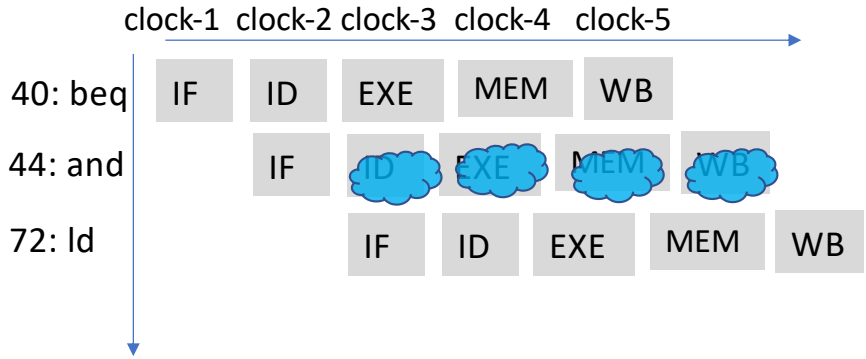


Reducing Branch Delay

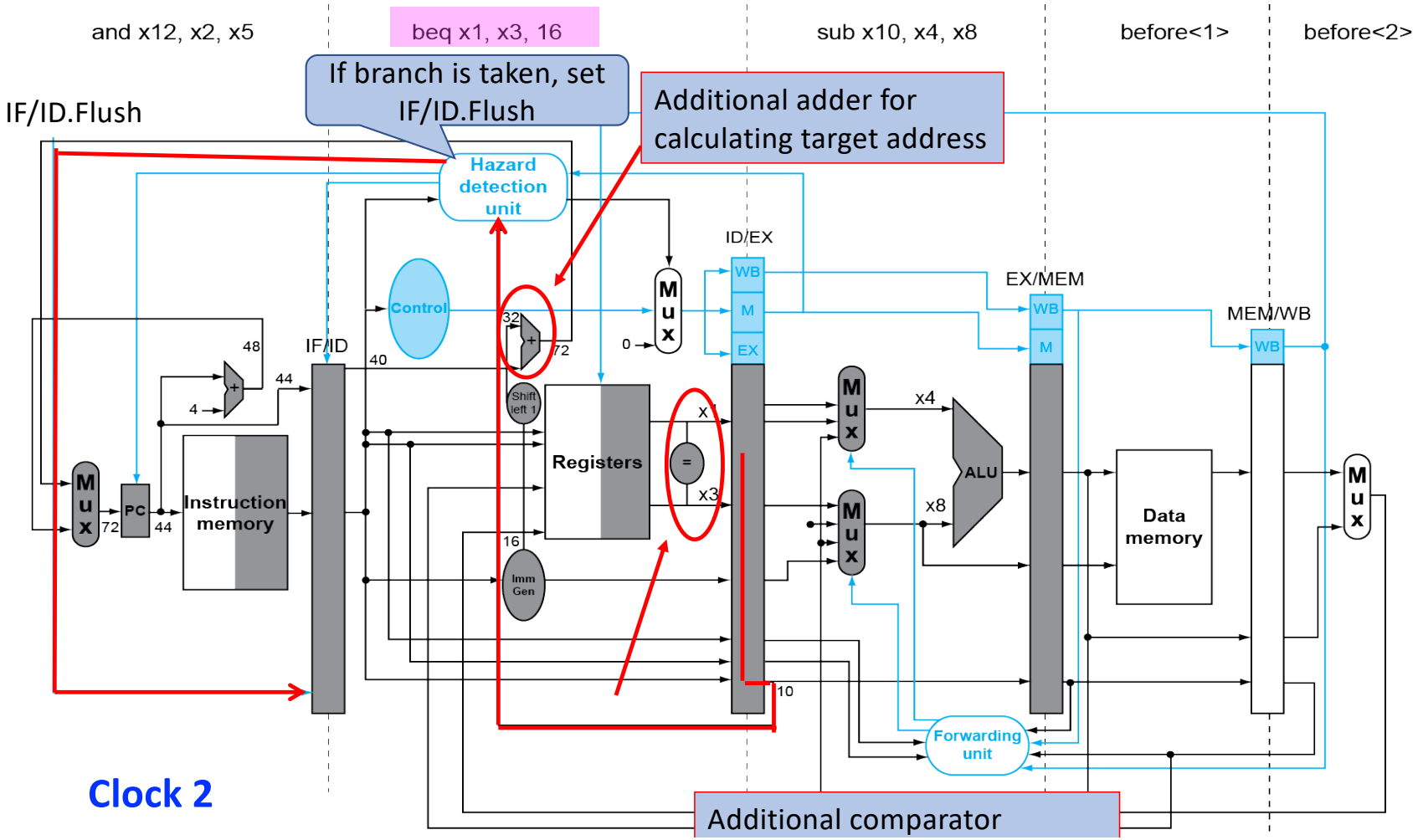
- Add hardware to determine branch outcome earlier (e.g. ID instead of MEM) → fewer instructions to flush

```
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)
```

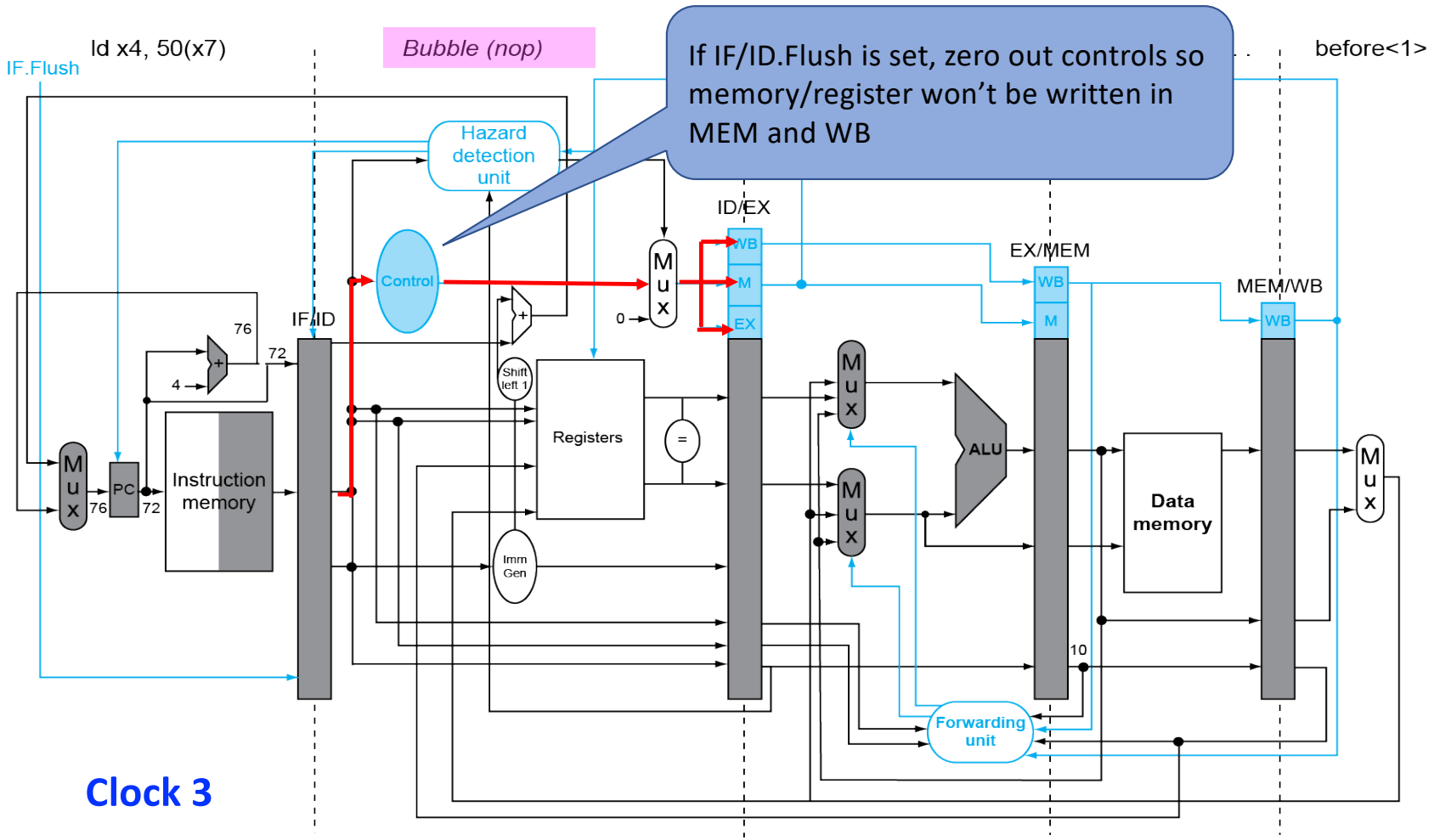
How many instructions to flush if branch outcome is known in ID?



Branch determined in ID and is taken



Branch determined in ID and is taken

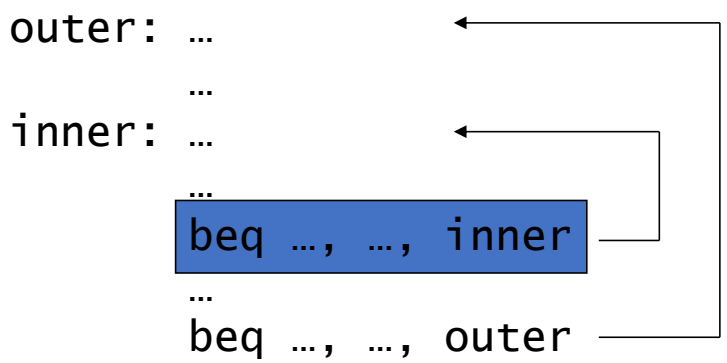


Dynamic Branch Prediction

- 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

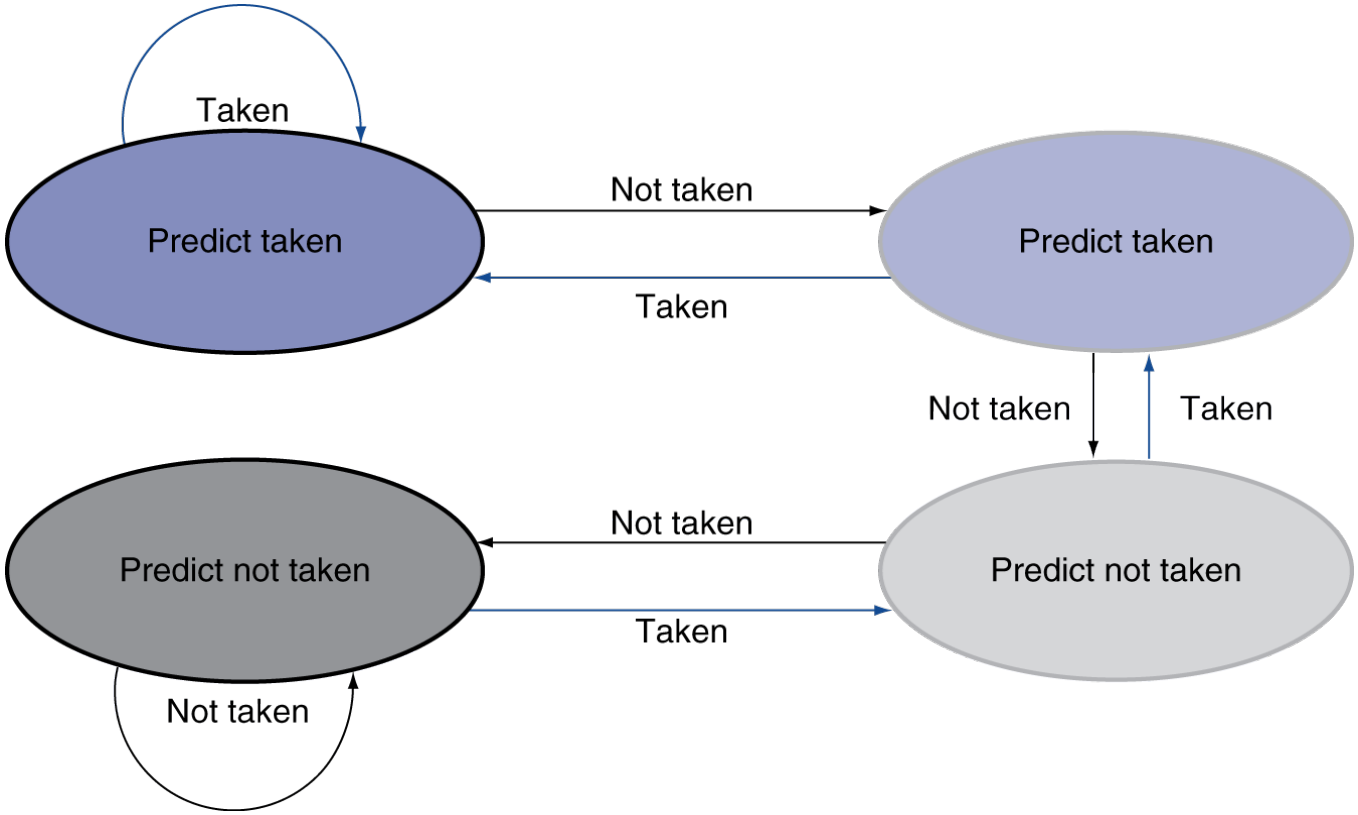
- Inner loop branches mispredicted twice!



  - 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 I/O controller
- 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 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`
  - 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

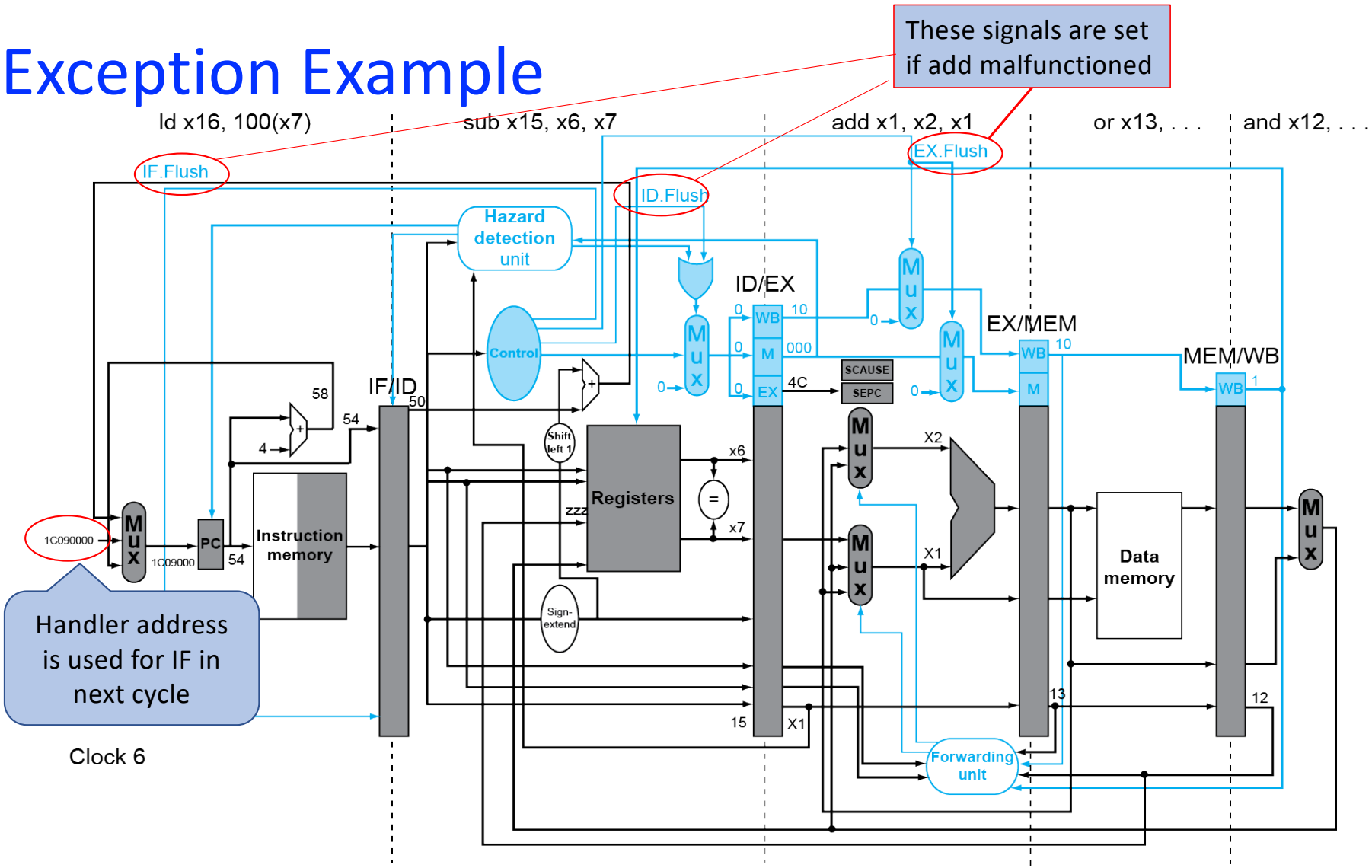
- Exception on `add` in

```
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)
…
```

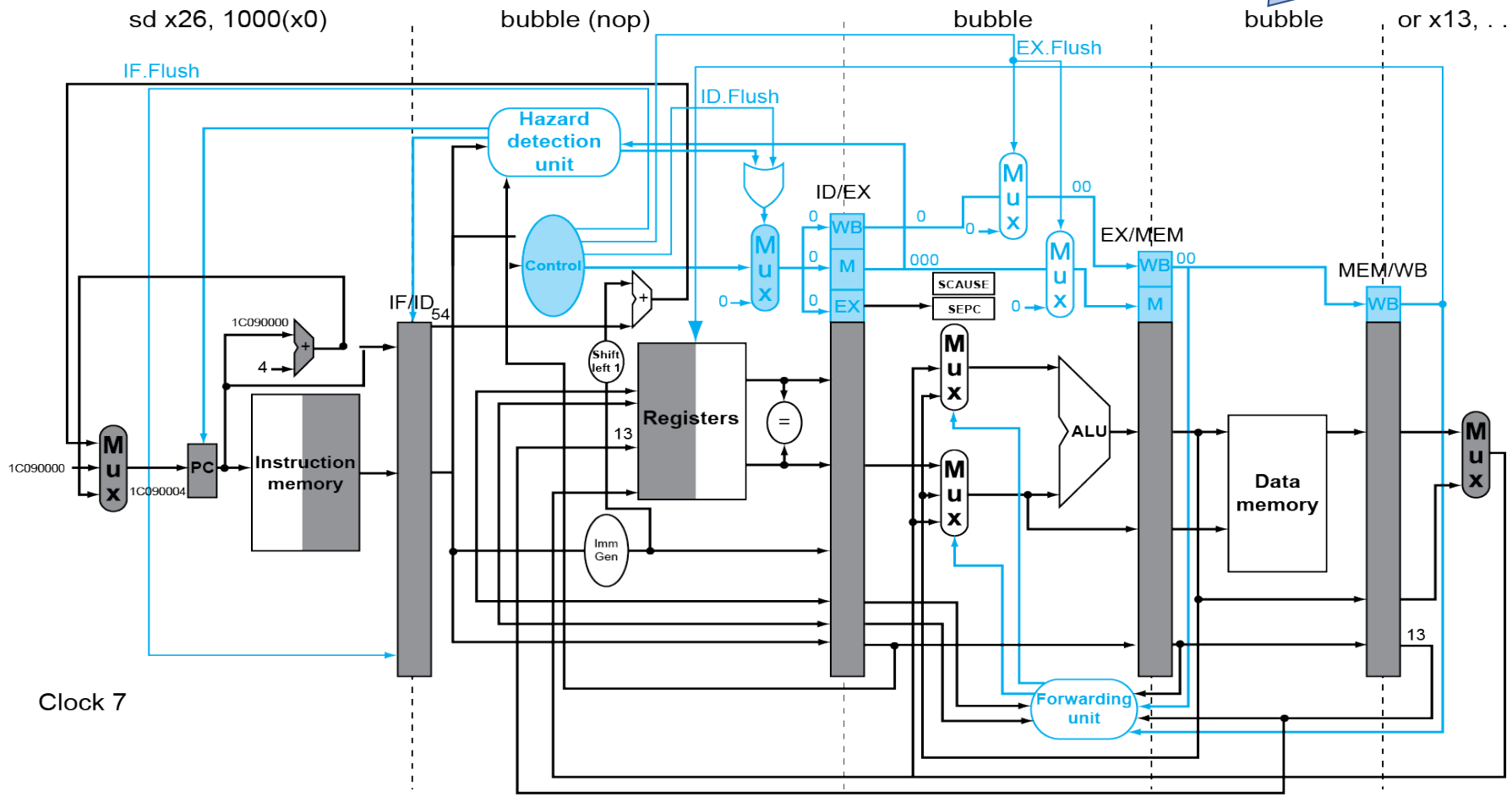
Exception Example



Exception Example

Instructions in exception handler

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



Instruction-Level Parallelism (ILP)

- Pipelining: execute multiple instructions in parallel
- To increase ILP
  - Deeper pipeline
    - Less work per stage ⇒ shorter clock cycle
  - Multiple issue
    - Replicate pipeline stages ⇒ 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 this

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

Static Multiple Issue

- Compiler groups instructions into “issue packets”
  - Group >1 instructions in an issue packet to be issued in a single cycle
  - Instructions within a packet must have no dependencies
    - Reorder
    - Add nop
- Think of an issue packet as a very long instruction
  - Specifies multiple concurrent operations
  - $\Rightarrow$ Very Long Instruction Word (VLIW)

Dynamic Multiple Issue

- “Superscalar” processors
- CPU decides whether to issue 0, 1, 2, … each cycle
  - Avoiding structural and data hazards
- Avoids the need for compiler scheduling
  - Though it may still help
  - Code semantics ensured by the CPU

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

Does Multiple Issue Work?

- Yes, but not as much as we'd like
- Programs have real dependencies that limit ILP
- Some dependencies are hard to eliminate
  - e.g., pointer aliasing
- Memory delays and limited bandwidth
  - Hard to keep pipelines full
- Speculation can help if done well

Multiple-issue benefits from loop unrolling

- Replicate loop body to expose more parallelism
  - Reduces loop-control overhead
  - Avoid loop-carried dependency
    - Store followed by a load of the same register

```
long x[100];
…
for (int i = 0; i < 100; i++) {
    x[i] *= 2;
}
```

```
//suppose x1=&x[0]
//x2 = &x[100]
Loop:
    ld  x31, 0(x1)
    add x31, x31, x30
    sd  x30, 0(x1)
    addi x1, x1, 8
    blt x1, x2, Loop
```

```
//suppose x20=&x[0]
//x22 = &x[100]
Loop:
    ld  x31, 0(x1)
    add x31, x31, x30
    sd  x30, 0(x1)
    ld  x29, 8(x1)
    add x29, x29, x28
    sd  x28, 8(x1)
    addi x20, x20, 16
    blt x20, x22, Loop
```

Power Efficiency

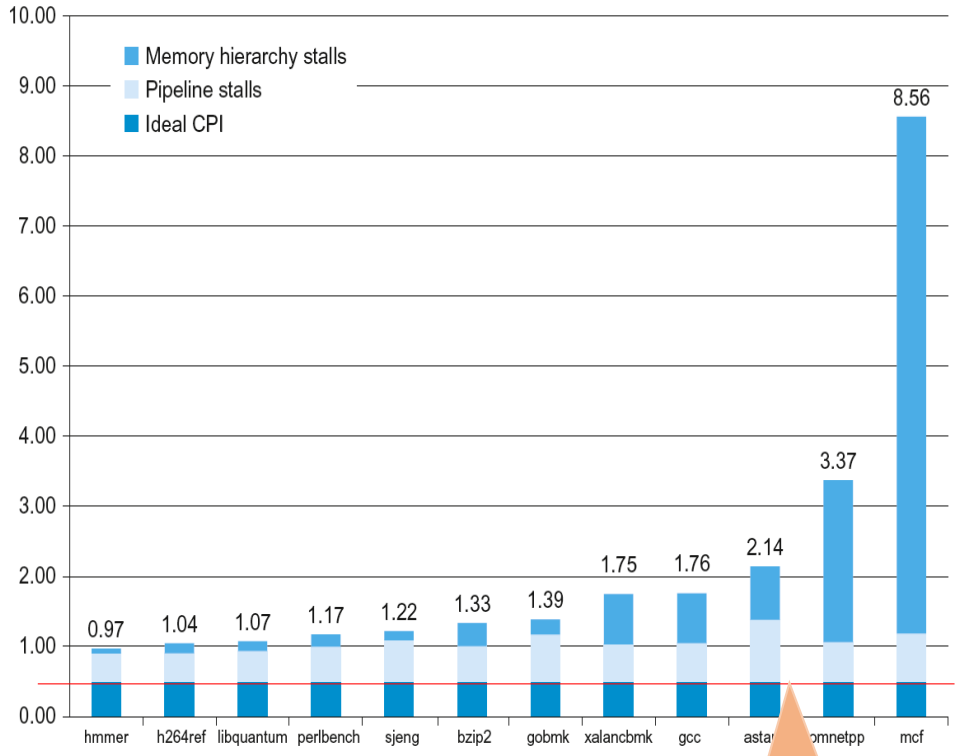
- Complexity of dynamic scheduling and speculations requires power
- Multiple simpler cores may be better

Microprocessor	Year	Clock Rate	Pipeline Stages	Issue width	Out-of-order/ Speculation	Cores	Power
i486	1989	25MHz	5	1	No	1	5W
Pentium	1993	66MHz	5	2	No	1	10W
Pentium Pro	1997	200MHz	10	3	Yes	1	29W
P4 Willamette	2001	2000MHz	22	3	Yes	1	75W
P4 Prescott	2004	3600MHz	31	3	Yes	1	103W
Core	2006	2930MHz	14	4	Yes	2	75W
UltraSparc III	2003	1950MHz	14	4	No	1	90W
UltraSparc T1	2005	1200MHz	6	1	No	8	70W

ARM Cortex A53 and Intel i7

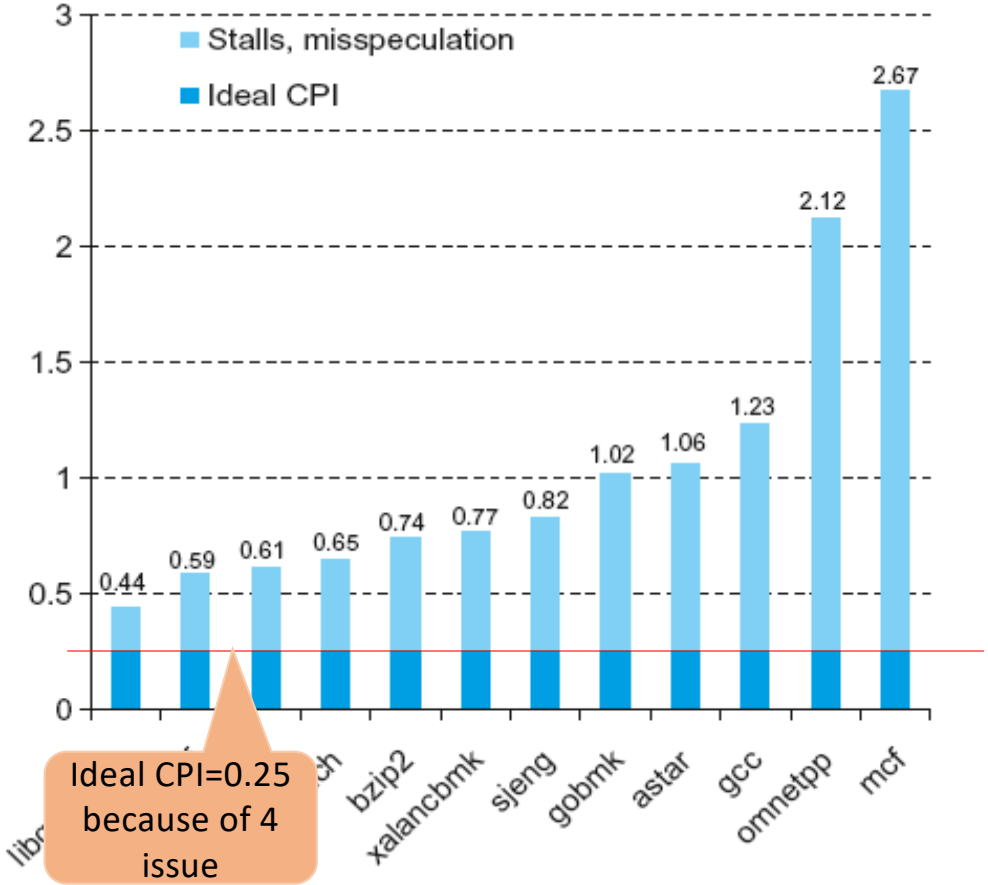
Processor	ARM A53	Intel Core i7 920
Market	Personal Mobile Device	Server, cloud
Thermal design power	100 milliWatts (1 core @ 1 GHz)	130 Watts
Clock rate	1.5 GHz	2.66 GHz
Cores/Chip	4 (configurable)	4
Floating point?	Yes	Yes
Multiple issue?	Dynamic	Dynamic
Peak instructions/clock cycle	2	4
Pipeline stages	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

```
1. void dgemm (int n, double* A, double* B, double* C)
2. {
3.  for (int i = 0; i < n; ++i)
4.    for (int j = 0; j < n; ++j)
5.    {
6.       double cij = C[i+j*n];
7.       for(int k = 0; k < n; k++ )
8.          cij += A[i+k*n] * B[k+j*n];
9.       C[i+j*n] = cij;
10.   }
11. }
```

Matrix Multiply: optimized C

```
1. #include <x86intrin.h>
2. void dgemm (int n, double* A, double* B, double* C)
3. {
4.  for ( int i = 0; i < n; i+=4 )
5.   for ( int j = 0; j < n; j++ ) {
6.    __m256d c0 = _mm256_load_pd(C+i+j*n); /* c0 = C[i][j] */
7.    for( int k = 0; k < n; k++ )
8.     c0 = _mm256_add_pd(c0, /* c0 += A[i][k]*B[k][j] */
9.               _mm256_mul_pd(_mm256_load_pd(A+i+k*n),
10.              _mm256_broadcast_sd(B+k+j*n)));
11.   _mm256_store_pd(C+i+j*n, c0); /* C[i][j] = c0 */
12.  }
13. }
```

AVX Optimized Matrix multiply


```
1. vmovapd (%r11),%ymm0       # Load 4 elements of C into %ymm0
2. mov %rbx,%rcx              # register %rcx = %rbx
3. xor %eax,%eax              # register %eax = 0
4. vbroadcastsd (%rax,%r8,1),%ymm1 # Make 4 copies of B element
5. add $0x8,%rax              # register %rax = %rax + 8
6. vmulpd (%rcx),%ymm1,%ymm1 # Parallel mul %ymm1,4 A elements
7. add %r9,%rcx               # register %rcx = %rcx + %r9
8. cmp %r10,%rax              # compare %r10 to %rax
9. vaddpd %ymm1,%ymm0,%ymm0  # Parallel add %ymm1, %ymm0
10. jne 50 <dgemm+0x50>       # jump if not %r10 != %rax
11. add $0x1,%esi             # register % esi = % esi + 1
12. vmovapd %ymm0,(%r11)      # Store %ymm0 into 4 C elements
```

Ways to improve performance: loop unrolling

- Replicate loop body to expose more parallelism
  - Reduces loop-control overhead
  - Avoid loop-carried dependency
    - Store followed by a load of the same register

Ways to improve performance: loop unrolling

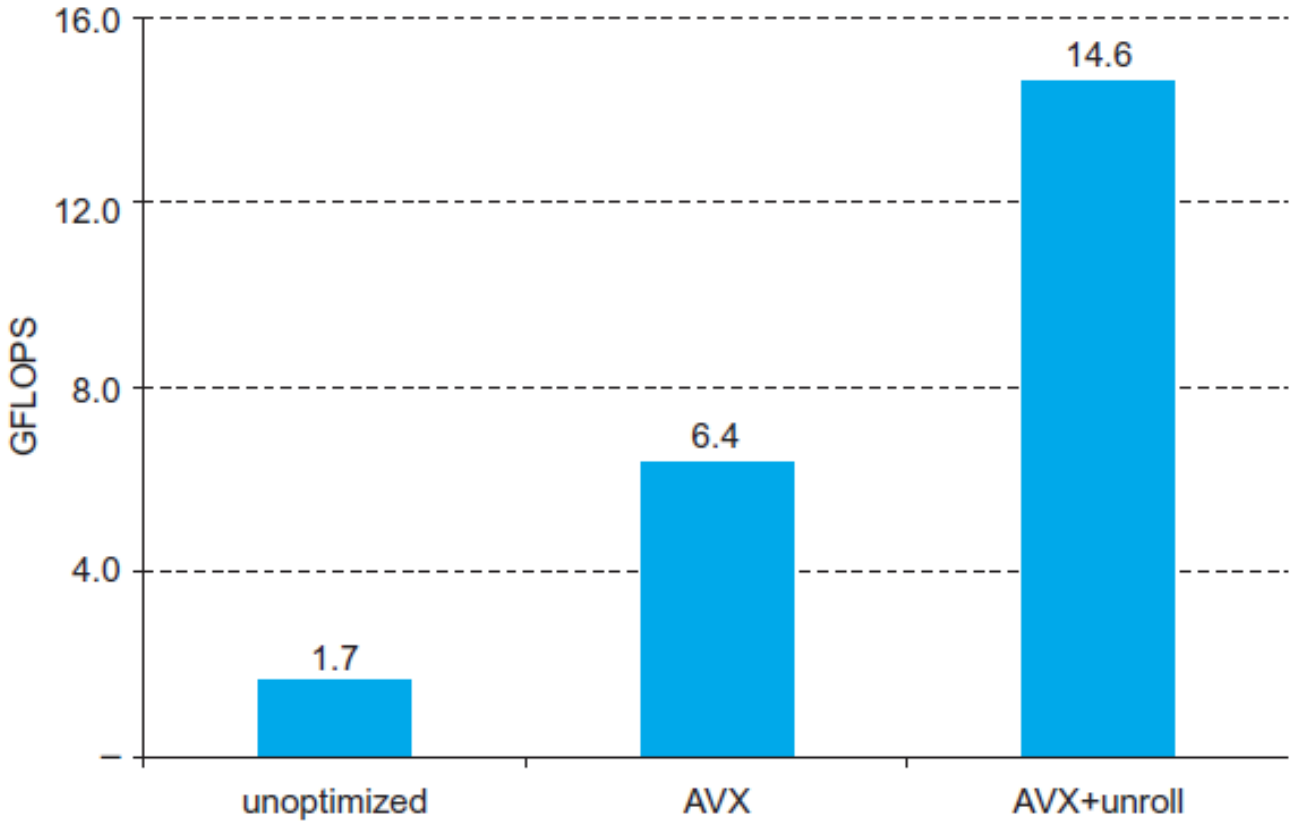
- Loop unrolling: replicate loop body
- Unrolling exposes more parallelism for multi-issue pipelined CPU
  - Reduces loop-control overhead
  - Avoid loop-carried dependency
    - Store followed by a load of the same register

```
long x[100];
…
for (int i = 0; i < 100; i++) {
    x[i] *= 2;
}
```

```
//suppose x1=&x[0]
//x2 = &x[100]
Loop:
    ld  x31, 0(x1)
    add x31, x31, x30
    sd  x30, 0(x1)
    addi x1, x1, 8
    blt x1, x2, Loop
```

```
//suppose x20=&x[0]
//x22 = &x[100]
Loop:
    ld  x31, 0(x1)
    add x31, x31, x30
    sd  x30, 0(x1)
    ld  x29, 8(x1)
    add x29, x29, x28
    sd  x28, 8(x1)
    addi x20, x20, 16
    blt x20, x22, Loop
```

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