Data Hazards

Advanced Pipelining and I/O

- Data hazards: forwarding vs. stalling
- Control hazards
- Exceptions
- Parallelism via instructions
- ARM Cortex A8 and Intel i7 pipelines
- I/O system characteristics
- Interconnecting components
- Parallelism and I/O
- I/O system design

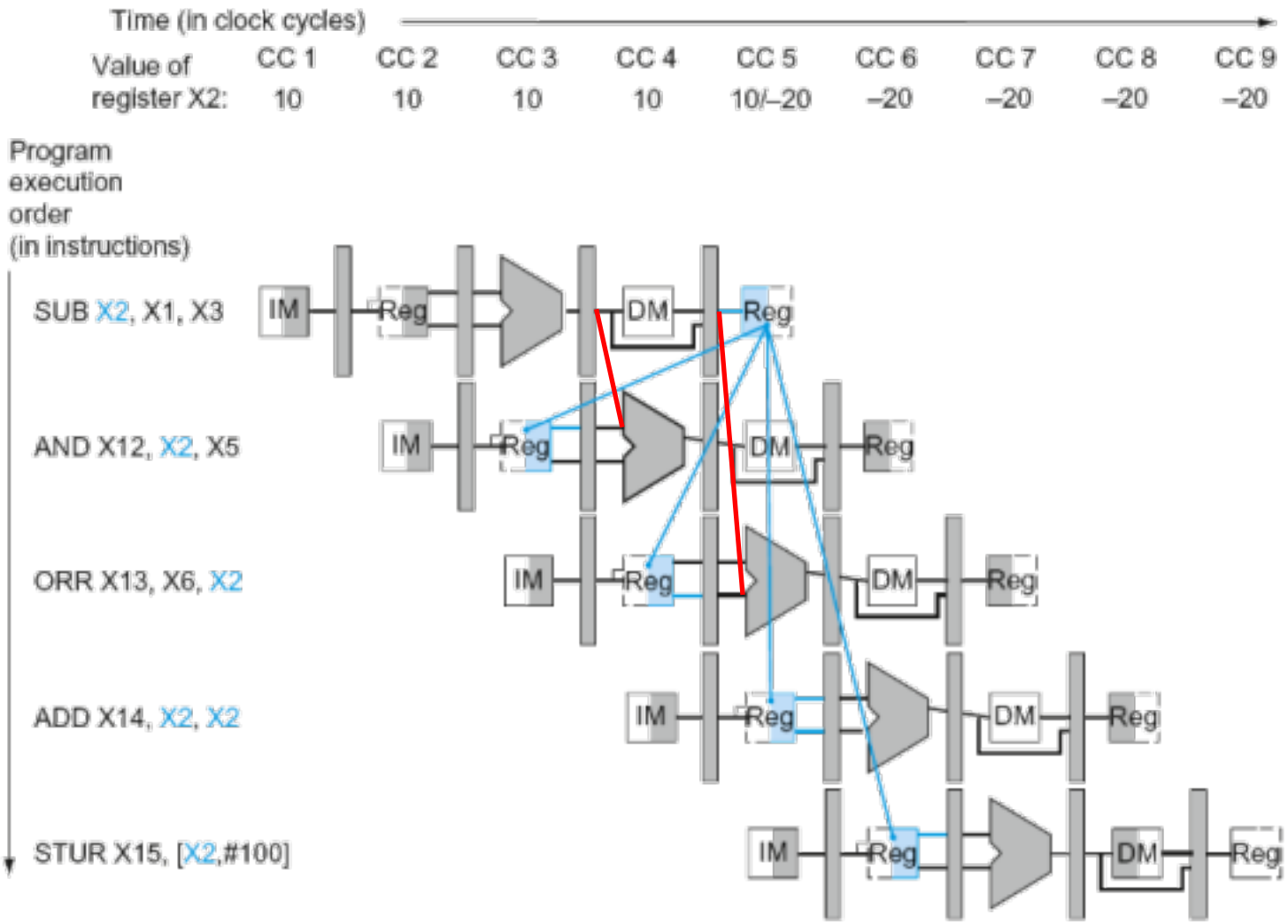
Data Hazards in ALU Instructions

- Consider this sequence:

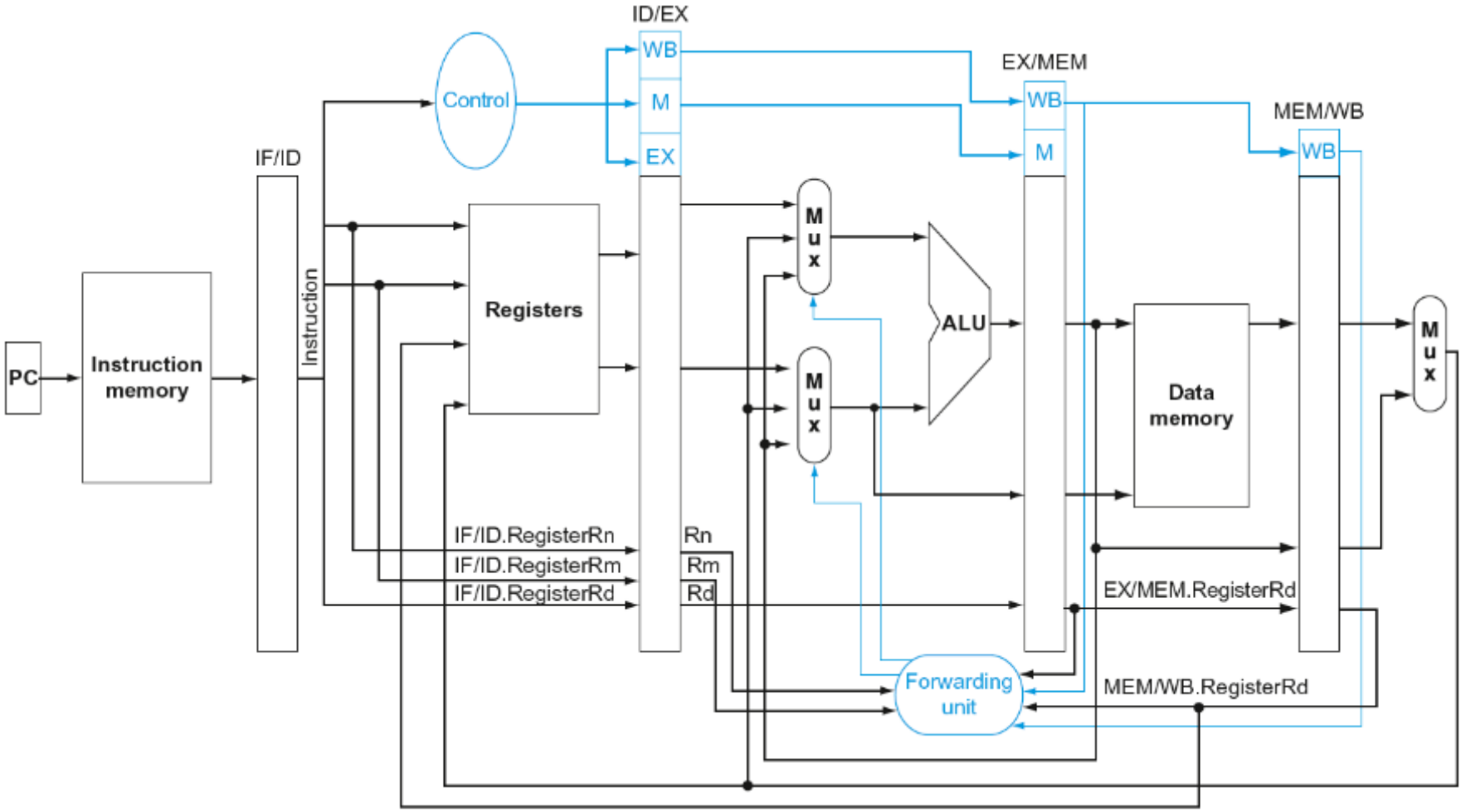
```
SUB   x2, x1,x3
AND   x12,x2,x5
OR    x13,x6,x2
ADD   x14,x2,x2
STUR  x15,[x2,#100]
```

- We can resolve hazards with forwarding.
  - How do we detect when to forward?

Dependencies and Forwarding



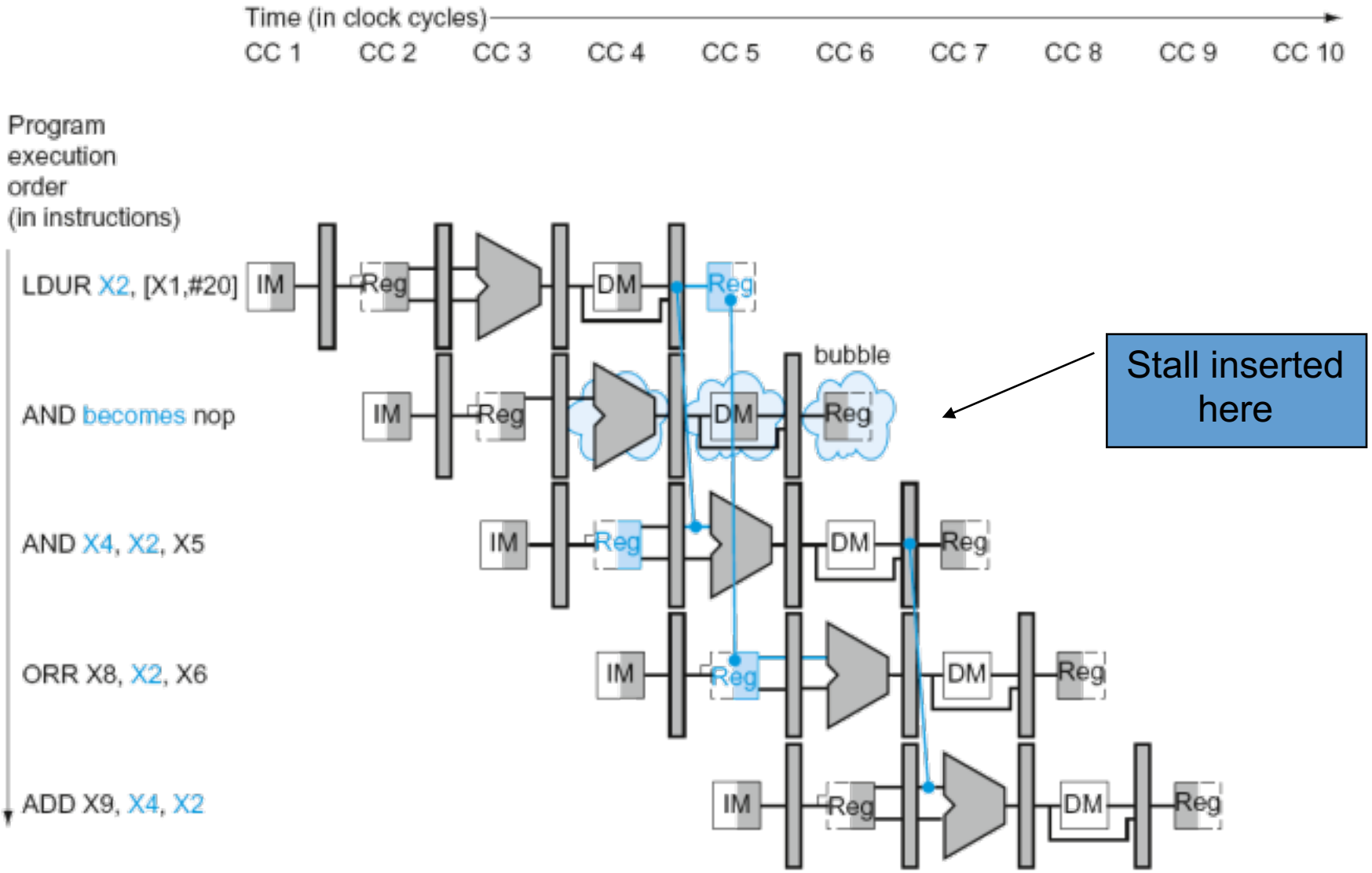
Datapath with Forwarding



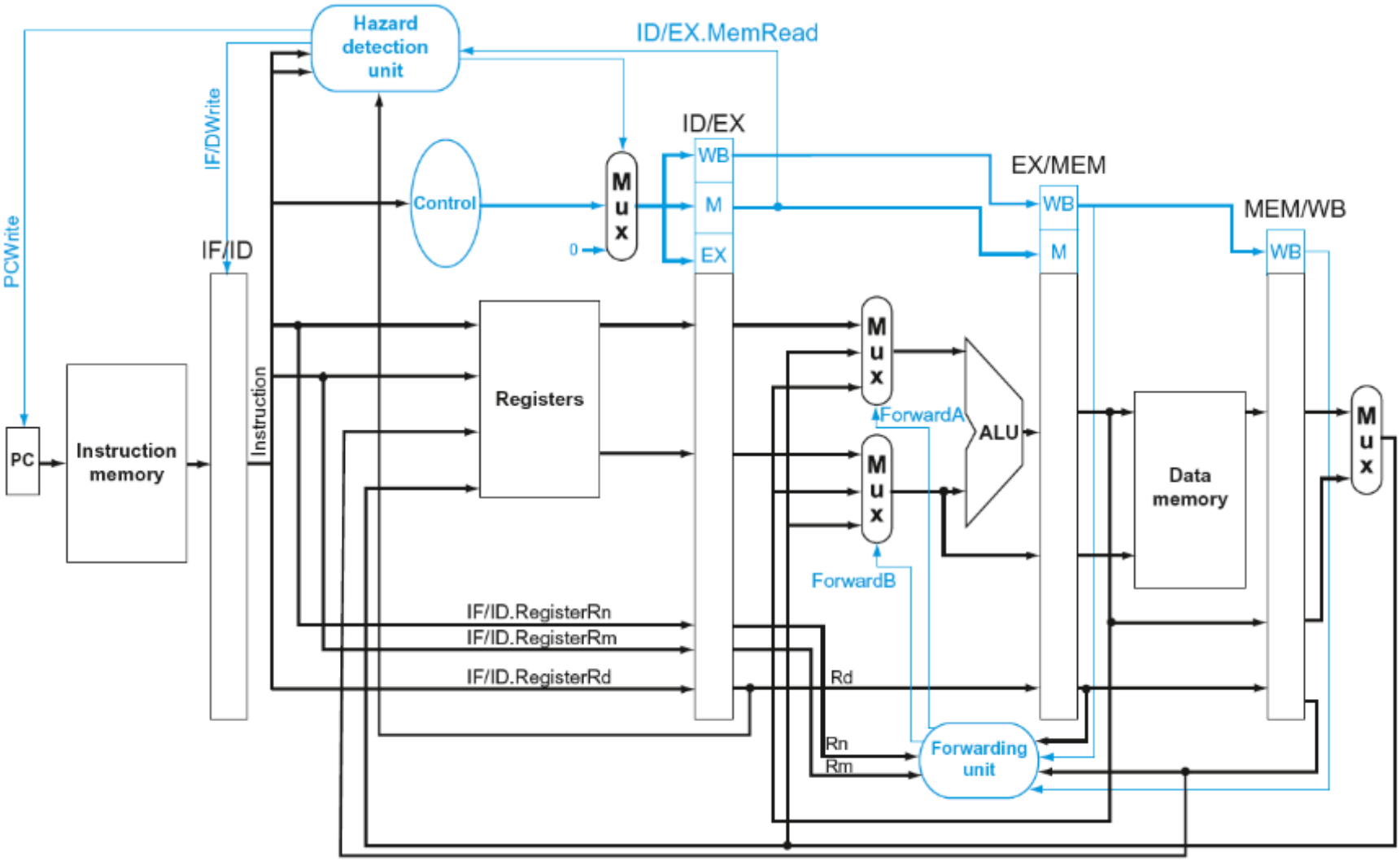
How to Stall the Pipeline

- Force control values in ID/EX register to 0.
  - EX, MEM, and WB do `nop` (no-operation).
- Prevent update of PC and IF/ID register.
  - Using instruction is decoded again.
  - Following instruction is fetched again.
  - One-cycle stall allows MEM to read data for `LDUI`.
    - Can subsequently forward to EX stage

Load-Use Data Hazard



Datapath with Hazard Detection



Stalls and Performance

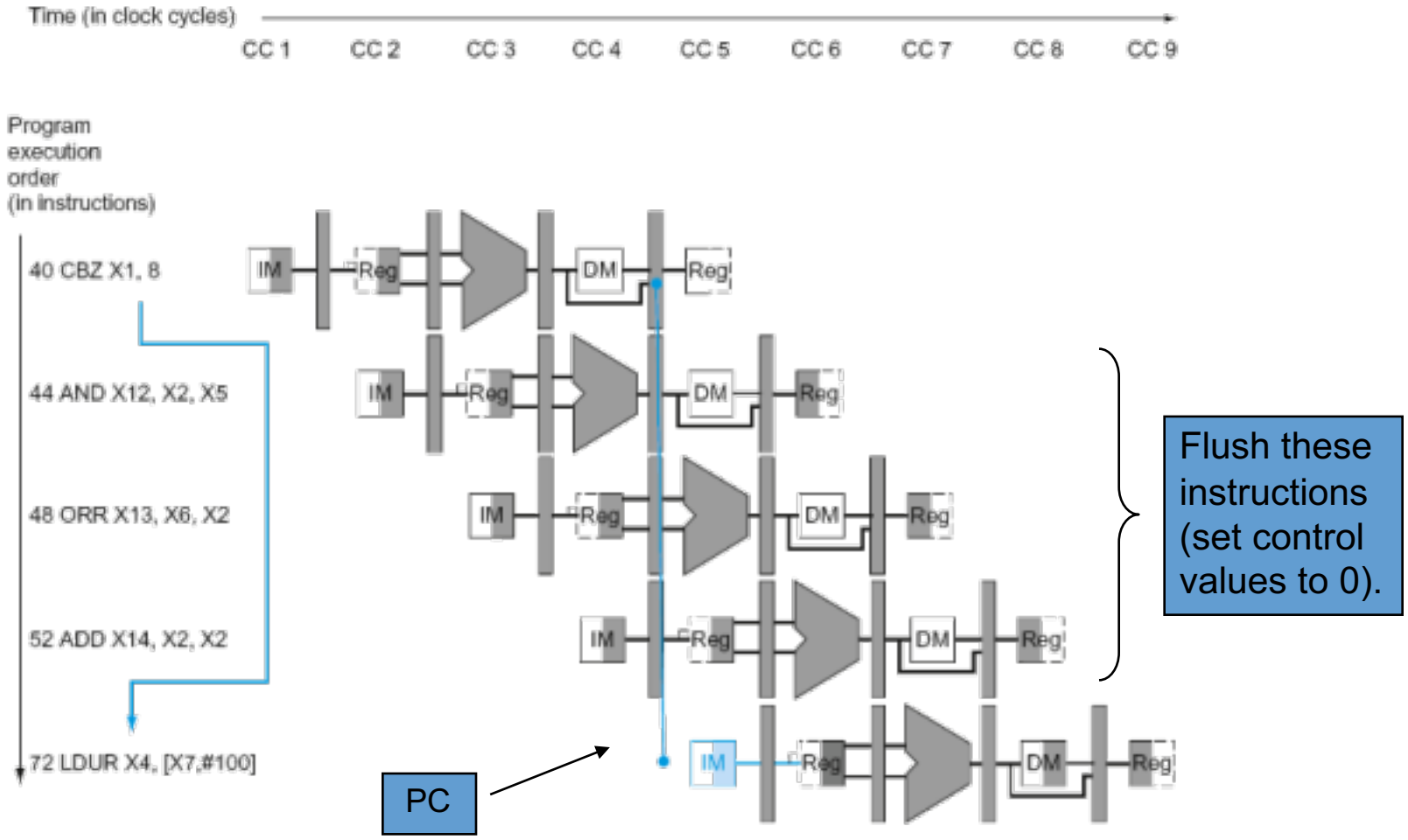
- Stalls reduce performance.
  - But are required to get correct results
- Compiler can arrange code to avoid hazards and stalls.
  - Requires knowledge of the pipeline structure

ENGINEERING@SYRACUSE

Control Hazards

Branch Hazards

- If branch outcome determined in MEM

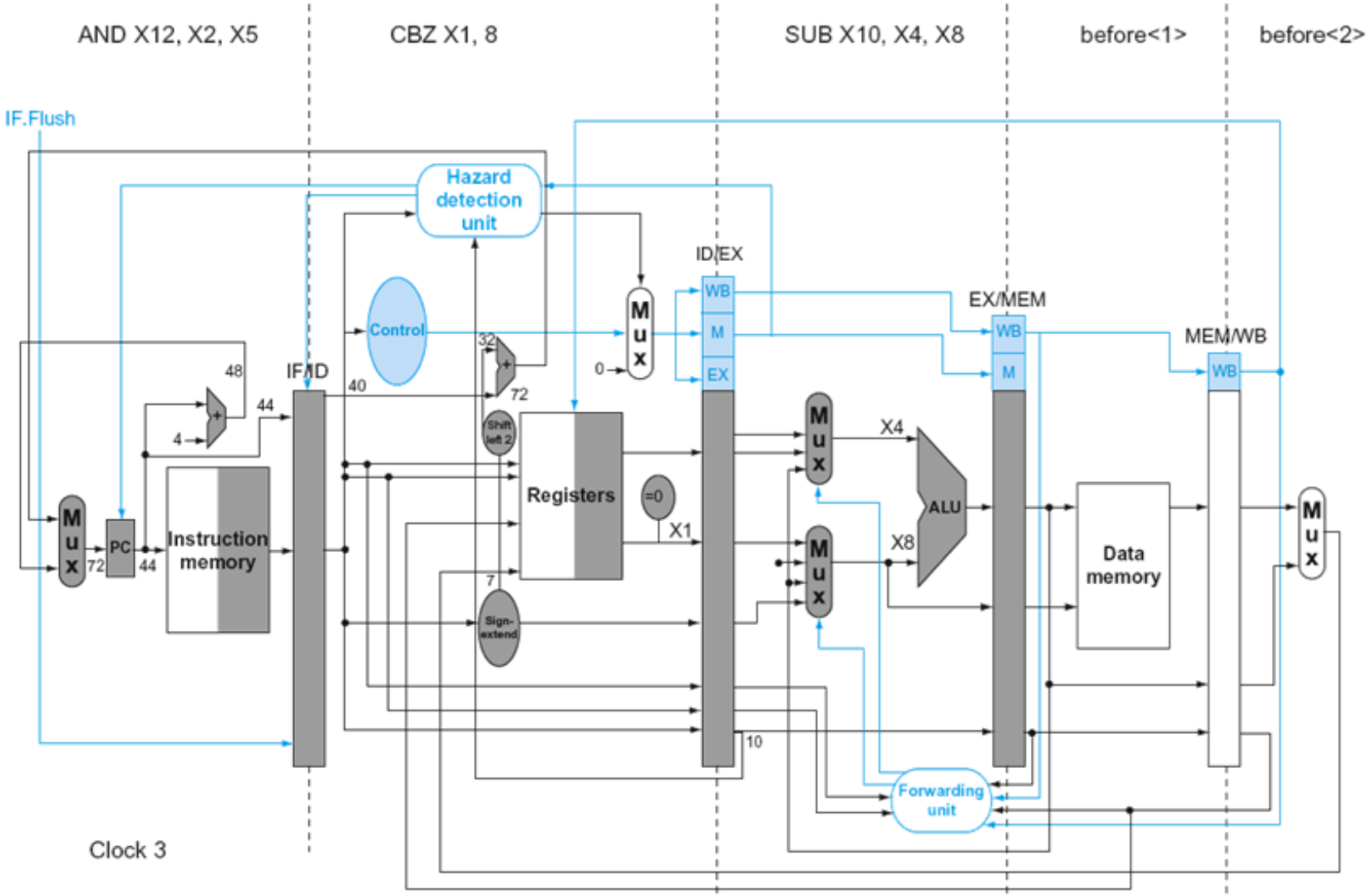


Reducing Branch Delay

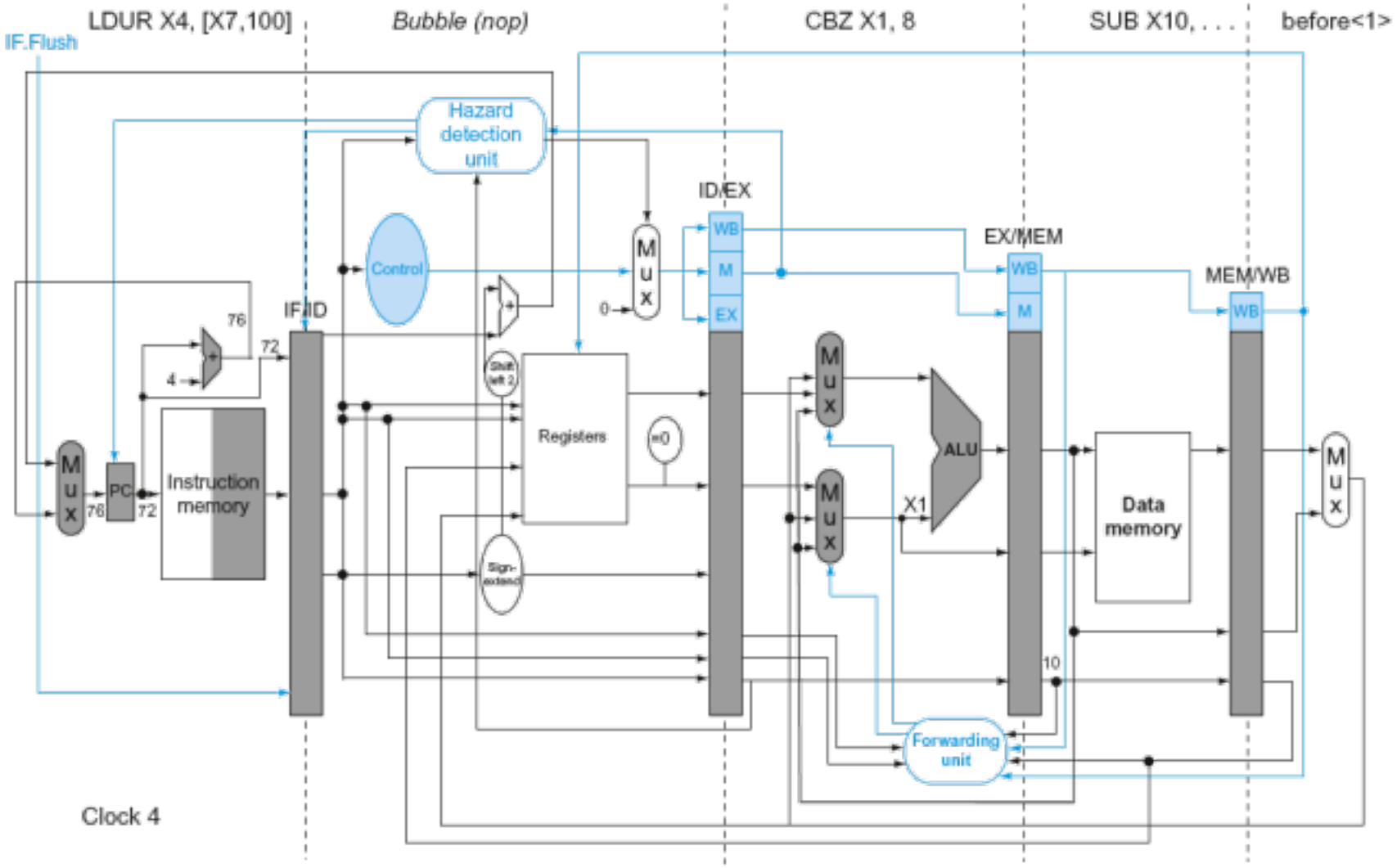
- Move hardware to determine outcome to ID stage.
  - Target address adder
  - Register comparator
- Example: branch taken.

```
36:   SUB  X10, X4, X8
40:   CBZ  X1,  X3, 8
44:   AND  X12, X2, X5
48:   ORR  X13, X2, X6
52:   ADD  X14, X4, X2
56:   SUB  X15, X6, X7
      ...
72:   LDUR X4, [X7,#50]
```

Example: Branch Taken



Example: Branch Taken (cont.)

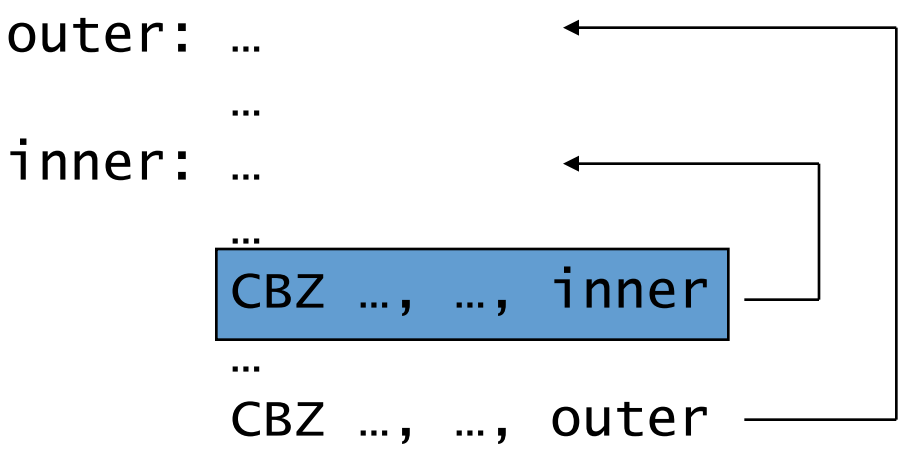


Dynamic Branch Prediction

- In deeper and superscalar pipelines, branch penalty is more significant.
- Use 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.

One-Bit Predictor: Shortcoming

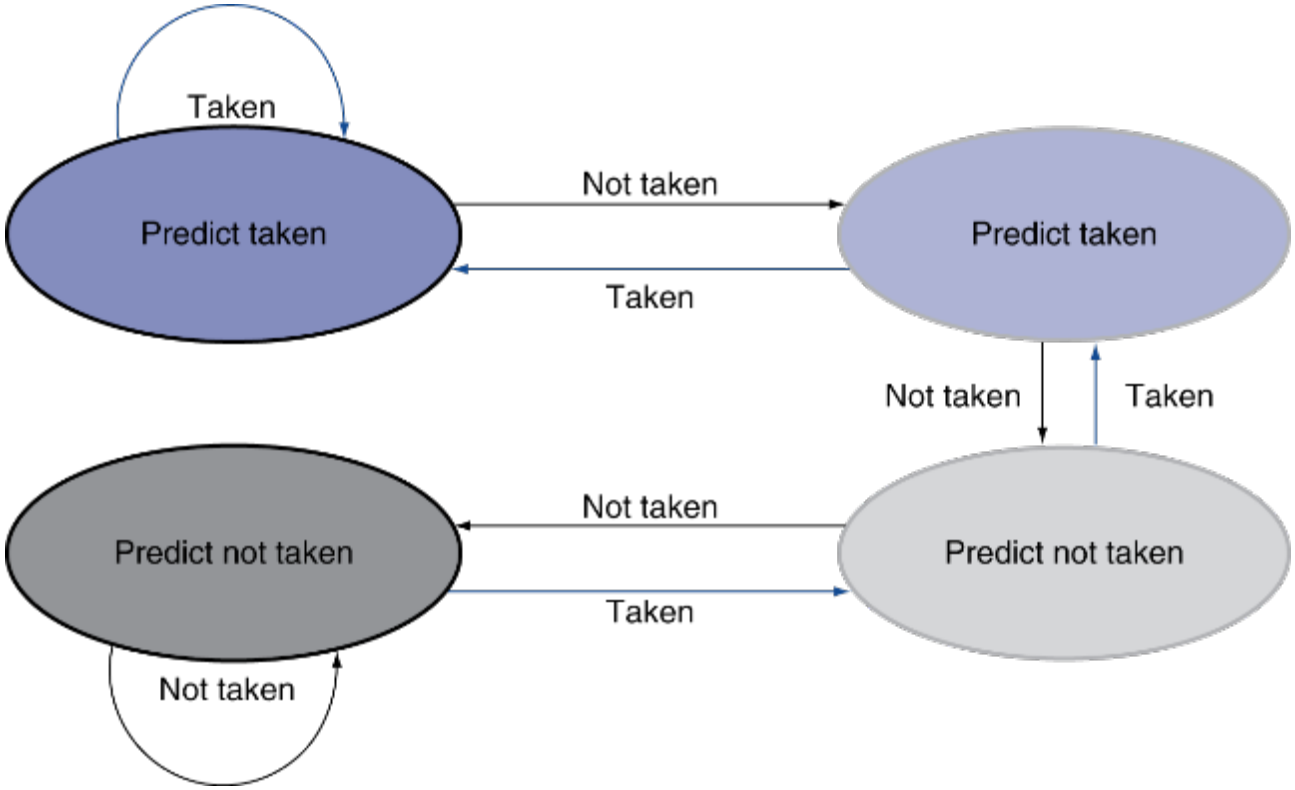
- Inner loop branches mispredicted twice!



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

Two-Bit Predictor

- Only change prediction on two successive mispredictions



Calculating the Branch Target

- Even with predictor, still need to calculate the target address
  - One-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

ENGINEERING@SYRACUSE

Exceptions

Exceptions and Interrupts

- "Unexpected" events requiring change in flow of control.
  - Different ISAs use the terms differently.
- Exception.
  - Arises within the CPU
    - E.g., undefined opcode, overflow, syscall, …
- 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 Av8: exception link register (ELR)
- Save indication of the problem.
  - In Av8: exception syndrome register (ESR).
  - We'll assume one-bit.
    - 0 for undefined opcode, 1 for overflow

Handler Actions

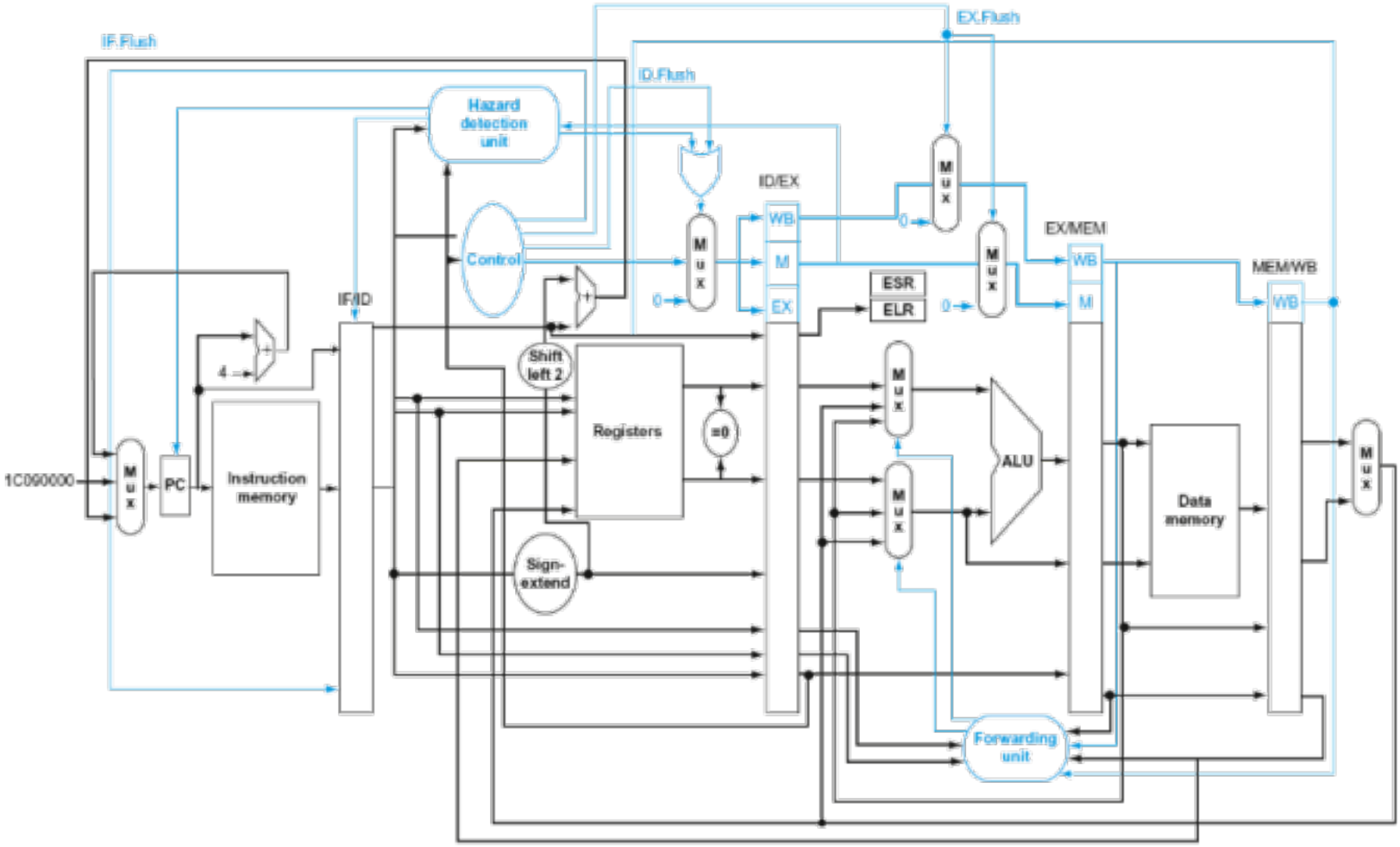
- Read cause, and transfer to relevant handler.
- Determine action required.
- If restartable:
  - Take corrective action.
  - Use EPC to return to program.
- Otherwise:
  - Terminate program.
  - Report error using EPC, cause, …

Exceptions in a Pipeline

- Another form of control hazard.
- Consider overflow on add in EX stage.

  `ADD X1, X2, X1`
  - Prevent X1 from being clobbered.
  - Complete previous instructions.
  - Flush add and subsequent instructions.
  - Set ESR and ELR register values.
  - Transfer control to handler.
- Similar to mispredicted branch.
  - Use much of the same hardware.

Pipeline with Exceptions



Exception Properties

- Restartable exceptions
  - Pipeline can flush the instruction.
  - Handler executes, then returns to the instruction.
    - Refetched and executed from scratch
- PC saved in ELR register
  - Identifies causing instruction.
  - Actually PC + 4 is saved.
    - Handler must adjust.

Exception Example

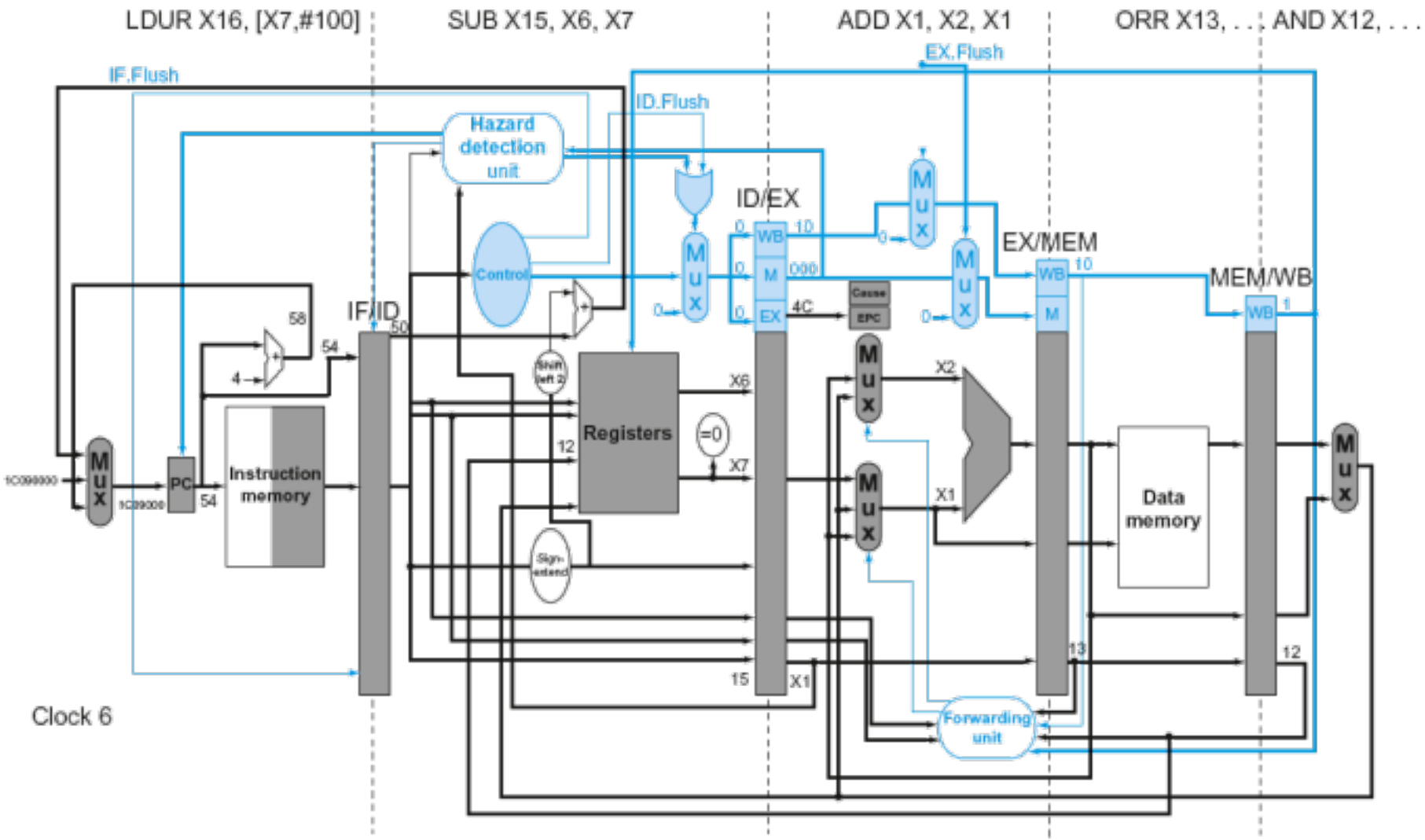
- Exception on ADD in

```
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 LDUR X16, [X7,#100]
…
```

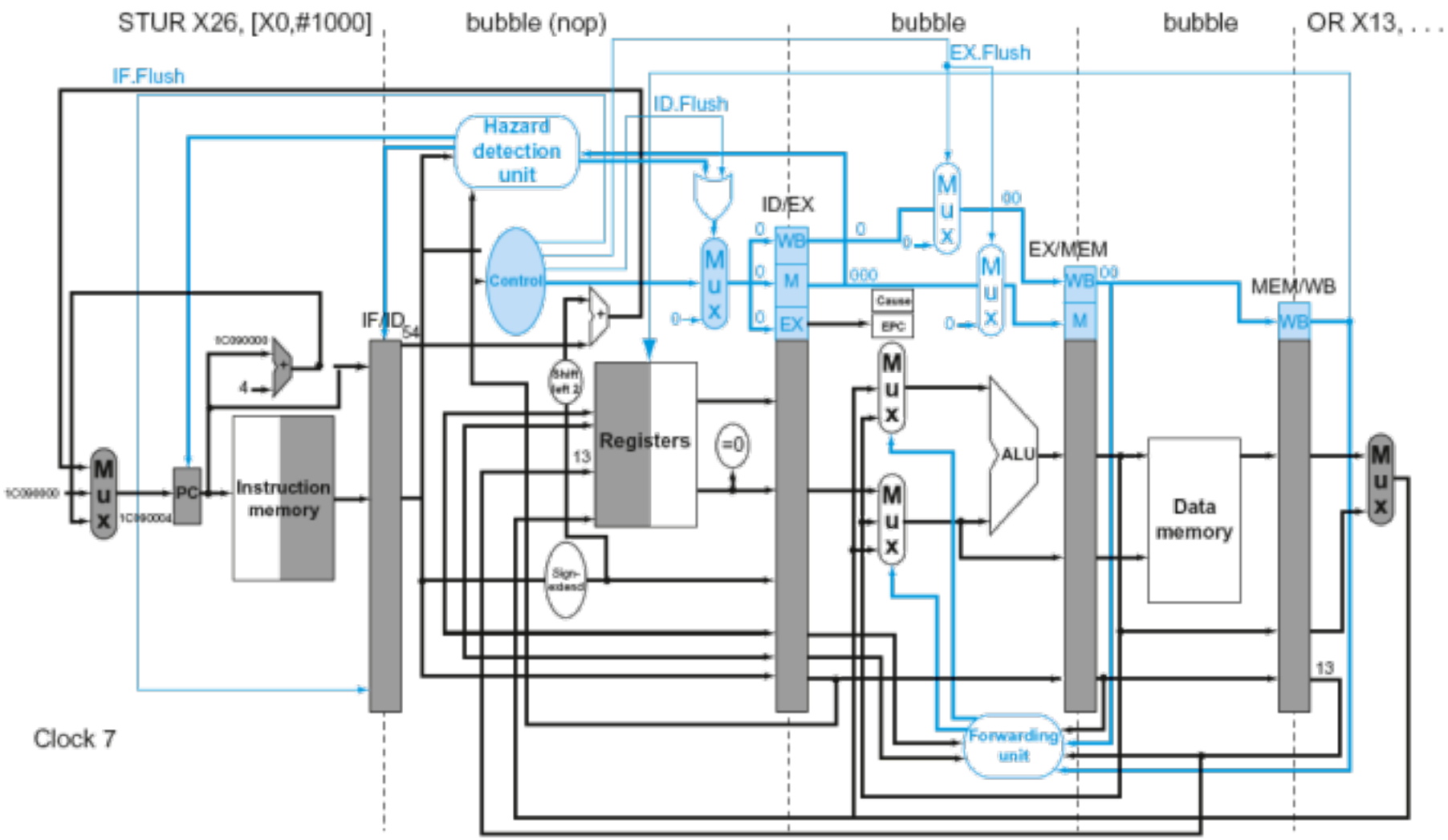
- Handler

```
80000180     STUR X26, [X0,#1000]
80000184   STUR X27, [X0,#1008]
…
```

Exception Example



Exception Example (cont.)



Multiple Exceptions

- Pipelining overlaps multiple instructions.
  - Could have multiple exceptions at once
- Simple approach: Deal with exception from earliest instruction.
  - Flush subsequent instructions.
  - “Precise” exceptions.
- In complex pipelines:
  - Multiple instructions issued per cycle.
  - Out-of-order completion.
  - Maintaining precise exceptions is difficult!

Imprecise Exceptions

- Just stop pipeline and save state.
  - Including exception cause(s)
- Let the handler work out.
  - Which instruction(s) had exceptions
  - Which to complete or flush
    - May require “manual” completion
- Simplifies hardware, but more complex handler software.
- Not feasible for complex multiple-issue out-of-order pipelines.

ENGINEERING@SYRACUSE

Parallelism via Instructions

Instruction-Level Parallelism (ILP)

- Pipelining: executing 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.
    - CPI < 1, so use instructions per cycle (IPC).
    - E.g., 4GHz four-way multiple-issue.
      - 16 BIPS, peak CPI = 0.25, peak IPC = 4
    - But dependencies reduce this in practice.

Multiple Issue

- Static multiple issue
  - Compiler groups instructions to be issued together.
  - Packages them into “issue slots.”
  - Compiler detects and avoids hazards.
- Dynamic multiple issue
  - 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.

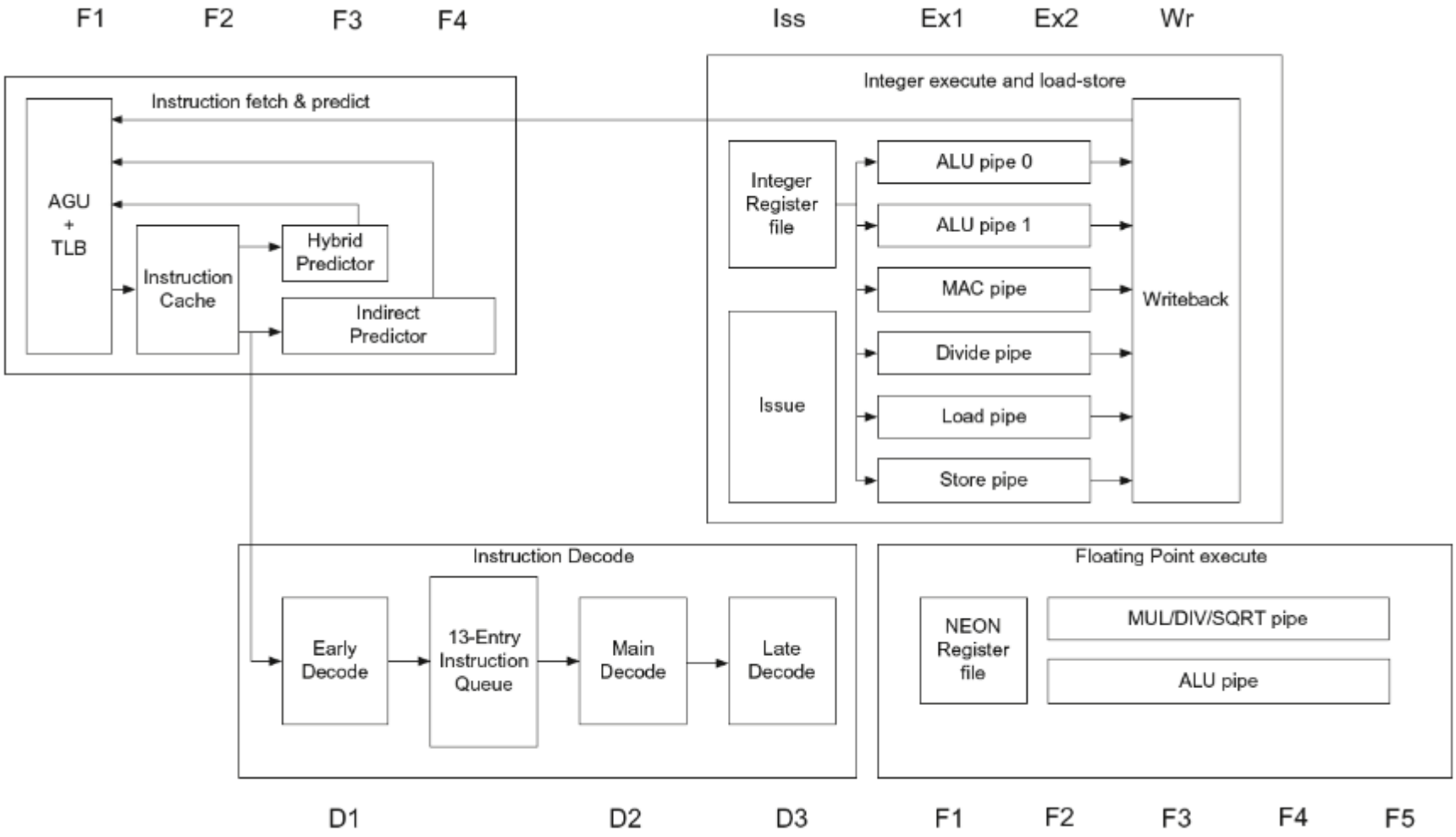
ENGINEERING@SYRACUSE

ARM Cortex A8 and Intel i7 Pipelines

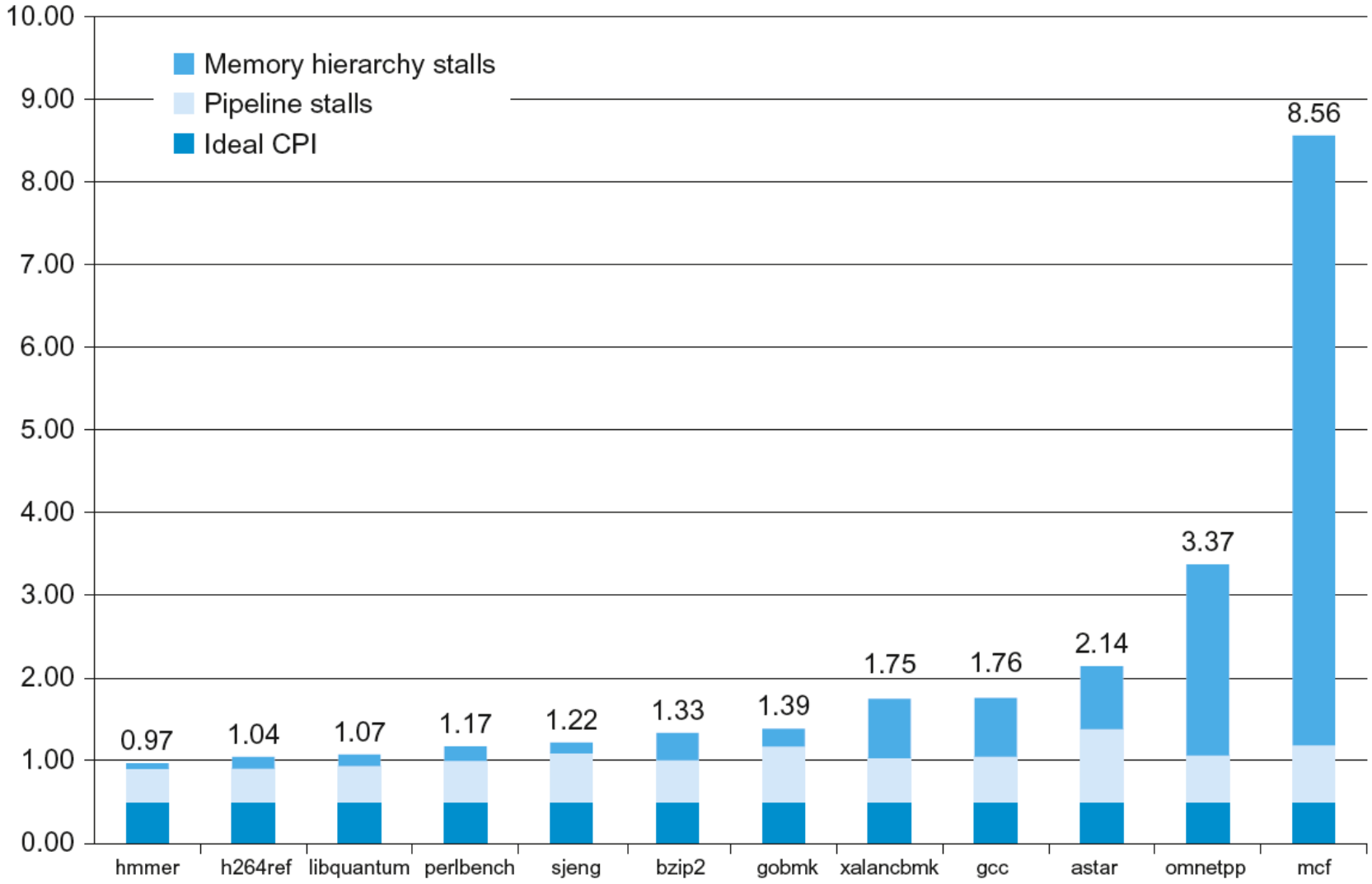
Cortex A53 and Intel i7

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	Two-level
First level caches/core	16-64 KiB I, 16-64 KiB D	32 KiB I, 32 KiB D
Second level caches/core	128-2048 KiB	256 KiB (per core)
Third level caches (shared)	(Platform dependent)	2-8 MB

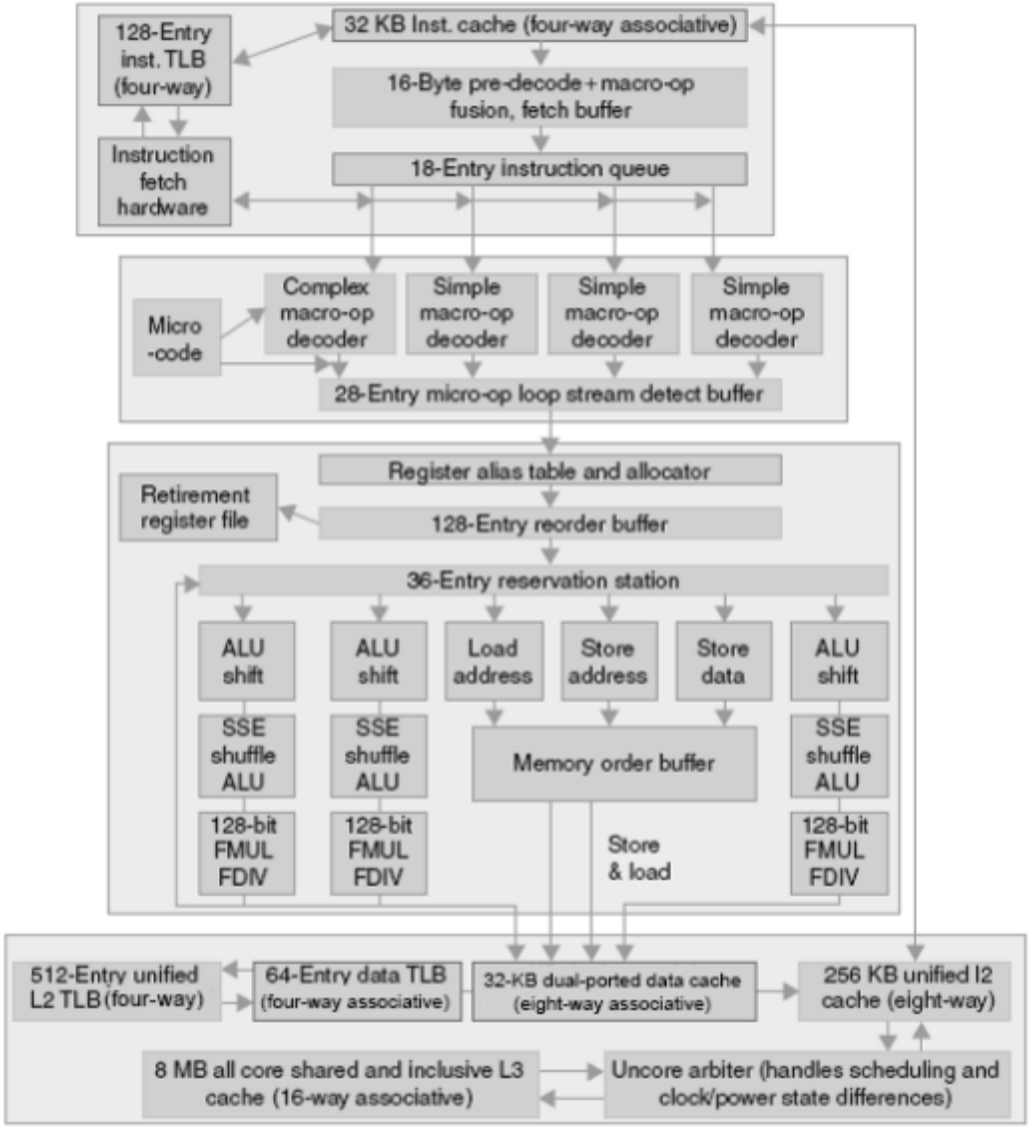
ARM Cortex-A53 Pipeline



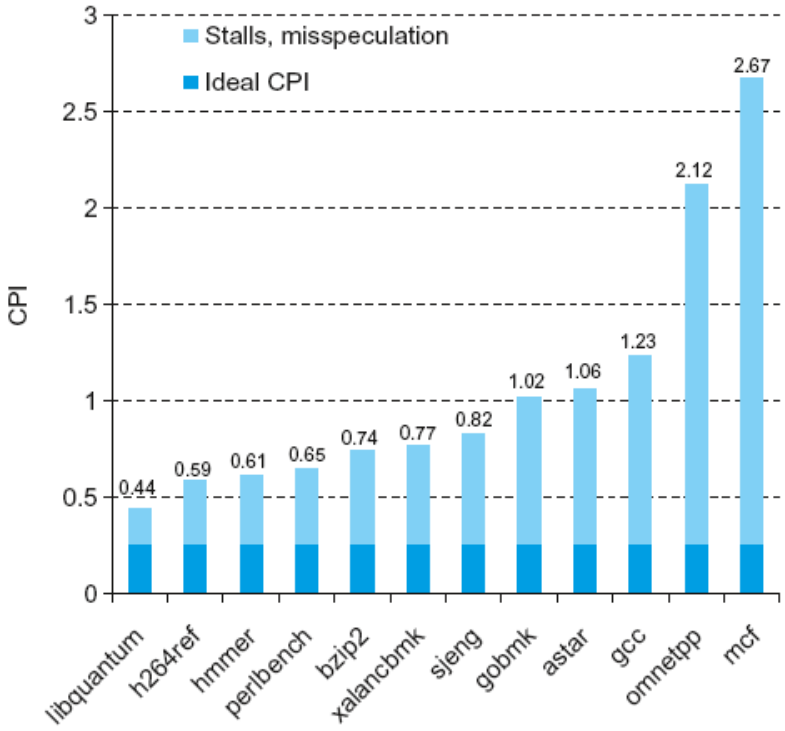
ARM Cortex-A53 Performance

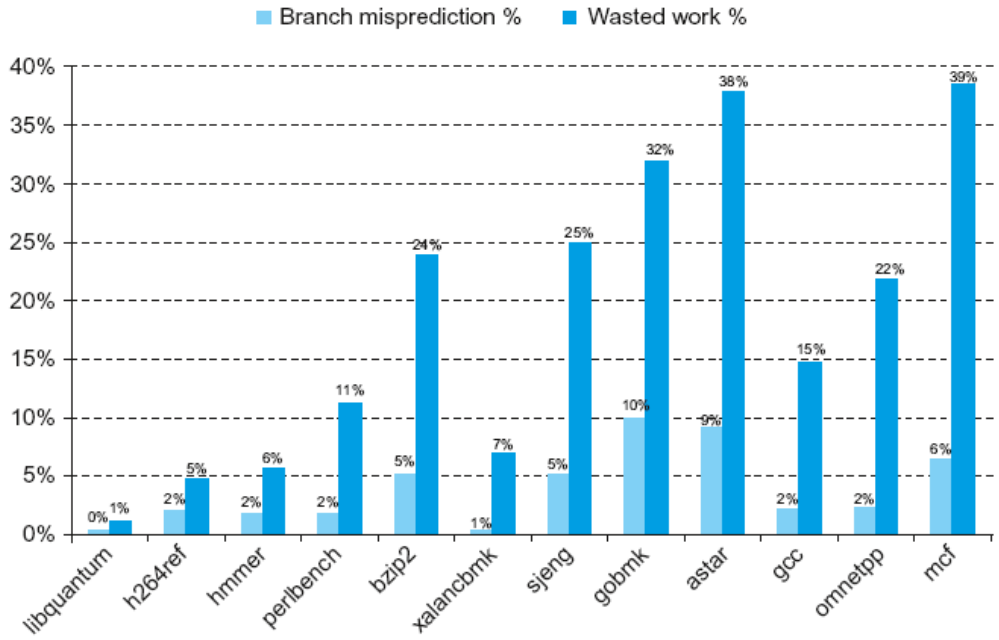


Core i7 Pipeline



Core i7 Performance





Matrix Multiply

- Unrolled C code

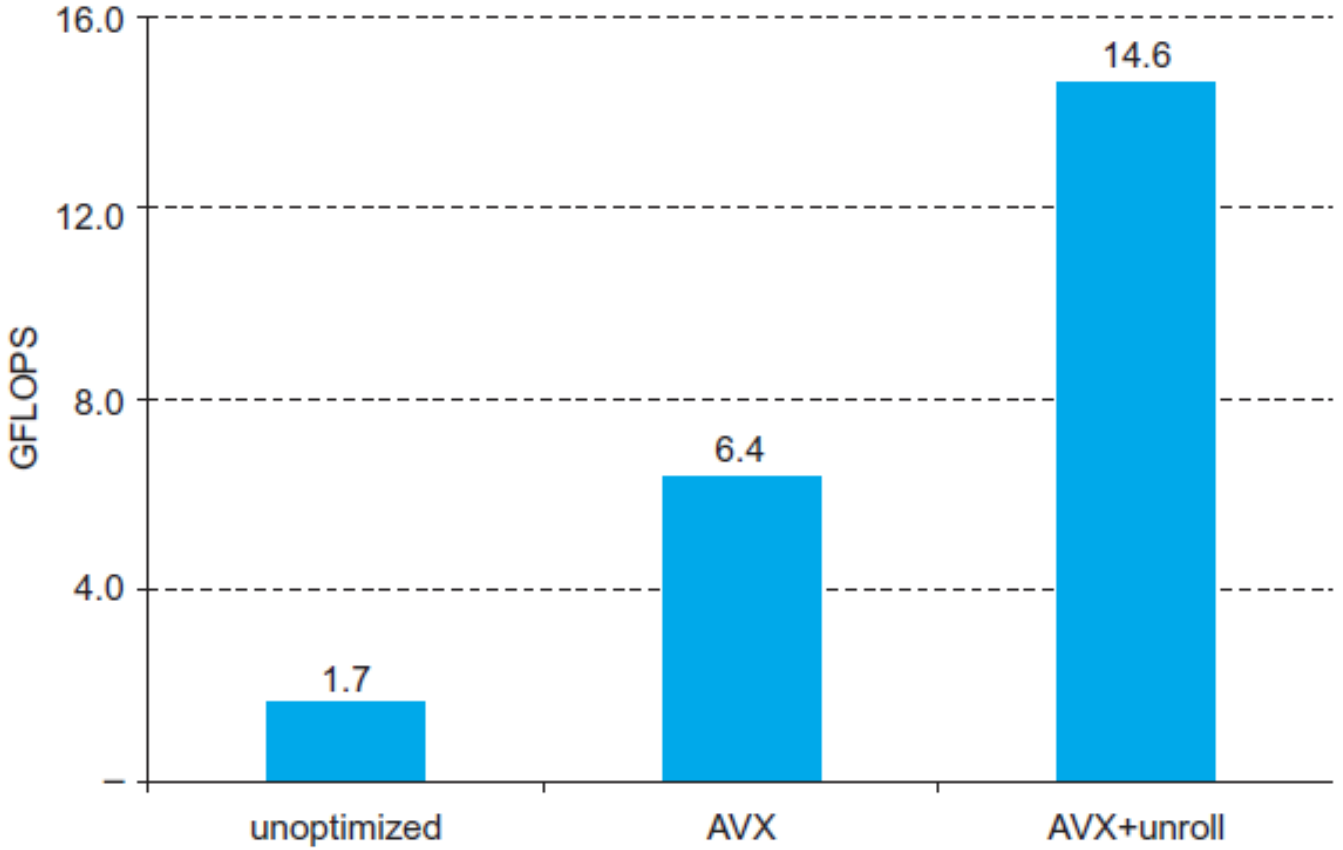
```
1 #include <x86intrin.h>
2 #define UNROLL (4)
3
4 void dgemm (int n, double* A, double* B, double* C)
5 {
6  for ( int i = 0; i < n; i+=UNROLL*4 )
7   for ( int j = 0; j < n; j++ ) {
8    __m256d c[4];
9    for ( int x = 0; x < UNROLL; x++ )
10    c[x] = _mm256_load_pd(C+i+x*4+j*n);
11
12   for( int k = 0; k < n; k++ )
13   {
14    __m256d b = _mm256_broadcast_sd(B+k+j*n);
15    for (int x = 0; x < UNROLL; x++)
16    c[x] = _mm256_add_pd(c[x],
17                         _mm256_mul_pd(_mm256_load_pd(A+n*k+x*4+i), b));
18   }
19
20    for ( int x = 0; x < UNROLL; x++ )
21     _mm256_store_pd(C+i+x*4+j*n, c[x]);
22  }
23 }
```

Matrix Multiply

- Assembly code (with advanced vector extensions):

```
1 vmovapd (%r11),%ymm4                  # Load 4 elements of C into %ymm4
2 mov %rbx,%rax                         # register %rax = %rbx
3 xor %ecx,%ecx                         # register %ecx = 0
4 vmovapd 0x20(%r11),%ymm3              # Load 4 elements of C into %ymm3
5 vmovapd 0x40(%r11),%ymm2              # Load 4 elements of C into %ymm2
6 vmovapd 0x60(%r11),%ymm1              # Load 4 elements of C into %ymm1
7 vbroadcastsd (%rcx,%r9,1),%ymm0       # Make 4 copies of B element
8 add $0x8,%rcx # register %rcx = %rcx + 8
9 vmulpd (%rax),%ymm0,%ymm5             # Parallel mul %ymm1,4 A elements
10 vaddpd %ymm5,%ymm4,%ymm4             # Parallel add %ymm5, %ymm4
11 vmulpd 0x20(%rax),%ymm0,%ymm5        # Parallel mul %ymm1,4 A elements
12 vaddpd %ymm5,%ymm3,%ymm3             # Parallel add %ymm5, %ymm3
13 vmulpd 0x40(%rax),%ymm0,%ymm5        # Parallel mul %ymm1,4 A elements
14 vmulpd 0x60(%rax),%ymm0,%ymm0        # Parallel mul %ymm1,4 A elements
15 add %r8,%rax                         # register %rax = %rax + %r8
16 cmp %r10,%rcx                        # compare %r8 to %rax
17 vaddpd %ymm5,%ymm2,%ymm2             # Parallel add %ymm5, %ymm2
18 vaddpd %ymm0,%ymm1,%ymm1             # Parallel add %ymm0, %ymm1
19 jne 68 <dgemm+0x68>                  # jump if not %r8 != %rax
20 add $0x1,%esi                        # register % esi = % esi + 1
21 vmovapd %ymm4,(%r11)                 # Store %ymm4 into 4 C elements
22 vmovapd %ymm3,0x20(%r11)             # Store %ymm3 into 4 C elements
23 vmovapd %ymm2,0x40(%r11)             # Store %ymm2 into 4 C elements
24 vmovapd %ymm1,0x60(%r11)             # Store %ymm1 into 4 C elements
```

Performance Impact

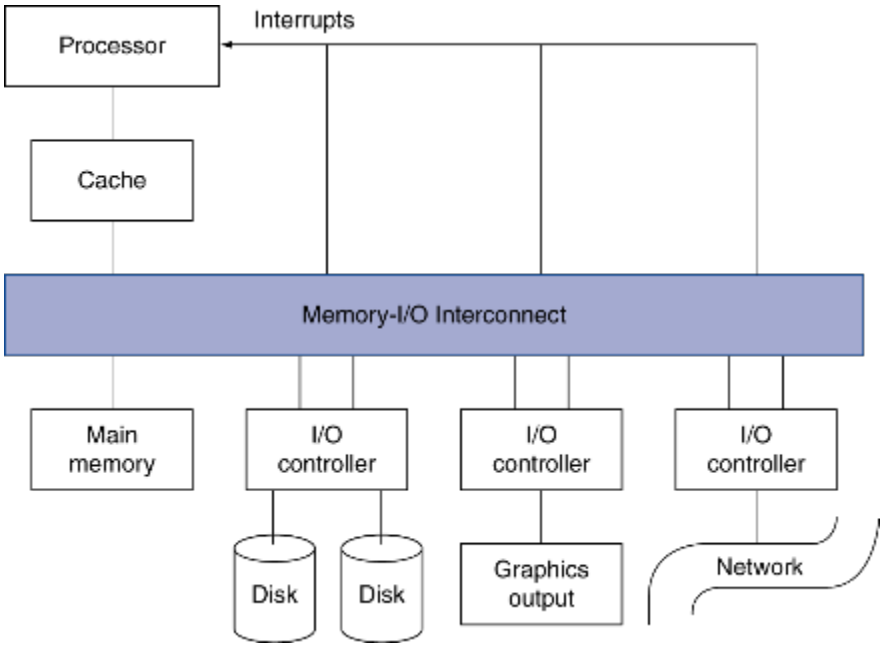


ENGINEERING@SYRACUSE

I/O System Characteristics

Introduction to I/O

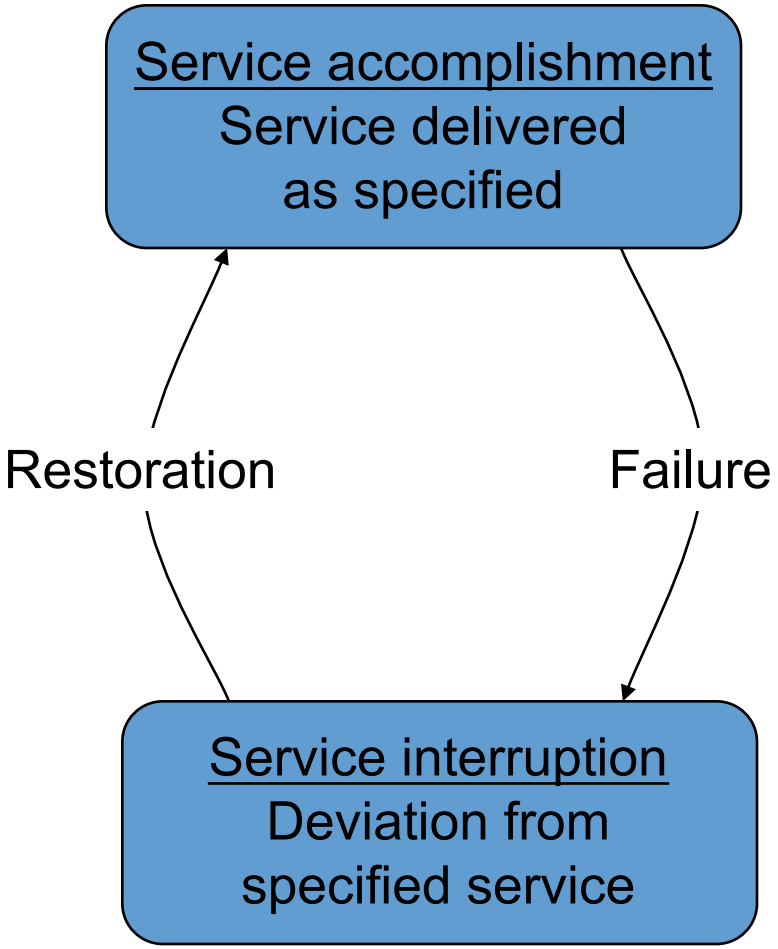
- I/O devices can be characterized by
  - Behavior: input, output, storage
  - Partner: human or machine
  - Data rate: bytes/second, transfers/second
- I/O bus connections



I/O System Characteristics

- Dependability is important.
  - Particularly for storage devices
- Performance measures:
  - Latency (response time)
  - Throughput (bandwidth)
  - Desktops and embedded systems
    - Mainly interested in response time and diversity of devices
  - Servers
    - Mainly interested in throughput and expandability of devices

Dependability



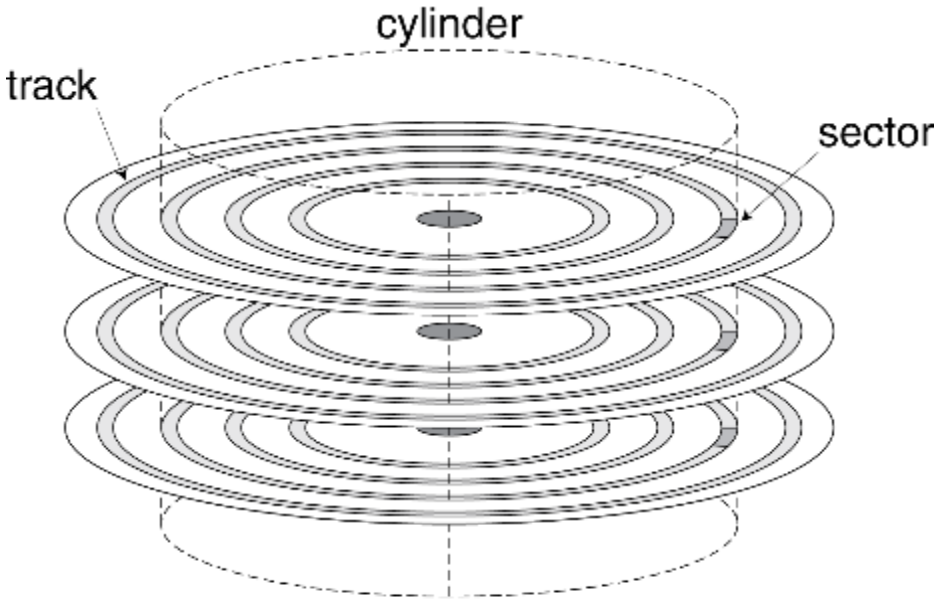
- Fault: failure of a component
  - May or may not lead to system failure

Dependability Measures

- Reliability: mean time to failure (MTTF)
- Service interruption: mean time to repair (MTTR)
- Mean time between failures
  - MTBF = MTTF + MTTR
- Availability = MTTF / (MTTF + MTTR)
- Improving Availability
  - Increase MTTF: fault avoidance, fault tolerance, fault forecasting
  - Reduce MTTR: improved tools and processes for diagnosis and repair

Disk Storage

- Nonvolatile, rotating magnetic storage



Disk Sectors and Access

- Each sector records
  - Sector ID
  - Data (512 bytes, 4096 bytes proposed)
  - Error correcting code (ECC)
    - Used to hide defects and recording errors
  - Synchronization fields and gaps
- Access to a sector involves
  1. Queuing delay if other accesses are pending
  2. Seek: move the heads
  3. Rotational latency
  4. Data transfer
  5. Controller overhead

Disk Access Example

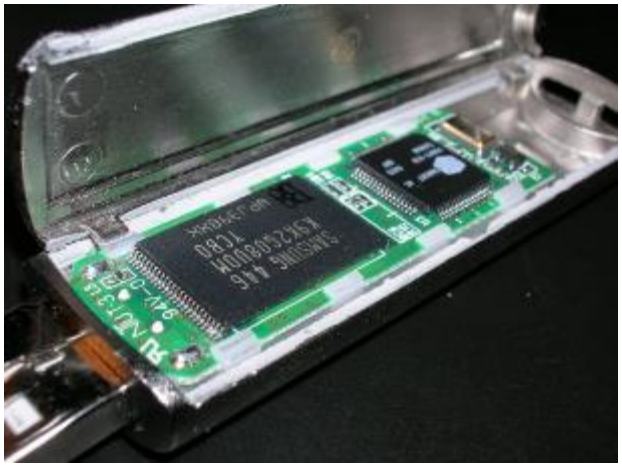
- Given
  - 512 B sector, 15,000 rpm, 4 ms average seek time, 100 MB/s transfer rate, 0.2 ms controller overhead, idle disk
- Average read time
  - 4 ms seek time
    + ½ /(15,000/60) = 2 ms rotational latency
    + 512/100 MB/s = 0.005 ms transfer time
    + 0.2 ms controller delay
    = 6. 2ms
- If actual average seek time is 1 ms
  - Average read time = 3.2 ms

Disk Performance Issues

- Manufacturers quote average seek time.
  - Based on all possible seeks.
  - Locality and OS scheduling lead to smaller actual average seek times.
- Smart disk controller allocate physical sectors on disk.
  - Present logical sector interface to host.
  - SCSI, ATA, SATA.
- Disk drives include caches.
  - Prefetch sectors in anticipation of access.
  - Avoid seek and rotational delay.

Flash Storage

- Nonvolatile semiconductor storage
  - 100×–1000× faster than disk
  - Smaller, lower power, more robust
  - But more $/GB (between disk and DRAM)



Flash Types

- NOR flash: bit cell like a NOR gate.
  - Random read/write access
  - Used for instruction memory in embedded systems
- NAND flash: bit cell like a NAND gate.
  - Denser (bits/area), but block-at-a-time access
  - Cheaper per GB
  - Used for USB keys, media storage, …
- Flash bits wears out after 1000s of accesses.
  - Not suitable for direct RAM or disk replacement.
  - Wear leveling: Remap data to less-used blocks.

ENGINEERING@SYRACUSE

Interconnecting Components

Interconnecting Components

- Need interconnections between
  - CPU, memory, I/O controllers
- Bus: shared communication channel
  - Parallel set of wires for data and synchronization of data transfer
  - Can become a bottleneck
- Performance limited by physical factors
  - Wire length, number of connections
- More recent alternative: high-speed serial connections with switches
  - Like networks

Bus Types

- Processor-memory buses
  - Short, high speed.
  - Design is matched to memory organization.
- I/O buses
  - Longer, allowing multiple connections.
  - Specified by standards for interoperability.
  - Connect to processor memory bus through a bridge.

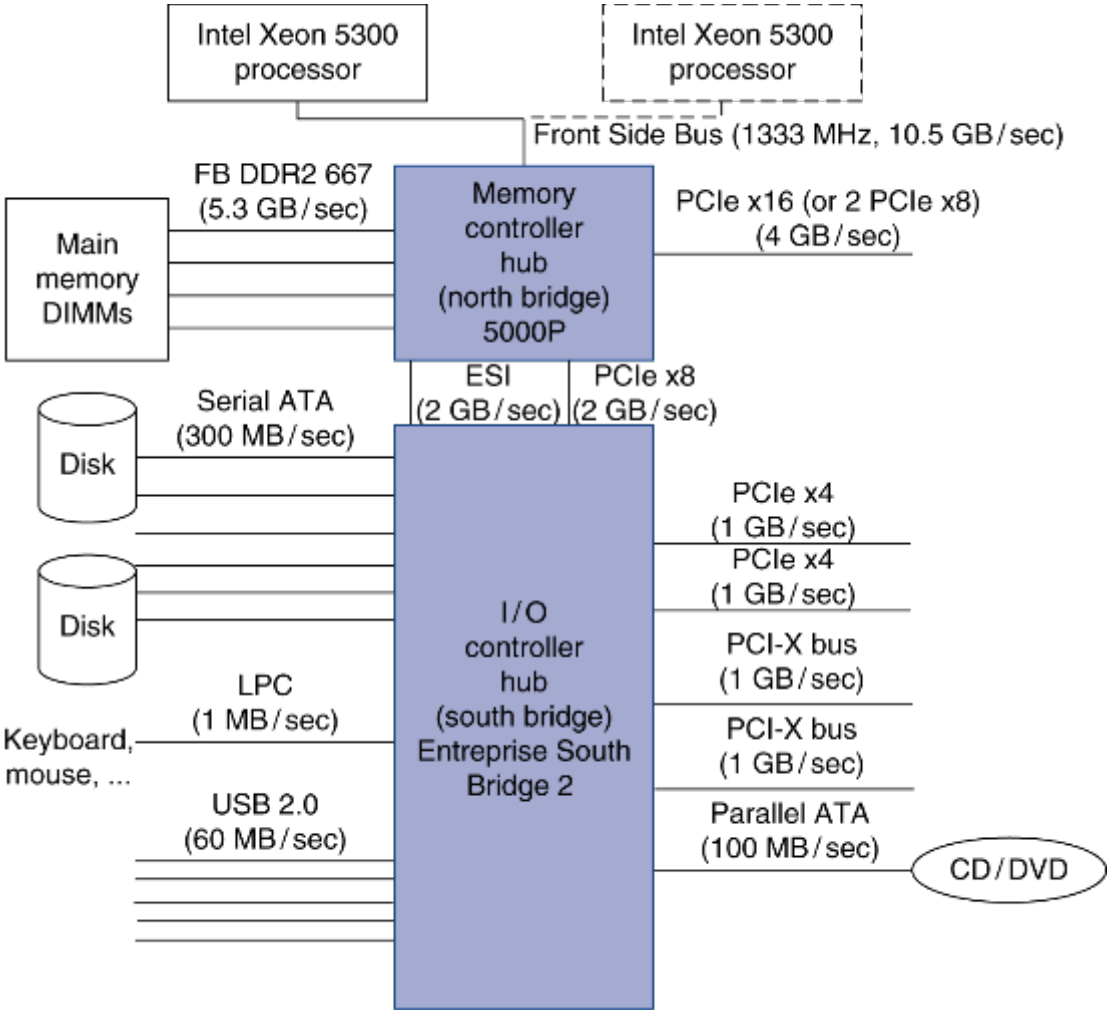
Bus Signals and Synchronization

- Data lines
  - Carry address and data
  - Multiplexed or separate
- Control lines
  - Indicate data type, synchronize transactions
- Synchronous
  - Uses a bus clock
- Asynchronous
  - Uses request/acknowledge control lines for handshaking

I/O Bus Examples

	Firewire	USB 2.0	PCI Express	Serial ATA	Serial Attached SCSI
Intended use	External	External	Internal	Internal	External
Devices per channel	63	127	1	1	4
Data width	4	2	2/lane	4	4
Peak bandwidth	50 MB/s or 100 MB/s	0.2 MB/s, 1.5 MB/s, or 60 MB/s	250 MB/s/lane 1×, 2×, 4×, 8×, 16×, 32×	300 MB/s	300 MB/s
Hot pluggable	Yes	Yes	Depends	Yes	Yes
Max length	4.5 m	5 m	0.5 m	1 m	8 m
Standard	IEEE 1394	USB Implementers Forum	PCI-SIG	SATA-IO	INCITS TC T10

Typical x86 PC I/O System



I/O Management

- I/O is mediated by the OS.
  - Multiple programs share I/O resources.
    - Need protection and scheduling
  - I/O causes asynchronous interrupts.
    - Same mechanism as exceptions
  - I/O programming is fiddly.
    - OS provides abstractions to programs.

I/O Commands

- I/O devices are managed by I/O controller hardware.
  - Transfers data to/from device
  - Synchronizes operations with software
- Command registers.
  - Cause device to do something
- Status registers.
  - Indicate what the device is doing and occurrence of errors
- Data registers.
  - Write: transfer data to a device
  - Read: transfer data from a device

I/O Register Mapping

- Memory mapped I/O
  - Registers are addressed in same space as memory.
  - Address decoder distinguishes between them.
  - OS uses address translation mechanism to make them only accessible to kernel.
- I/O instructions
  - Separate instructions to access I/O registers
  - Can only be executed in kernel mode
  - Example: x86

Polling

- Periodically check I/O status register.
  - If device ready, do operation.
  - If error, take action.
- Common in small or low-performance real-time embedded systems.
  - Predictable timing
  - Low hardware cost
- In other systems, wastes CPU time.

Interrupts

- When a device is ready or error occurs.
  - Controller interrupts CPU.
- Interrupt is like an exception.
  - But not synchronized to instruction execution.
  - Can invoke handler between instructions.
  - Cause information often identifies the interrupting device.
- Priority interrupts.
  - Devices needing more urgent attention get higher priority.
  - Can interrupt handler for a lower-priority interrupt.

I/O Data Transfer

- Polling and interrupt-driven I/O
  - CPU transfers data between memory and I/O data registers.
  - Time consuming for high-speed devices.
- Direct memory access (DMA)
  - OS provides starting address in memory.
  - I/O controller transfers to/from memory autonomously.
  - Controller interrupts on completion or error.

DMA/Cache Interaction

- If DMA writes to a memory block that is cached:
  - Cached copy becomes stale
- If write-back cache has dirty block, and DMA reads memory block:
  - Reads stale data
- Need to ensure cache coherence.
  - Flush blocks from cache if they will be used for DMA.
  - Or use noncacheable memory locations for I/O.

DMA/VM Interaction

- OS uses virtual addresses for memory.
  - DMA blocks may not be contiguous in physical memory.
- Should DMA use virtual addresses?
  - Would require controller to do translation
- If DMA uses physical addresses:
  - May need to break transfers into page-sized chunks
  - Or chain multiple transfers
  - Or allocate contiguous physical pages for DMA

Measuring I/O Performance

- I/O performance depends on:
  - Hardware: CPU, memory, controllers, buses
  - Software: operating system, database management system, application
  - Workload: request rates and patterns
- I/O system design can trade off between response time and throughput.
  - Measurements of throughput often done with constrained response-time.

Transaction Processing Benchmarks

- Transactions
  - Small data accesses to a DBMS.
  - Interested in I/O rate, not data rate.
- Measure throughput
  - Subject to response-time limits and failure handling
  - ACID (atomicity, consistency, isolation, durability)
  - Overall cost per transaction
- Transaction Processing Council (TPC) benchmarks (www.tcp.org)
  - TPC-APP: B2B application server and web services
  - TCP-C: online order entry environment
  - TCP-E: online transaction processing for brokerage firm
  - TPC-H: decision support—business-oriented ad-hoc queries

File System and Web Benchmarks

- SPEC system file system (SFS)
  - Synthetic workload for NFS server, based on monitoring real systems
  - Results
    - Throughput (operations/sec)
    - Response time (average ms/operation)
- SPEC web server benchmark
  - Measures simultaneous user sessions, subject to required throughput/session
  - Three workloads: banking, ecommerce, and support

ENGINEERING@SYRACUSE

Parallelism and I/O

I/O vs. CPU Performance

- Amdahl's law
  - Don't neglect I/O performance as parallelism increases computing performance.
- Example
  - Benchmark takes 90 s CPU time, 10 s I/O time.
  - Double the number of CPUs/two years.
    - I/O unchanged

Year	CPU time	I/O time	Elapsed time	% I/O time
now	90s	10s	100s	10%
+2	45s	10s	55s	18%
+4	23s	10s	33s	31%
+6	11s	10s	21s	47%

RAID

- Redundant array of inexpensive (independent) disks
  - Use multiple smaller disks (c.f., one large disk).
  - Parallelism improves performance.
  - Plus extra disk(s) for redundant data storage.
- Provides fault-tolerant storage system
  - Especially if failed disks can be “hot swapped”
- RAID 0
  - No redundancy (“AID”?).
    - Just stripe data over multiple disks.
  - But it does improve performance.

RAID 1 and 2

- RAID 1: Mirroring
  - N + N disks, replicate data
    - Write data to both data disk and mirror disk.
    - On disk failure, read from mirror.
- RAID 2: Error-correcting code (ECC)
  - N + E disks (e.g., 10 + 4).
  - Split data at bit level across N disks.
  - Generate E-bit ECC.
  - Too complex, not used in practice.

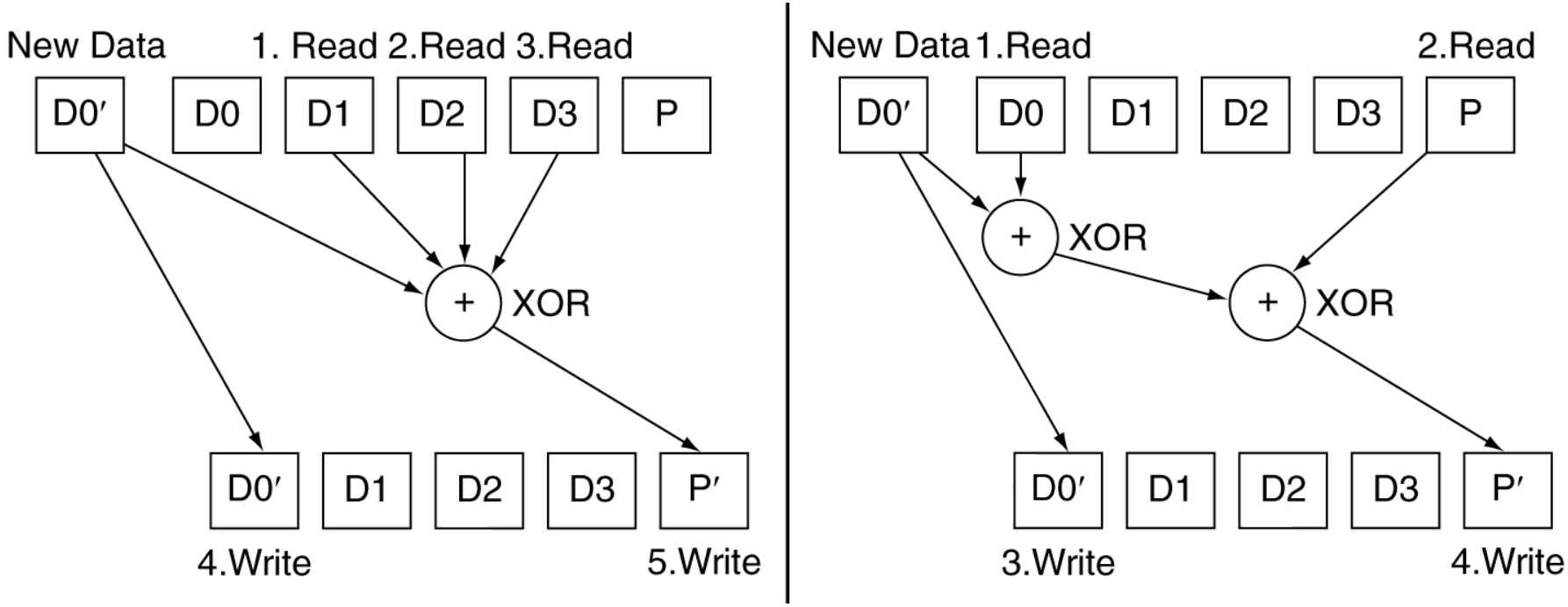
RAID 3: Bit-Interleaved Parity

- N + 1 disks
  - Data striped across N disks at byte level.
  - Redundant disk stores parity.
  - Read access:
    - Read all disks.
  - Write access:
    - Generate new parity and update all disks.
  - On failure:
    - Use parity to reconstruct missing data.
- Not widely used

RAID 4: Block-Interleaved Parity

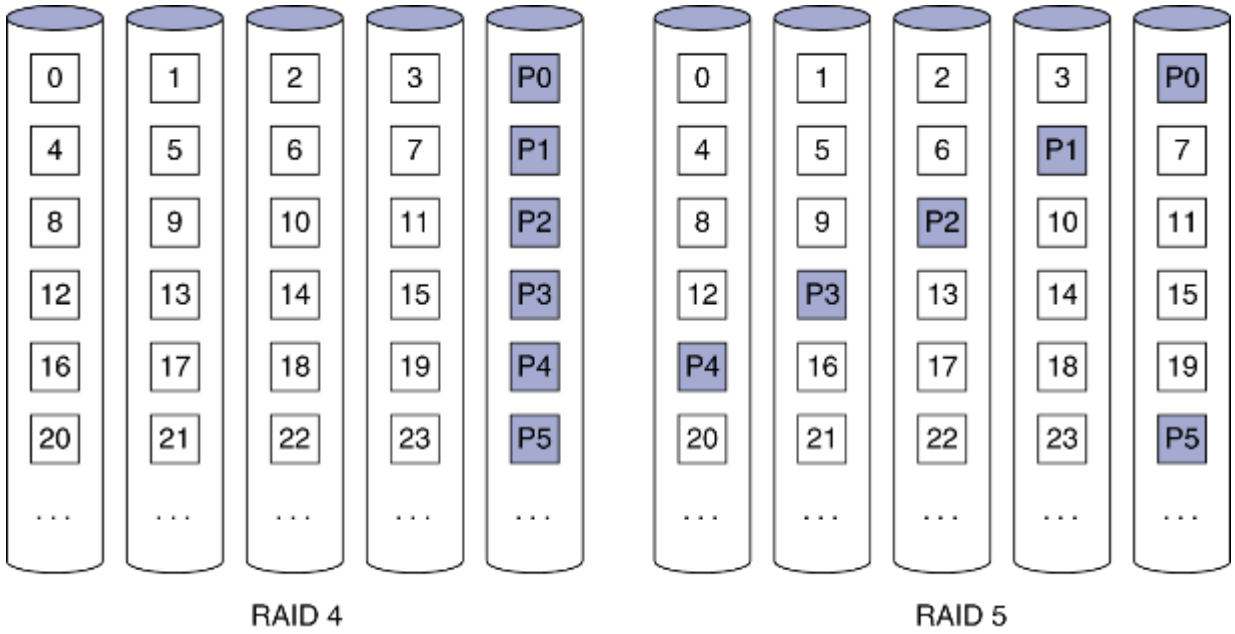
- N + 1 disks
  - Data striped across N disks at block level.
  - Redundant disk stores parity for a group of blocks.
  - Read access:
    - Read only the disk holding the required block.
  - Write access:
    - Just read disk containing modified block, and parity disk.
    - Calculate new parity, update data disk and parity disk.
  - On failure:
    - Use parity to reconstruct missing data.
- Not widely used

RAID 3 vs. RAID 4



RAID 5: Distributed Parity

- N + 1 disks
  - Like RAID 4, but parity blocks distributed across disks.
    - Avoids parity disk being a bottleneck.
- Widely used



RAID 6: P + Q Redundancy

- N + 2 disks
  - Like RAID 5, but lots of parity.
  - Greater fault tolerance through more redundancy.
- Multiple RAID
  - More advanced systems give similar fault tolerance with better performance.

RAID Summary

- RAID can improve performance and availability.
  - High availability requires hot swapping.
- Assumes independent disk failures.
  - Too bad if the building burns down!

ENGINEERING@SYRACUSE

I/O System Design

I/O System Design

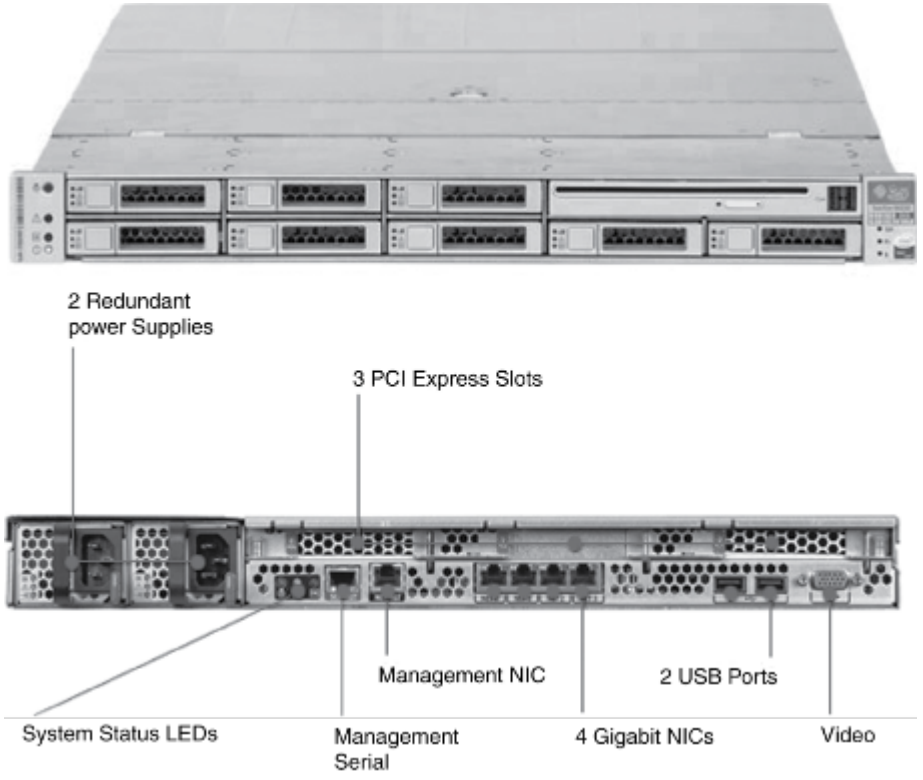
- Satisfying latency requirements.
  - For time-critical operations
  - If system is unloaded
    - Add up latency of components.
- Maximizing throughput.
  - Find “weakest link” (lowest-bandwidth component).
  - Configure to operate at its maximum bandwidth.
  - Balance remaining components in the system.
- If system is loaded, simple analysis is insufficient.
  - Need to use queuing models or simulation

Server Computers

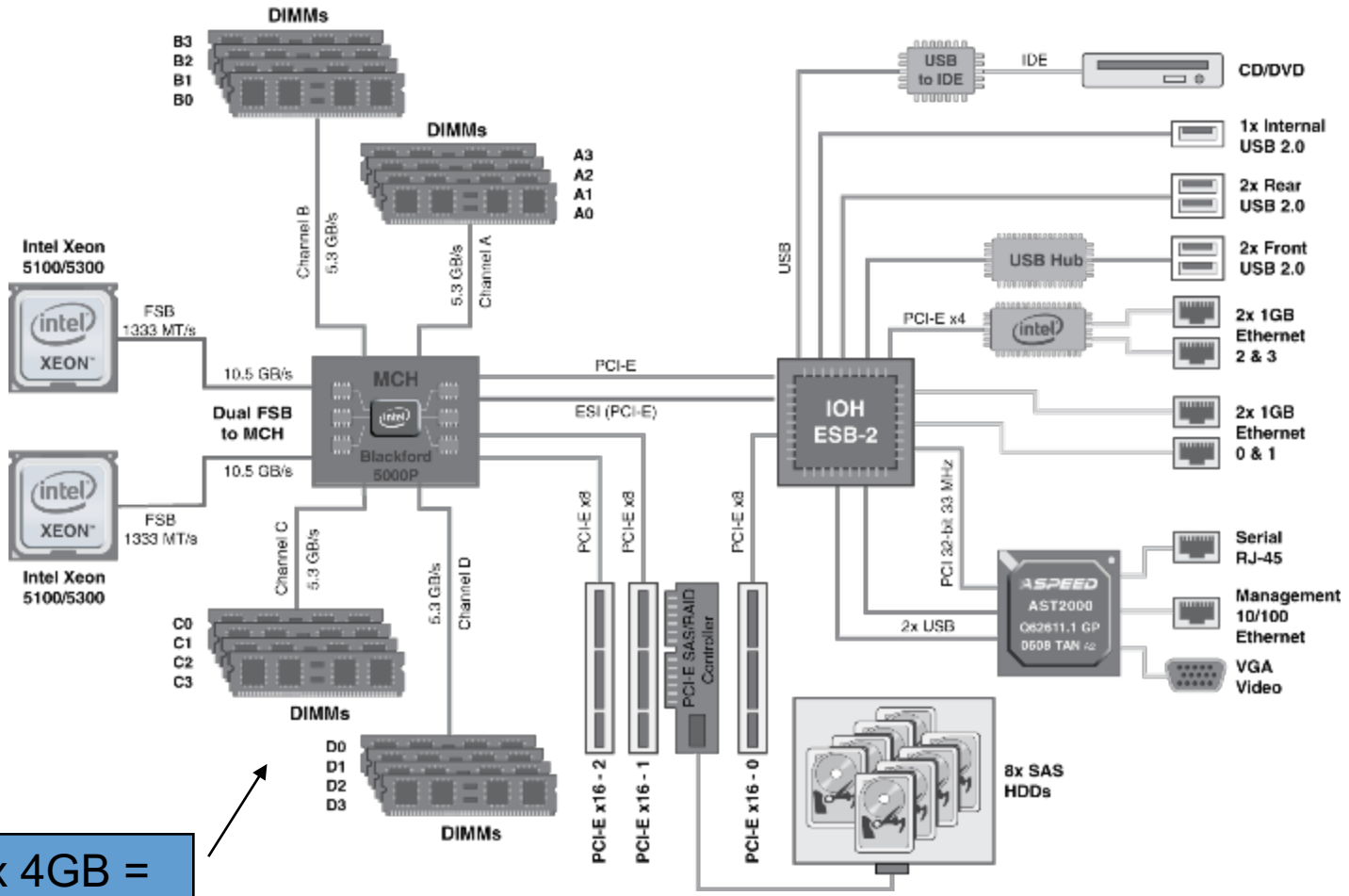
- Applications are increasingly run on servers.
  - Web search, office apps, virtual worlds, …
- Requires large data center servers.
  - Multiple processors, networks connections, massive storage
  - Space and power constraints
- Server equipment built for 19" racks.
  - Multiples of 1.75" (1U) high

Rack-Mounted Servers

Ex: Sun Fire x4150 1U server



Sun Fire x4150 1U server



Four cores each

16 x 4GB = 64GB DRAM

I/O System Design Example

- Given a Sun Fire x4150 system with
  - Workload: 64 KB disk reads
    - Each I/O op requires 200,000 user code instructions and 100,000 OS instructions.
  - Each CPU: 10^9 instructions/sec
  - FSB: 10.6 GB/sec peak
  - DRAM DDR2 667 MHz: 5.336 GB/sec
  - PCI-E 8× bus: 8 × 250 MB/sec = 2 GB/sec
  - Disks: 15,000 rpm, 2.9 ms average seek time, 112 MB/sec transfer rate
- What I/O rate can be sustained?
  - For random reads, and for sequential reads

Design Example (cont.)

- I/O rate for CPUs
  - Per core: $10^9/(100{,}000 + 200{,}000) = 3{,}333$
  - Eight cores: 26,667 ops/sec
- Random reads, I/O rate for disks
  - Assume actual seek time is average/4.
  - Time/op = seek + latency + transfer
    = 2.9 ms/4 + 4 ms/2 + 64 KB/(112 MB/s) = 3.3 ms.
  - 303 ops/sec per disk, 2424 ops/sec for eight disks
- Sequential reads
  - 112 MB/s / 64 KB = 1750 ops/sec per disk
  - 14,000 ops/sec for eight disks

Design Example (cont.)

- PCI-E I/O rate
  - 2 GB/sec/64 KB = 31,250 ops/sec
- DRAM I/O rate
  - 5.336 GB/sec/64 KB = 83,375 ops/sec
- FSB I/O rate
  - Assume we can sustain half the peak rate.
  - 5.3 GB/sec/64 KB = 81,540 ops/sec per FSB.
  - 163,080 ops/sec for two FSBs.
- Weakest link: disks
  - 2424 ops/sec random, 14,000 ops/sec sequential.
  - Other components have ample headroom to accommodate these rates.

ENGINEERING@SYRACUSE

Conclusions

In Conclusion

- Hazards: structural, data, control
- Exception handling
- CPI<1: parallelism via instructions
- Implementations: ARM vs. Intel
- I/O: latency, throughput
- Storage: disk, flash drive, RAID
- Buses: parallel, serial
- Control: interruptions, polling

ENGINEERING@SYRACUSE