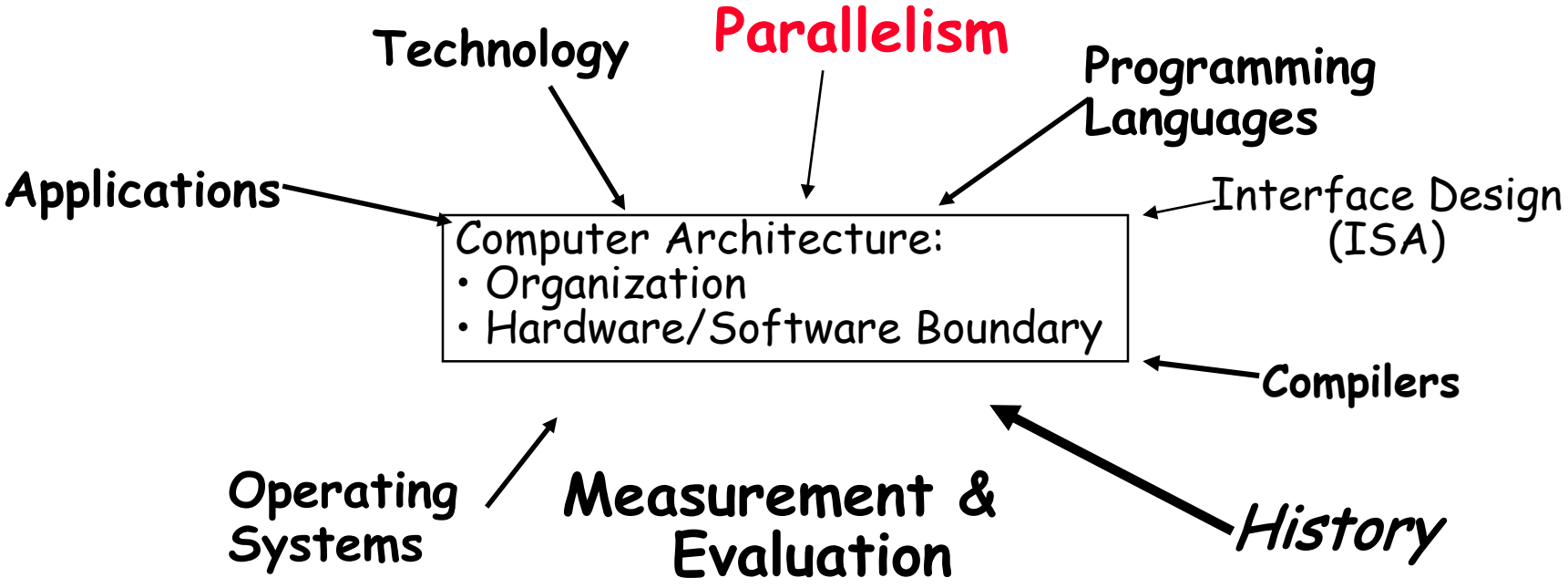
Computer Architecture and System

Chung-Ho Chen
Computer Architecture and System Laboratory
Department of Electrical Engineering
National Cheng-Kung University

Course Focus

Understanding the design techniques, machine structures, technology factors, evaluation methods that will determine the form of computers in the 21st Century



Your Goals

- **Computer architecture**
  - **Power, dependability, multi CPU vs. 1 CPU performance.**
- **Mix of lecture vs. discussion**
  - **Depends on how well reading is done before class**
- **Goal is to learn how to do good system research**
  - **Learn a lot from looking at good work in the past.**
  - **At commit point, you may choose to pursue your own new idea instead.**
  - **Learn the strategies in writing a good paper, paragraph by paragraph, sentence by sentence.**

Research Paper Reading

- **As graduate students, you are now researchers**
- **Most information of importance to you will be in research papers**
- **Ability to rapidly scan and understand research papers is key to your success. Also from the Internet for quick learning.**
- **So:**
  - **(1) You will read a few papers in this course**
  - **(2) Get assignment for new technology presentation.**
  - **Paper and assignment will be announced.**

Grading

- 60% Examinations (3 Exams)
- 20% Class participation, presentation of new technology, reading write-ups.
- 20% Full system simulation project/GPU exercises
  - Learn how to design an ASIC or hardware of your choice
  - Learn how to hook up your hardware into a Linux system which runs on a ARM-based platform.
    - » Learn how to write a device driver and perform hardware-software co-simulation and functional verification
  - Experience a GPGPU simulator and programming.

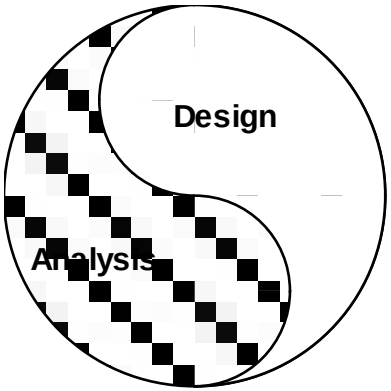
Performance Wall Ahead

- **Power wall**
  - **Power expensive, transistor free**
    (Can put more on chip than can afford to turn it on)
- **ILP wall**
  - **Instruction level parallelism diminishes on more HW for ILP**
- **Memory wall**
  - **Memory, the legacy problem**
- **Previously uniprocessor performance 2X / 1.5 yrs**
- **Now: Power Wall + ILP Wall + Memory Wall**
  - **Uniprocessor performance now 2X / 5(?) yrs**
- **Big change in direction: multiple cores**
  (2X processors per chip / ~ 2 years)

Power Formula

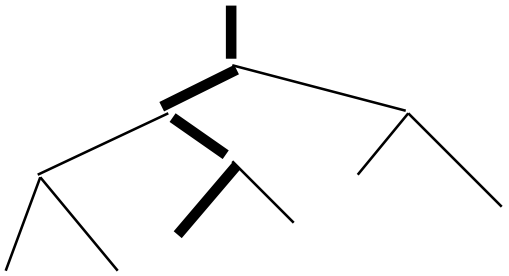
- **Dynamic power = ½ x Capacitive load x voltage^2 x frequency switched (V^2/R)**
- **Dynamic energy = Capacitive load x voltage^2**
  - **Capacitances come from transistors and wires.**
- **Static power = Static current x voltage**
  - **More than 25 percent of the total power consumption resulting from leakage current**

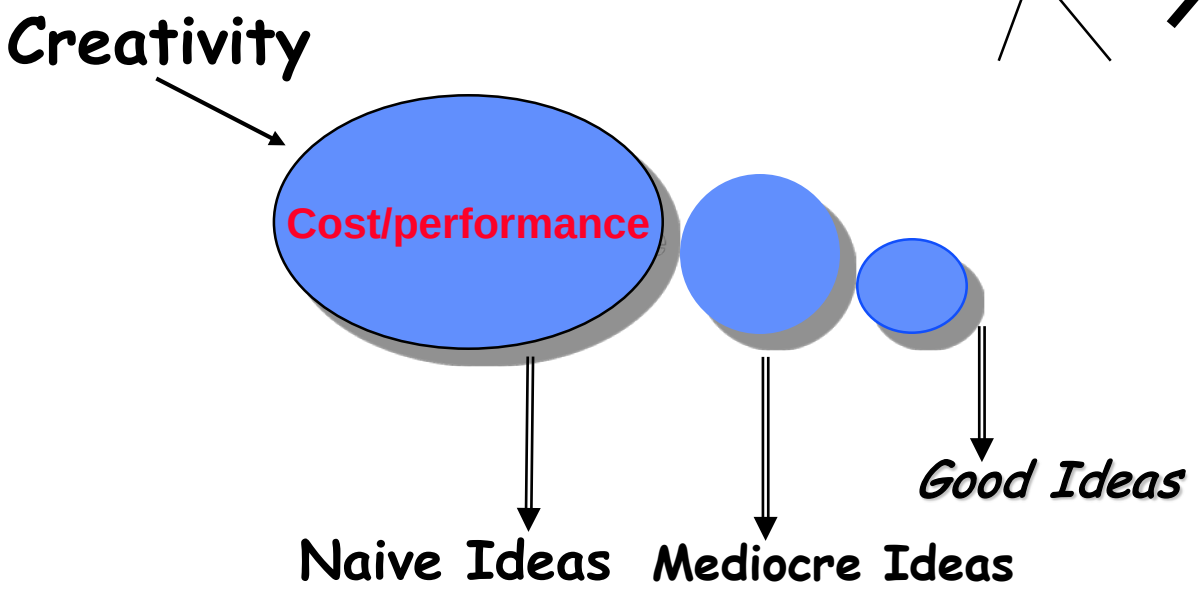
Computer Architecture is Design and Analysis



Architecture is an iterative process:

- **Searching the space of possible designs**
- **At all levels of computer systems**



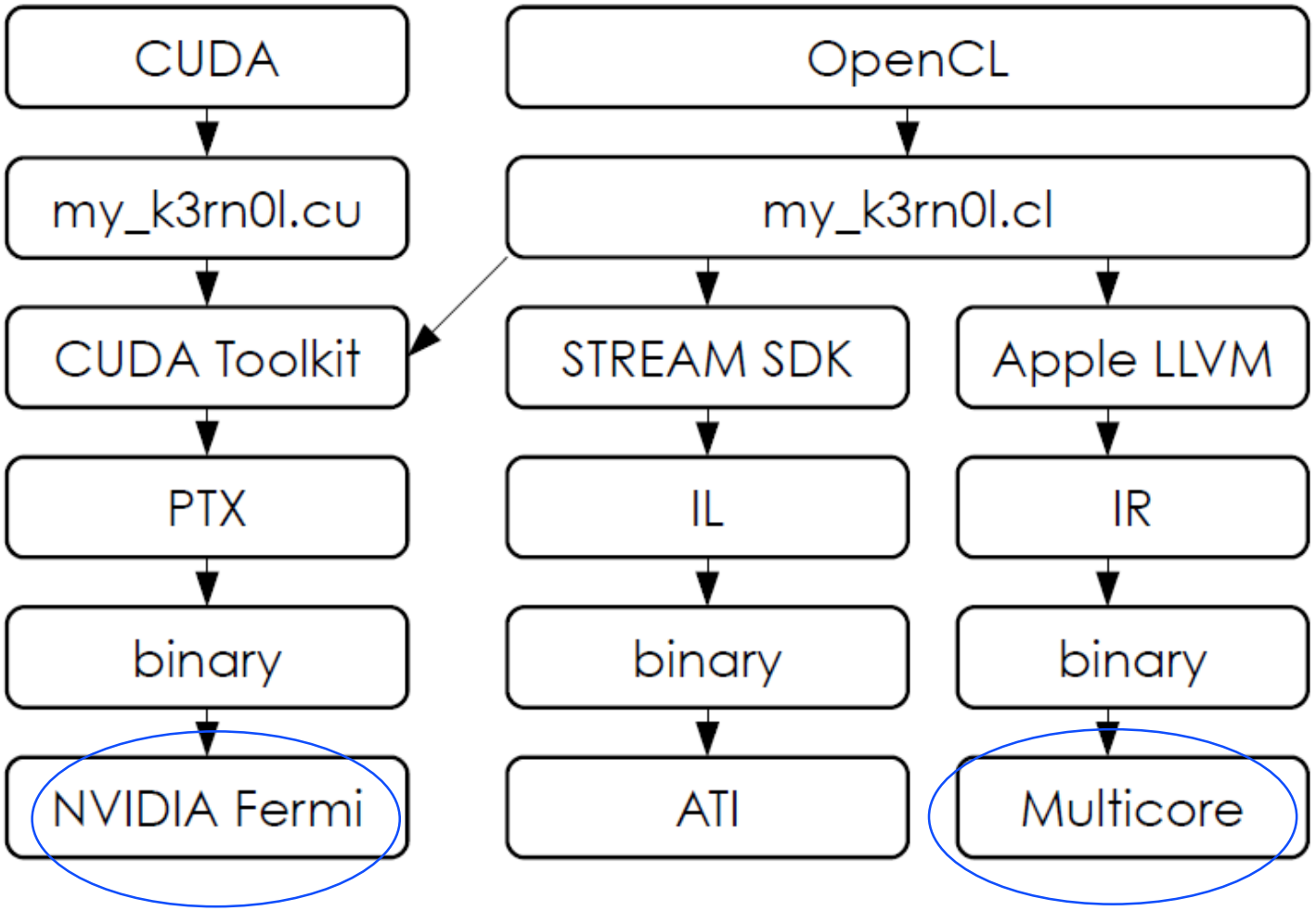


What Computer Architecture Brings to You

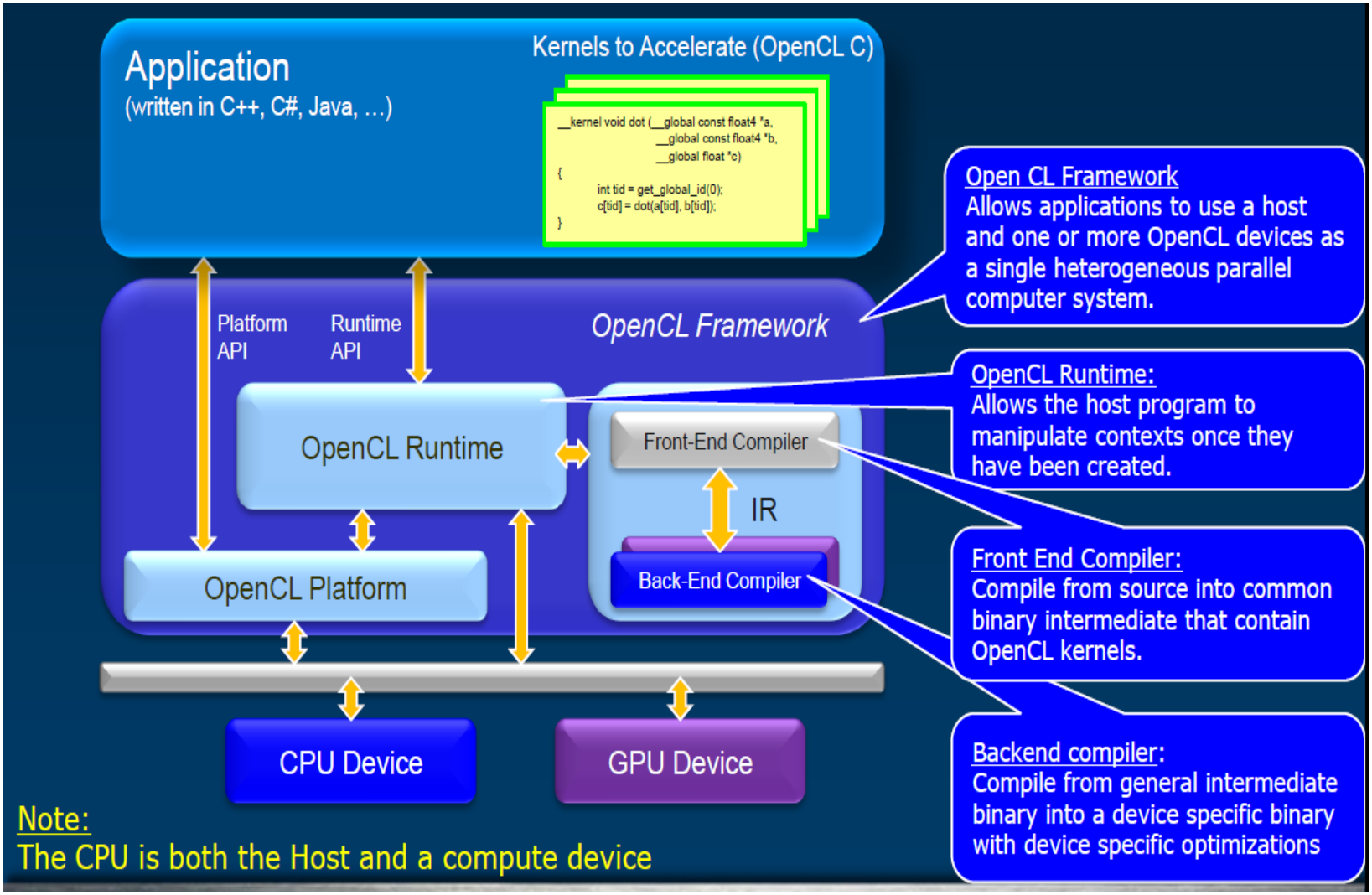
- Other fields often borrow ideas from architecture
- Quantitative Principles of Design
  1. Take Advantage of Parallelism
  2. Principle of Locality
  3. Focus on the Common Case
  4. Amdahl's Law
  5. The Processor Performance Equation
- Careful, quantitative comparisons
  - Define, quantify, and summarize relative performance
  - Define and quantify relative cost
  - Define and quantify dependability
  - Define and quantify power
- Culture of anticipating and exploiting advances in technology
- Culture of well-defined interfaces that are carefully implemented and thoroughly checked

New things

- **From CUDA to OpenCL**



System Architecture of Application Processor?



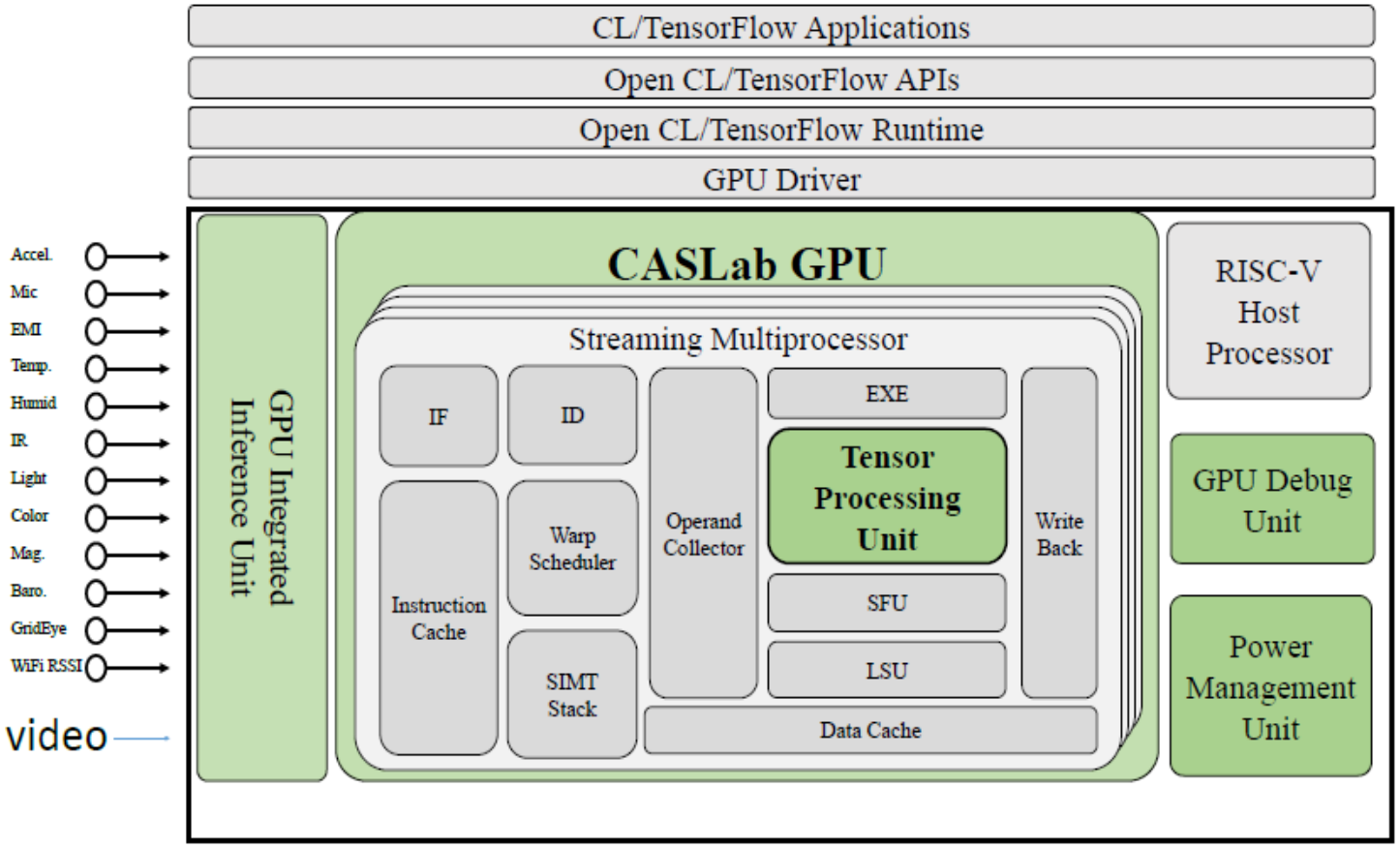
Terminology from Wiki

- **CUDA**
  - **Compute Unified Device Architecture** (**CUDA**) is a parallel computing architecture developed by Nvidia .
- **OpenCL**
  - **Open Computing Language** (**OpenCL**) is a framework for writing programs that execute across heterogeneous platforms consisting of central processing unit (CPUs), graphics processing unit (GPUs), and other processors.
  - OpenCL includes a language for writing *kernels* (functions that execute on OpenCL devices), plus application programming interfaces (APIs) that are used to define and then control the platforms.
  - OpenCL provides parallel computing using task-based and data-based parallelism. OpenCL is an open standard maintained by the non-profit technology consortium Khronos Group. It has been adopted by Intel, Advanced Micro Devices, Nvidia, and ARM Holdings.

Terminology from Wiki-cont.

- **PTX**
  - **Parallel Thread Execution (PTX)** is a pseudo-assembly language used in Nvidia's CUDA programming environment. The nvcc compiler translates code written in CUDA, a C-like language, into PTX, and the graphics driver contains a compiler which translates the PTX into a binary code which can be run on the processing cores.
- **LLVM**
  - **LLVM** (formerly Low Level Virtual Machine) is compiler infrastructure written in C++; it is designed for compile-time, link-time, run-time, and "idle-time" optimization of programs written in arbitrary programming languages.
  - **Compile the source code to the intermediate representation(LLVM-IR).**

Edge AI GPU


CL/TensorFlow Applications
Open CL/TensorFlow APIs
Open CL/TensorFlow Runtime
GPU Driver
Accel.
Mic
EMI
Temp.
Humid
IR
Light
Color
Mag.
Baro.
GridEye
WiFi RSSI
video
GPU Integrated Inference Unit
CASLab GPU
Streaming Multiprocessor
IF
ID
Instruction Cache
Warp Scheduler
SIMT Stack
Operand Collector
EXE
Tensor Processing Unit
SFU
LSU
Write Back
Data Cache
RISC-V Host Processor
GPU Debug Unit
Power Management Unit

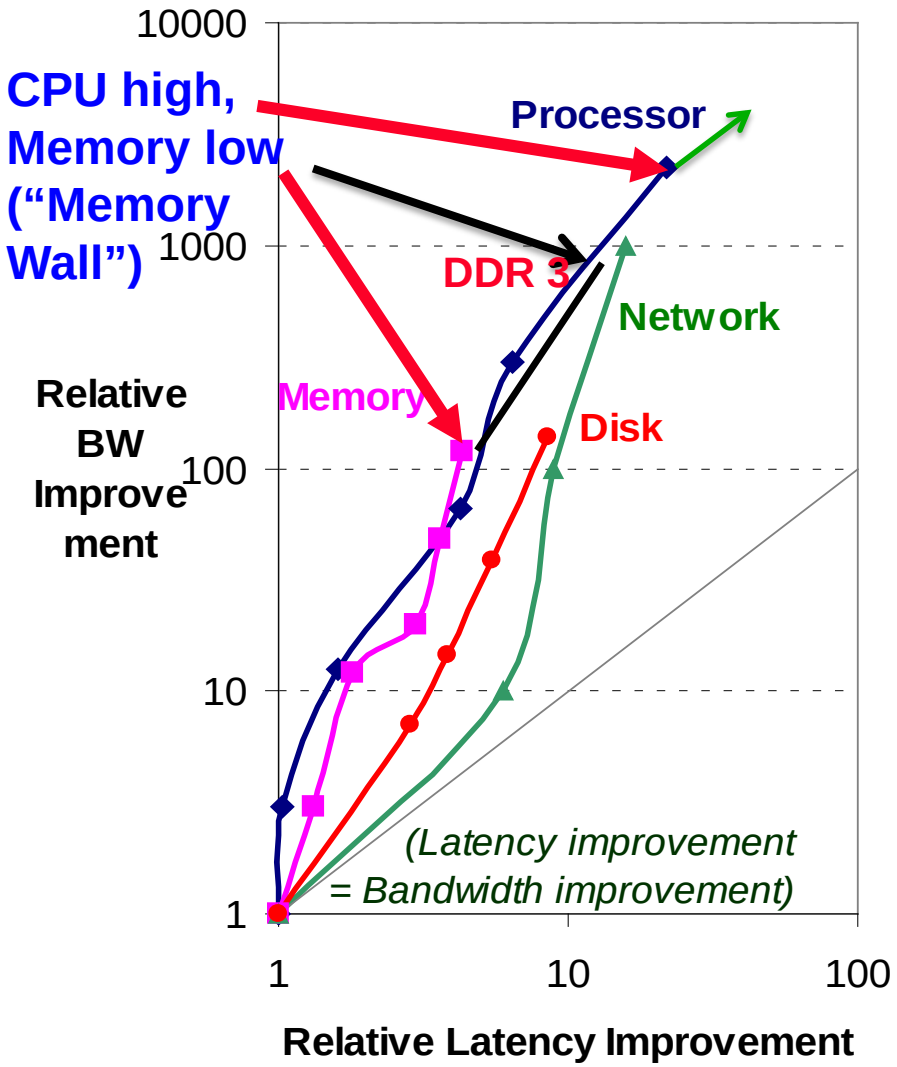
TensorFlow is an Open Source Software Library for Machine Intelligence

- TensorFlow™ is an open source software library for numerical computation using data flow graphs.
- Nodes in the graph represent mathematical operations, while the graph edges represent the multidimensional data arrays (tensors) communicated between them.

Tracking Technology Performance Trends

- Drill down into 4 technologies for a 20-year+ span:
  - Disks,
  - Memory,
  - Network,
  - Processors
- Compare ~1980 vs. ~2000+ Modern
  - Performance Milestones in each technology
- Compare for Bandwidth vs. Latency improvements in performance over time
- Bandwidth: number of events per unit time
  - E.g., M bits / second over network, M bytes / second from disk
- Latency: elapsed time for a single event
  - E.g., one-way network delay in microseconds, average disk access time in milliseconds

Latency Lags Bandwidth (last ~20+ years)



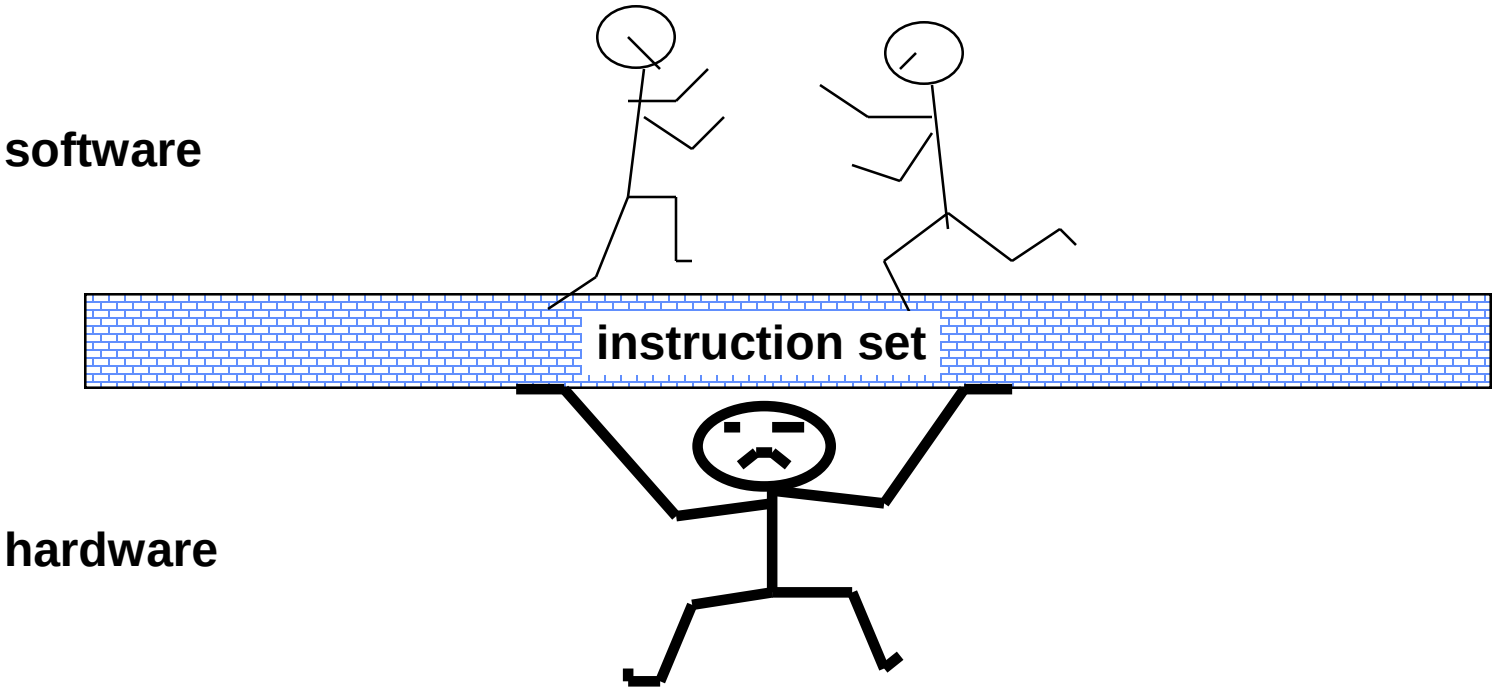
- **Performance Milestones**
- **Processor: '286, '386, '486, Pentium, Pentium Pro, Pentium 4 (21x,2250x), Core i7**
- **Ethernet: 10Mb, 100Mb, 1000Mb, 10000 Mb/s (16x,1000x)**
- **Memory Module: 16bit plain DRAM, Page Mode DRAM, 32b, 64b, SDRAM, DDR SDRAM (4x,120x) DDR3**
- **Disk : 3600, 5400, 7200, 10000, 15000 RPM (8x, 143x)**

A "Typical" RISC ISA

- 32-bit fixed format instruction (3 formats)
- 32 32-bit GPR (R0 contains zero, DP take pair)
- 3-address, reg-reg arithmetic instruction
- Single address mode for load/store:
  base + displacement
  - no indirection
- Simple branch conditions
- Delayed branch

see: SPARC, MIPS, HP PA-Risc, DEC Alpha, IBM PowerPC,
CDC 6600, CDC 7600, Cray-1, Cray-2, Cray-3

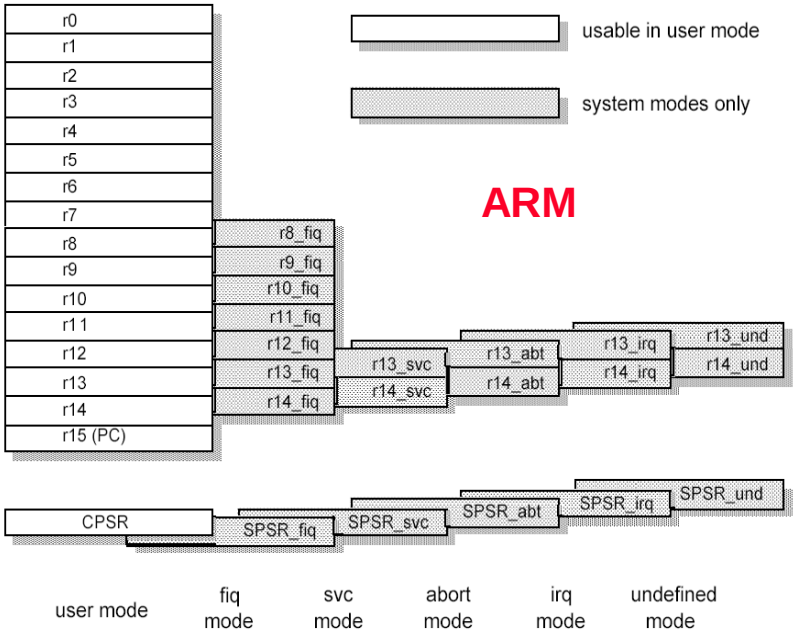
Instruction Set Architecture

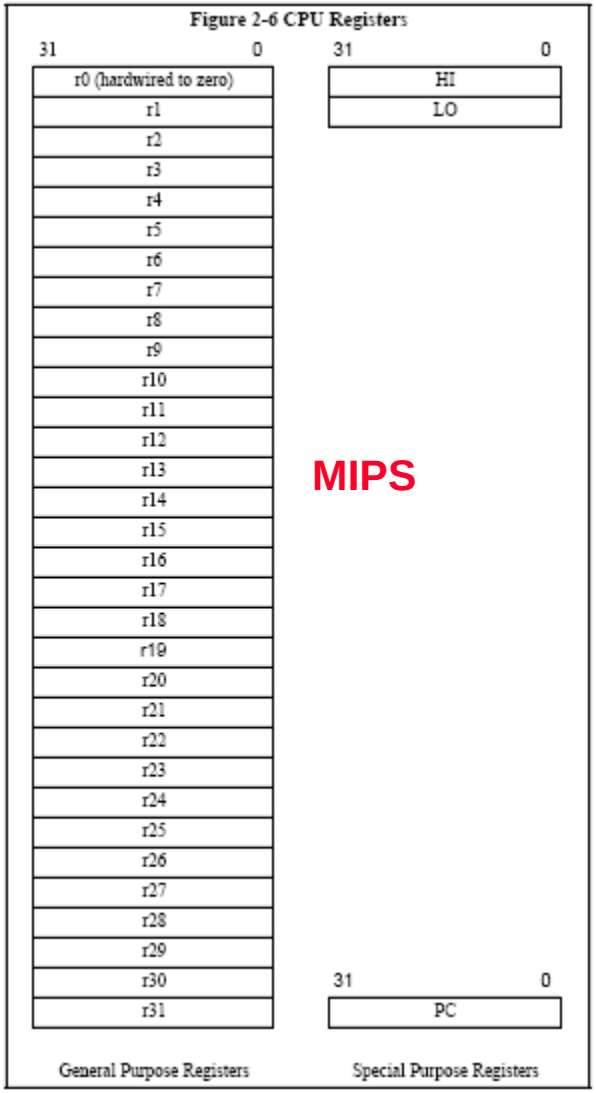


- **Properties of a good abstraction**
  - **Lasts through many generations (portability)**
  - **Used in many different ways (generality)**
  - **Provides convenient functionality to higher levels**
  - **Permits an efficient implementation at lower levels**

Register Bank

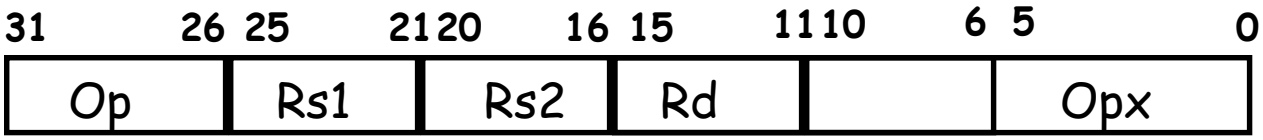
- **ARM : 37 registers**
  - **31 general-purpose registers**
  - **6 status registers**
- **MIPS : 35 registers**
  - **32 general-purpose registers**
  - **3 Special-Purpose Registers (PC, HI and LO)**



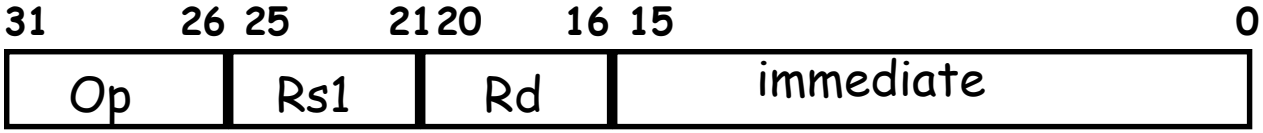


Example: MIPS

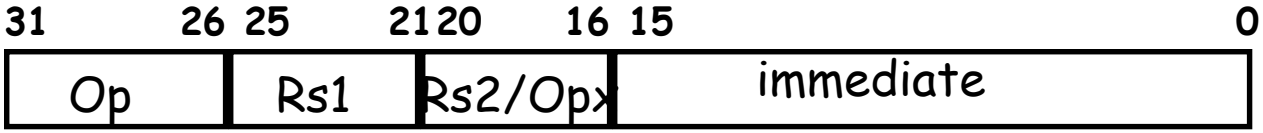
Register-Register



Register-Immediate

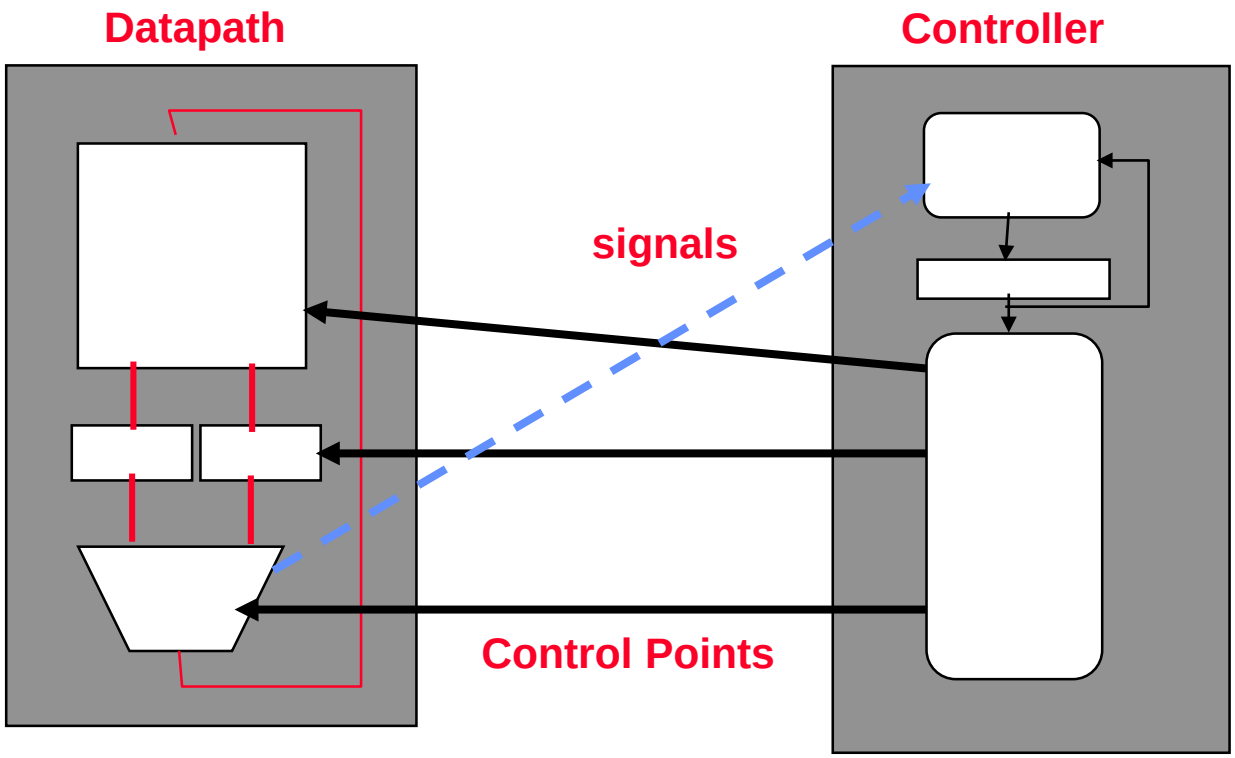


Branch



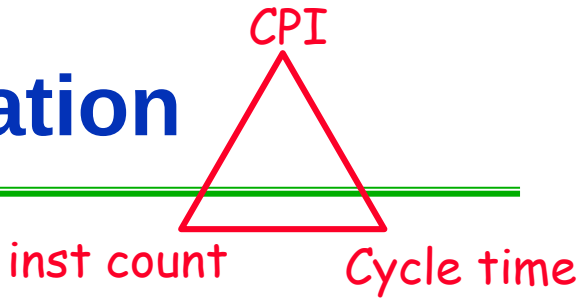
Jump / Call

Datapath vs Control



- **Datapath: Storage, FU, interconnect sufficient to perform the desired functions**
  - **Inputs are Control Points**
  - **Outputs are signals**
- **Controller: State machine to orchestrate operation on the data path**
  - **Based on desired function and signals**

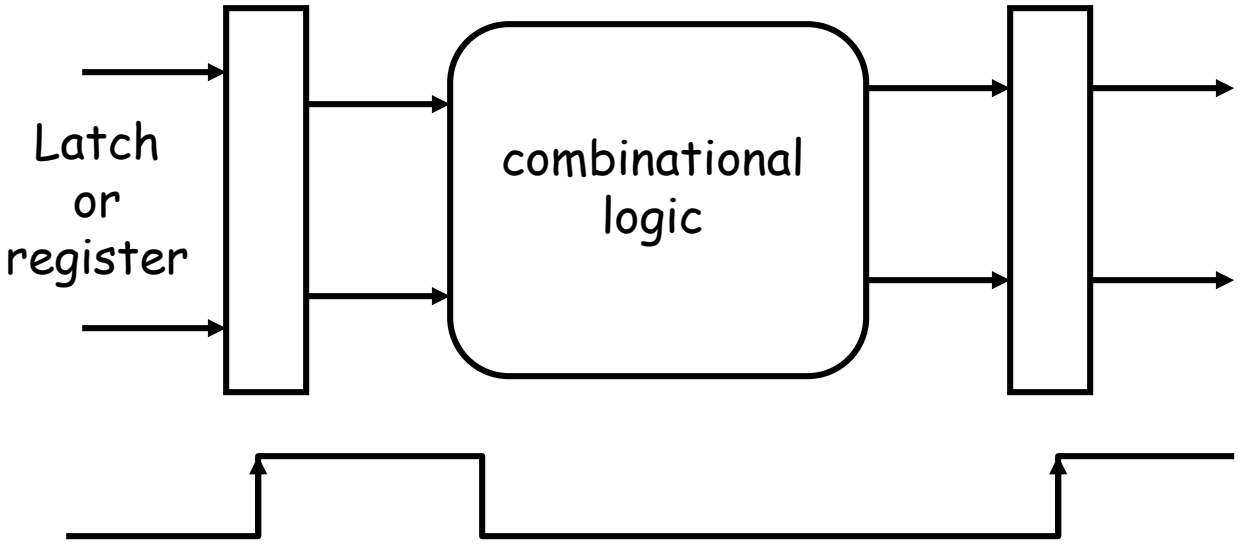
Processor Performance Equation



$$\text{CPU time} = \frac{\text{Seconds}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Cycle}}$$

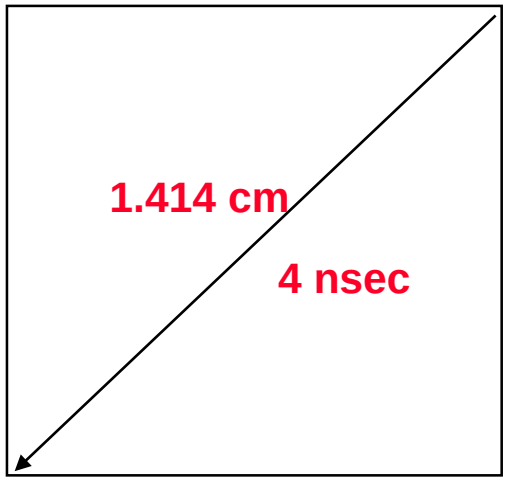
	Inst Count	CPI	Clock Rate
Program	X		
Compiler	X	(X)	
Inst. Set.	X	X	
Organization		X	X
Technology			X

What's a Clock Cycle?



- **Old days: 10 levels of gates**
- **Today: determined by numerous time-of-flight issues + gate delays**
  - **clock propagation, wire lengths, drivers**

Wire delays dominate



- **Gate delay**
  - **50ps (90nm)**
- **Long wire**
  - **1ns**
- **System bus runs through entire chip.**

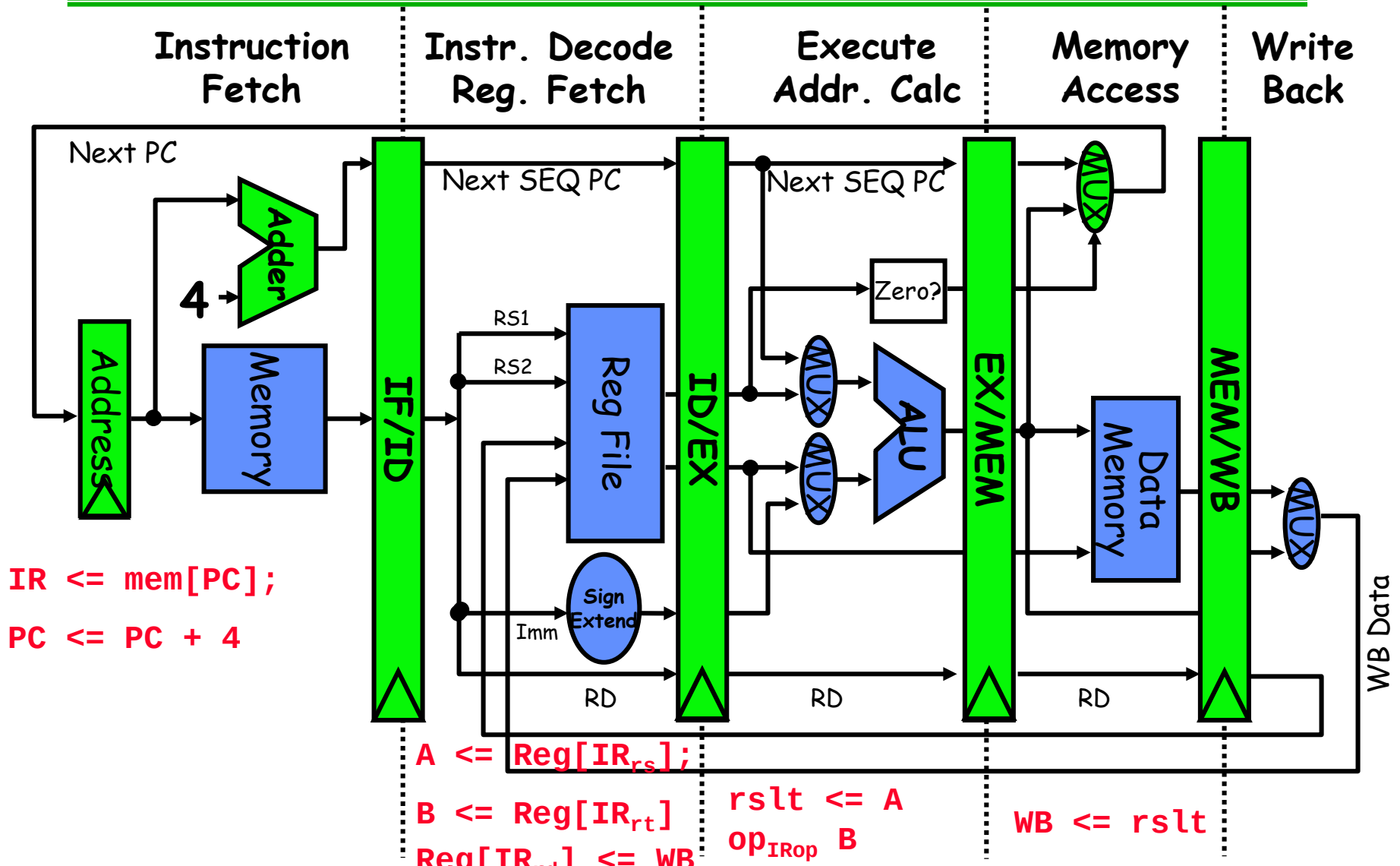
1.414 cm x 10^{-2}/3 x 10^{7}

= 4.7 nsec

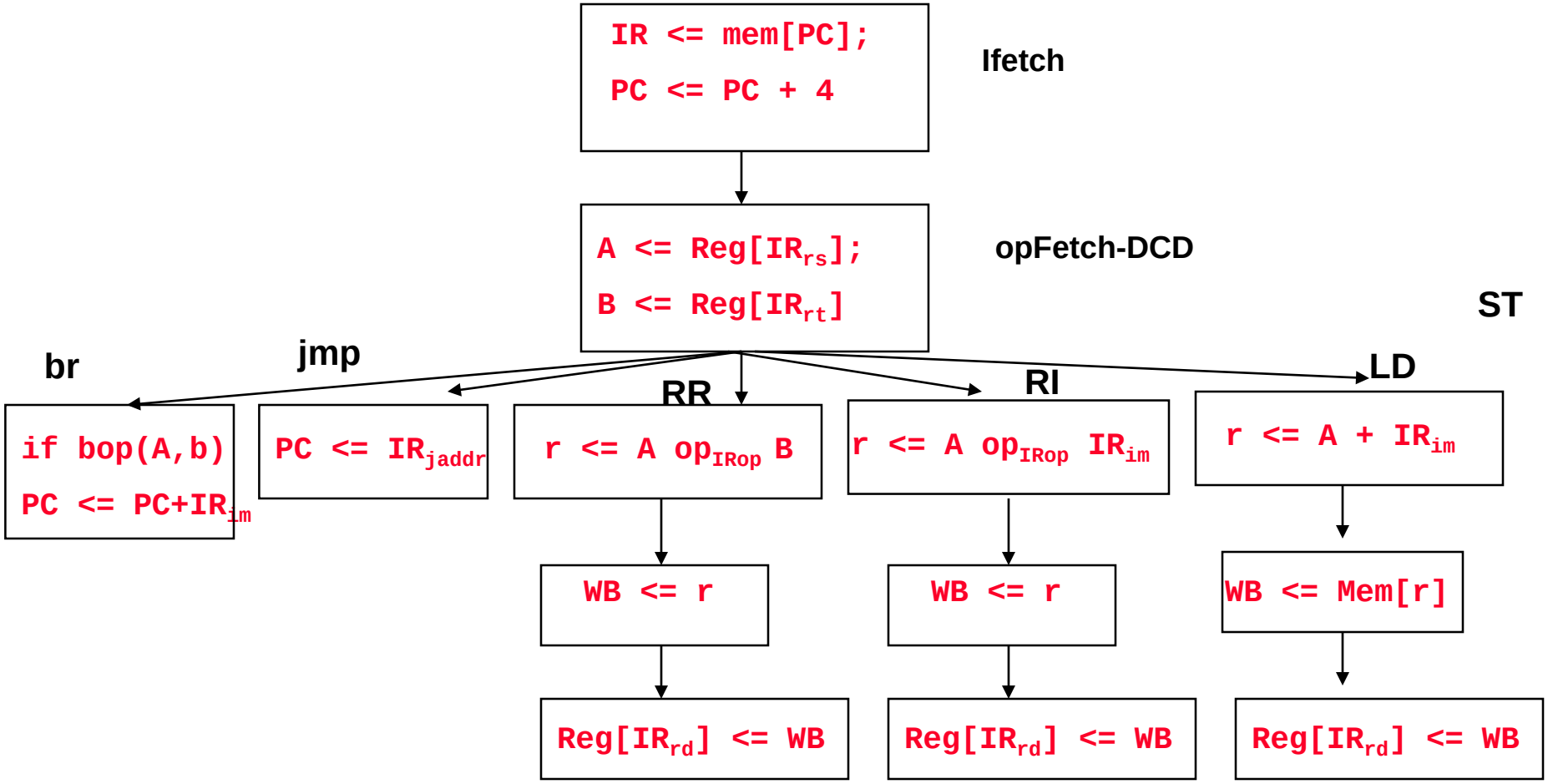
Approaching an ISA (Implementing an ISA)

- **Instruction Set Architecture**
  - **Defines set of operations, instruction format, hardware supported data types, named storage, addressing modes, sequencing**
- **Meaning of each instruction is described by RTL on *architected registers* and memory**
- **Given technology constraints assemble adequate datapath**
  - **Architected storage mapped to actual storage**
  - **Function units to do all the required operations**
  - **Possible additional storage (eg. MAR, …)**
  - **Interconnect to move information among regs and FUs**
- **Map each instruction to sequence of RTLs**
- **Collate sequences into symbolic controller state transition diagram (STD)**
- **Implement controller**

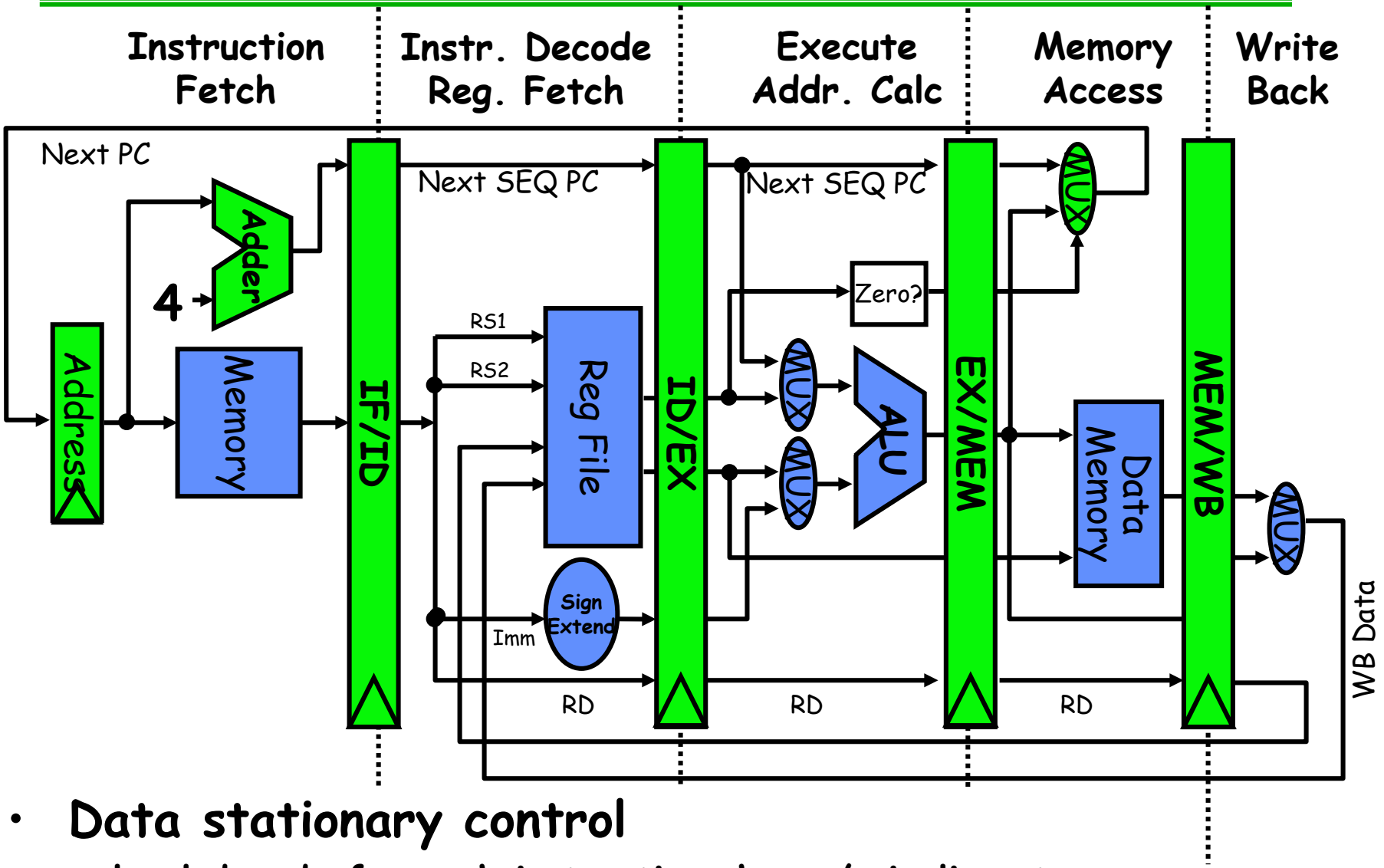
5 Steps of MIPS Datapath



Inst. Set Processor Controller

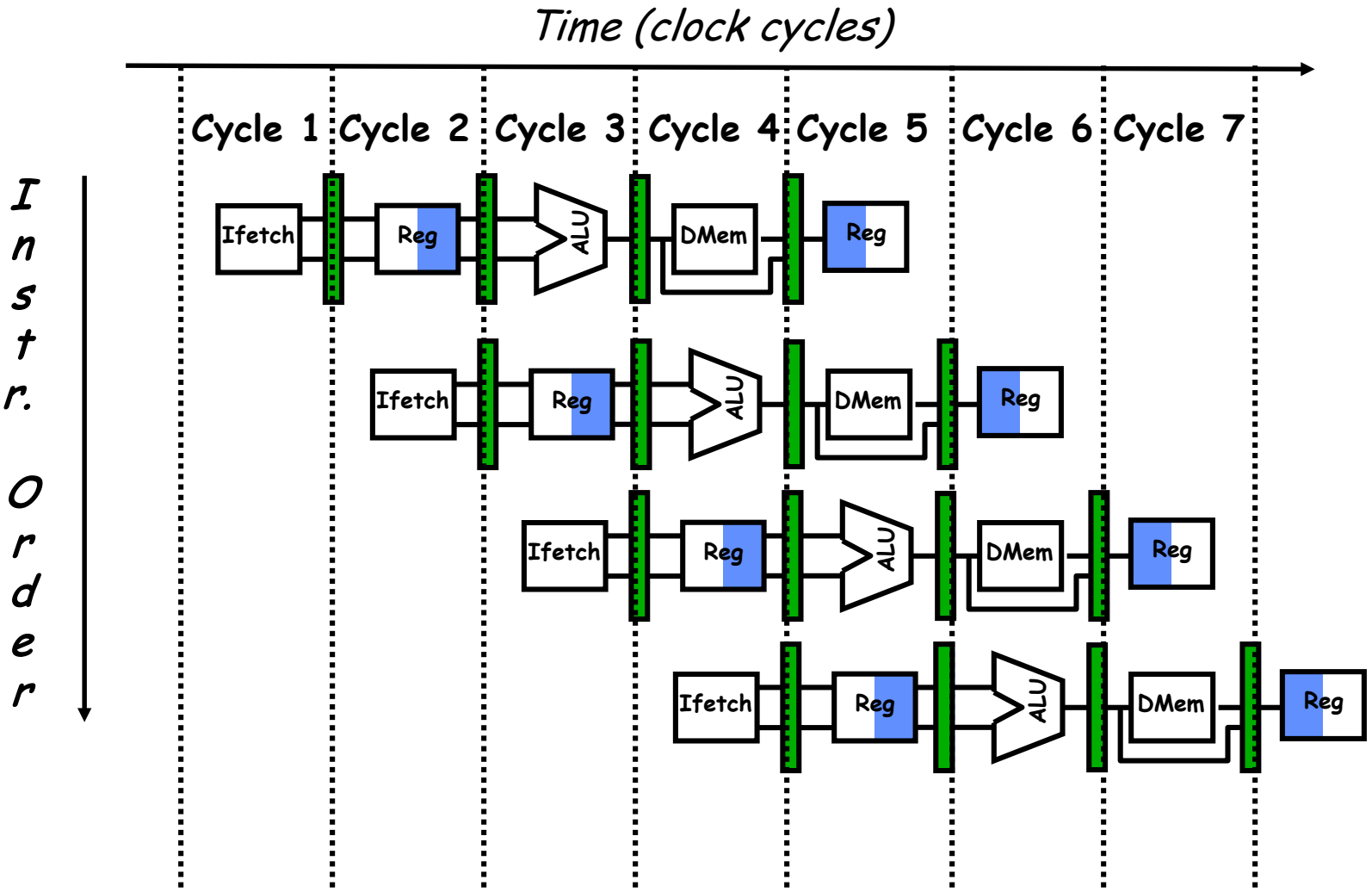


Control MIPS Datapath



- Data stationary control
  - local decode for each instruction phase / pipeline stage

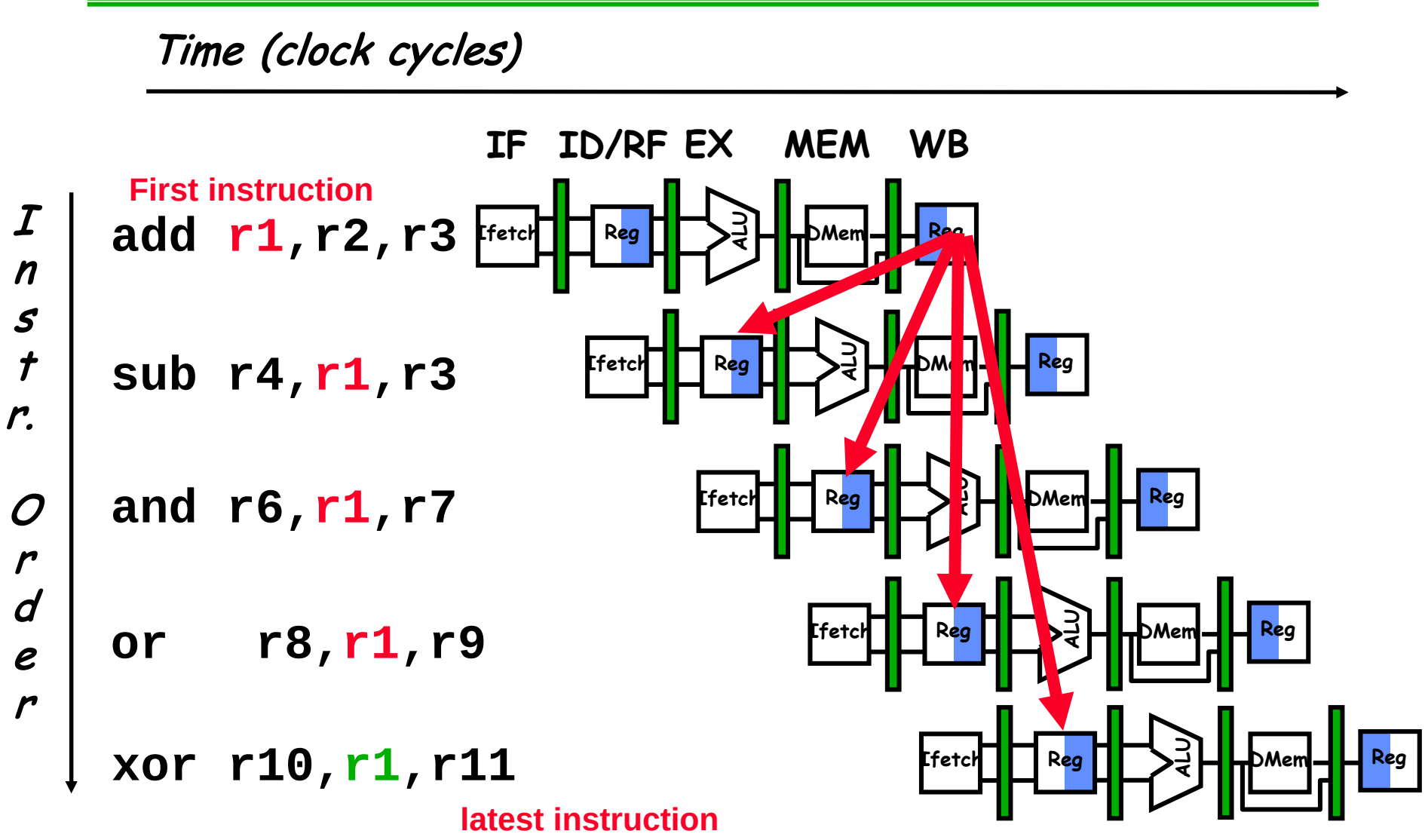
Visualizing Pipelining



Pipelining is not quite that easy!

- **Limits to pipelining: Hazards prevent next instruction from executing during its designated clock cycle**
  - **Structural hazards: HW cannot support this combination of instructions**
  - **Data hazards: Instruction depends on result of prior instruction still in the pipeline**
  - **Control hazards: Caused by delay between the fetching of instructions and decisions about changes in control flow (branches and jumps).**

Data Hazard on R1



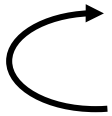
Three Generic Data Hazards

- **Read After Write (RAW)**
  $Instr_J$ tries to read operand before $Instr_I$ writes it

- **Caused by a "Dependence" (in compiler nomenclature). This hazard results from an actual need for communication.**

Three Generic Data Hazards

- **Write After Read (WAR)**
  $Instr_J$ writes operand *before* $Instr_I$ reads it



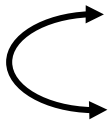
```
I: sub r4,r1,r3
J: add r1,r2,r3
K: mul r6,r1,r7
```

- **Called an "anti-dependence" by compiler writers. This results from reuse of the name "r1".**

- **Can't happen in MIPS 5 stage pipeline because:**
  - **All instructions take 5 stages, and**
  - **Reads are always in stage 2, and**
  - **Writes are always in stage 5**

Three Generic Data Hazards

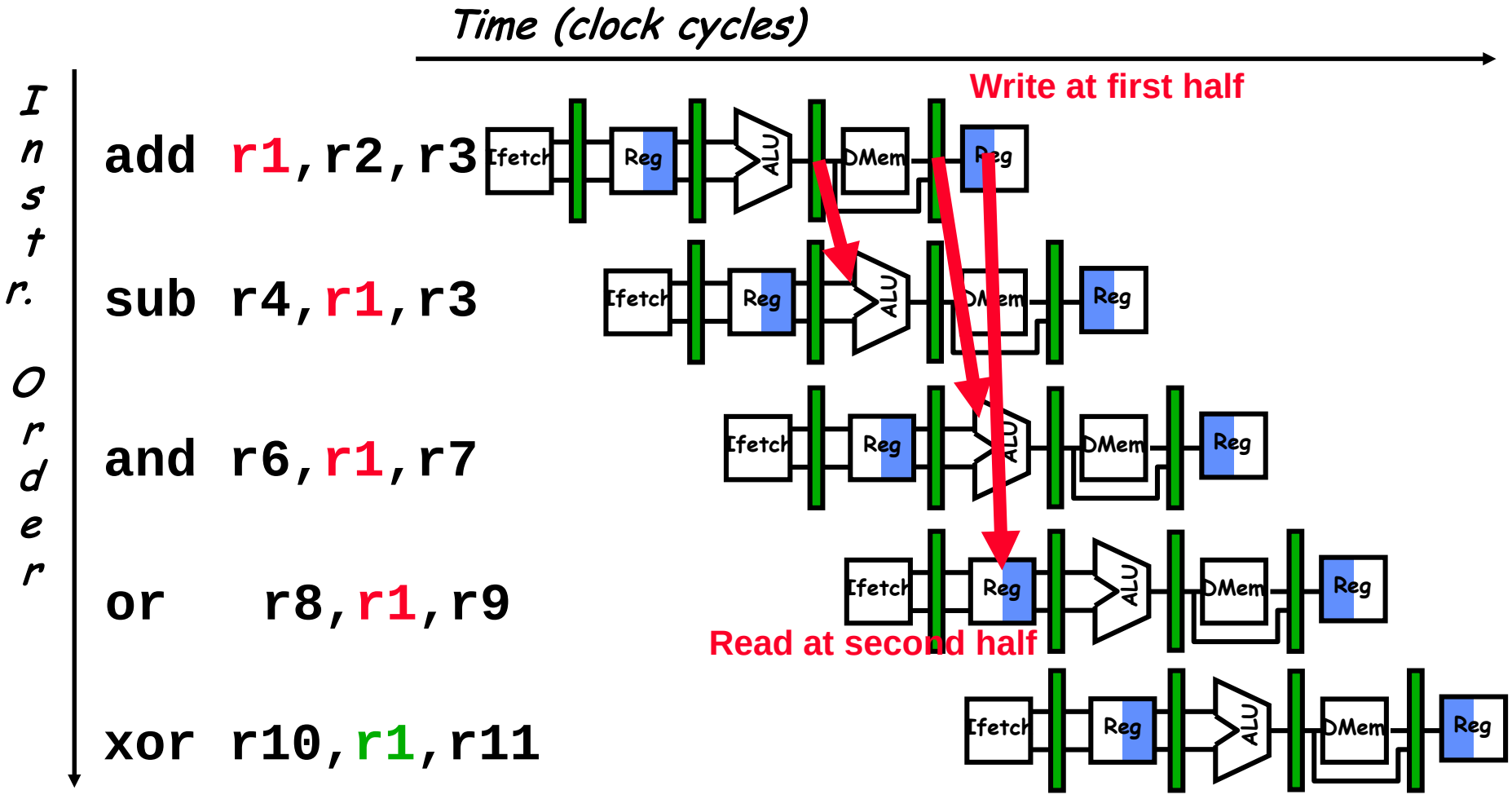
- **Write After Write (WAW)**
  $Instr_J$ writes operand *before* $Instr_I$ writes it.



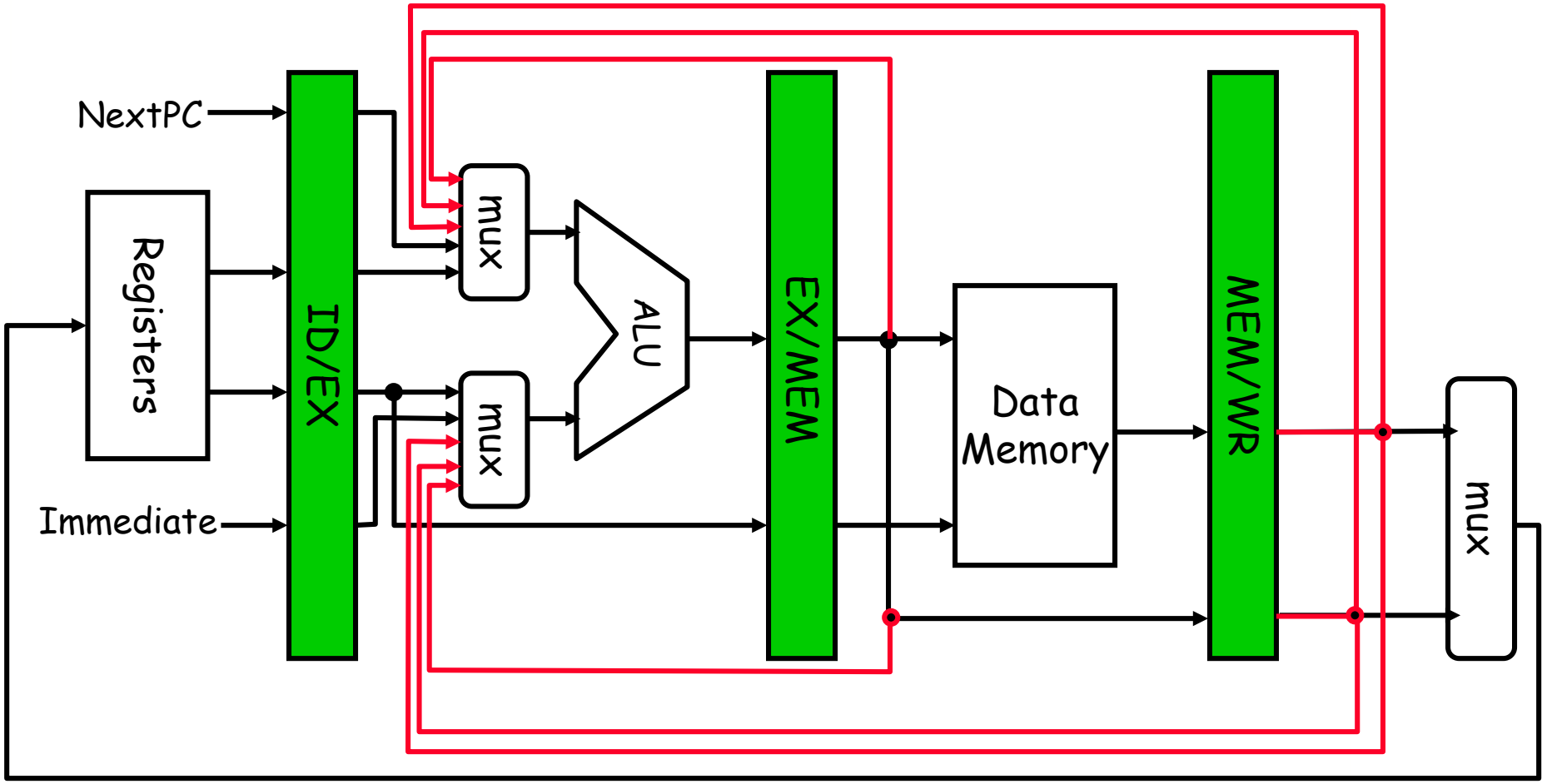
```
I: sub r1,r4,r3
J: add r1,r2,r3
K: mul r6,r1,r7
```

- **Called an "output dependence" by compiler writers This also results from the reuse of name "r1".**
- **Can't happen in MIPS 5 stage pipeline because:**
  - **All instructions take 5 stages, and**
  - **Writes are always in stage 5**
- **Will see WAR and WAW in more complicated pipes**

Forwarding to Avoid Data Hazard

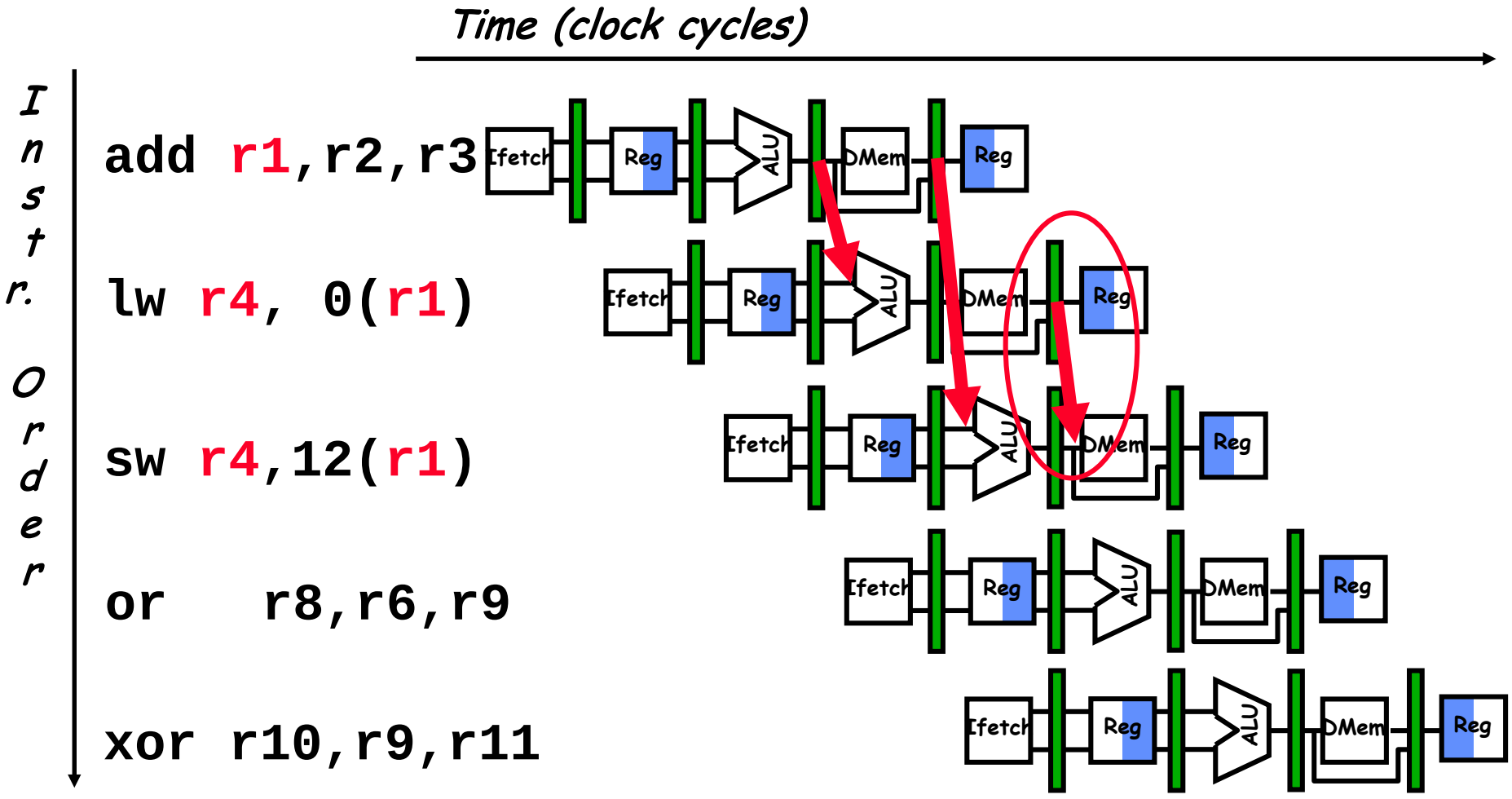


HW Change for Forwarding

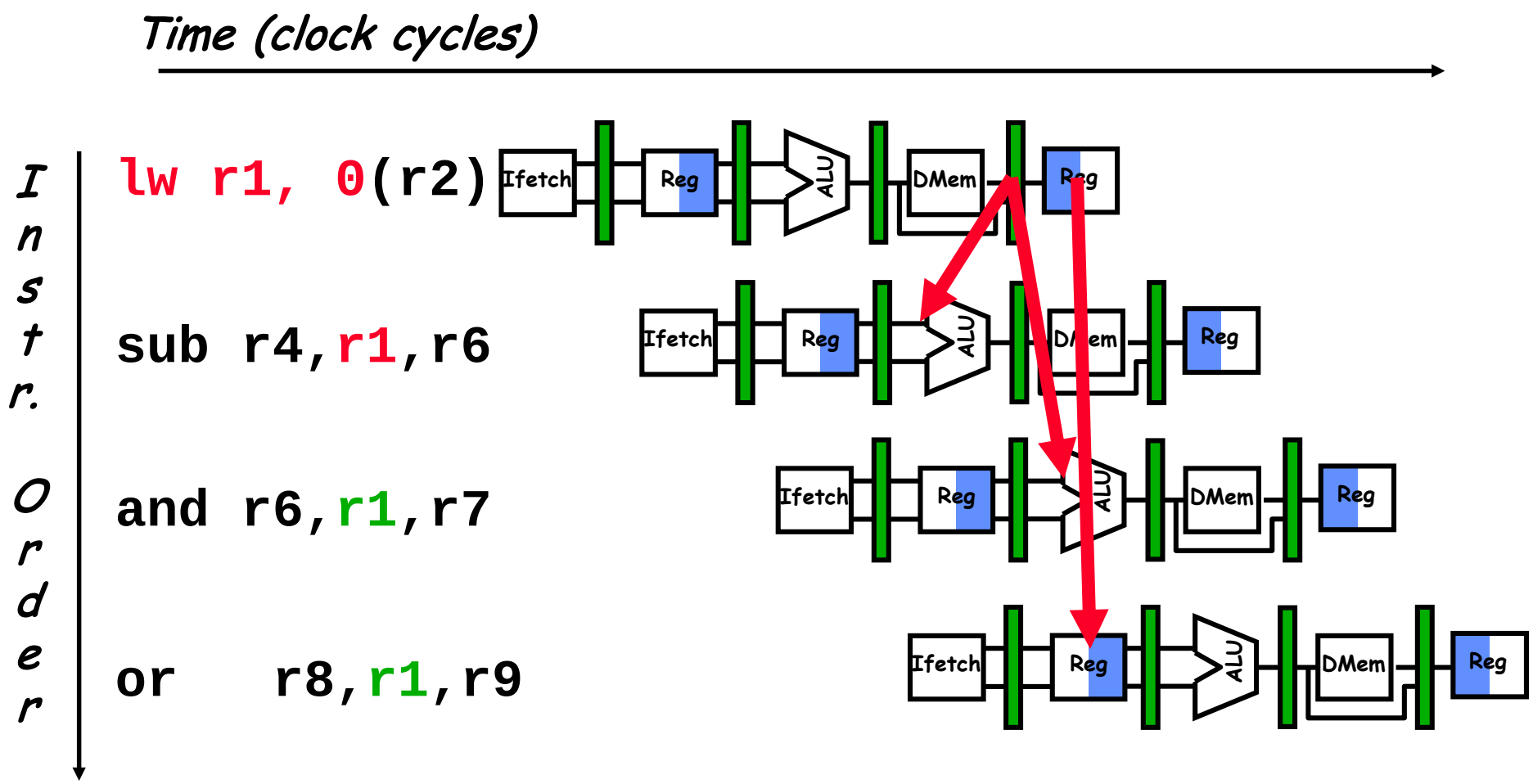


What circuit detects and resolves this hazard?

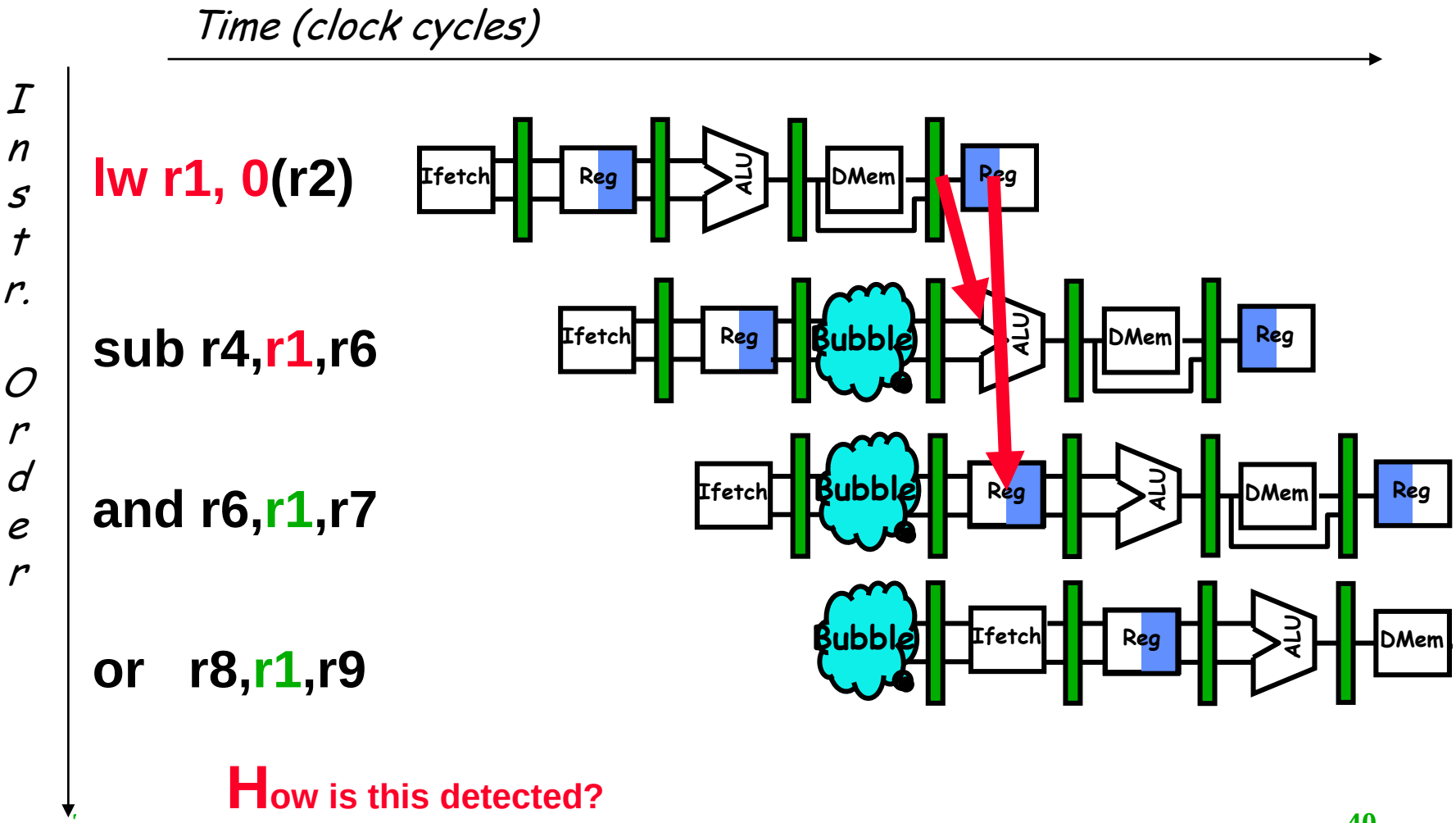
Forwarding to Avoid LW-SW Data Hazard



Data Hazard Even with Forwarding



Data Hazard Even with Forwarding



Software Scheduling to Avoid Load Hazards

Try producing fast code for

a = b + c;

d = e – f;

assuming a, b, c, d ,e, and f in memory.

Slow code:

	LW	Rb,b
	LW	Rc,c
stall		
	ADD	Ra,Rb,Rc
	SW	a,Ra
	LW	Re,e
	LW	Rf,f
stall		
	SUB	Rd,Re,Rf
	SW	d,Rd

Fast code:

LW	Rb,b
LW	Rc,c
LW	Re,e
ADD	Ra,Rb,Rc
LW	Rf,f
SW	a,Ra
SUB	Rd,Re,Rf
SW	d,Rd

Compiler optimizes for performance. Hardware checks for safety.

Control Hazard on Branches
Three Stage Stall

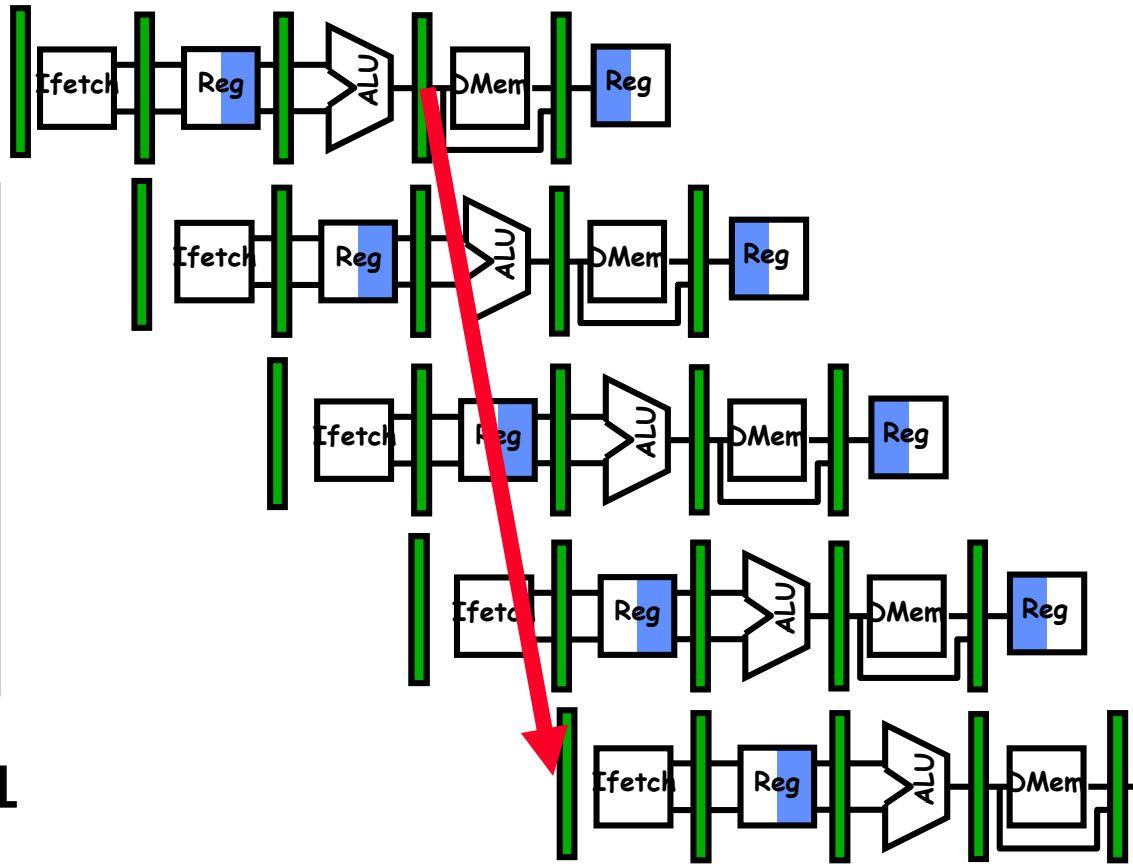
```
10: beq r1,r3,36

14: and r2,r3,r5

18: or  r6,r1,r7

22: add r8,r1,r9

36: xor r10,r1,r11
```



What do you do with the 3 instructions in between?

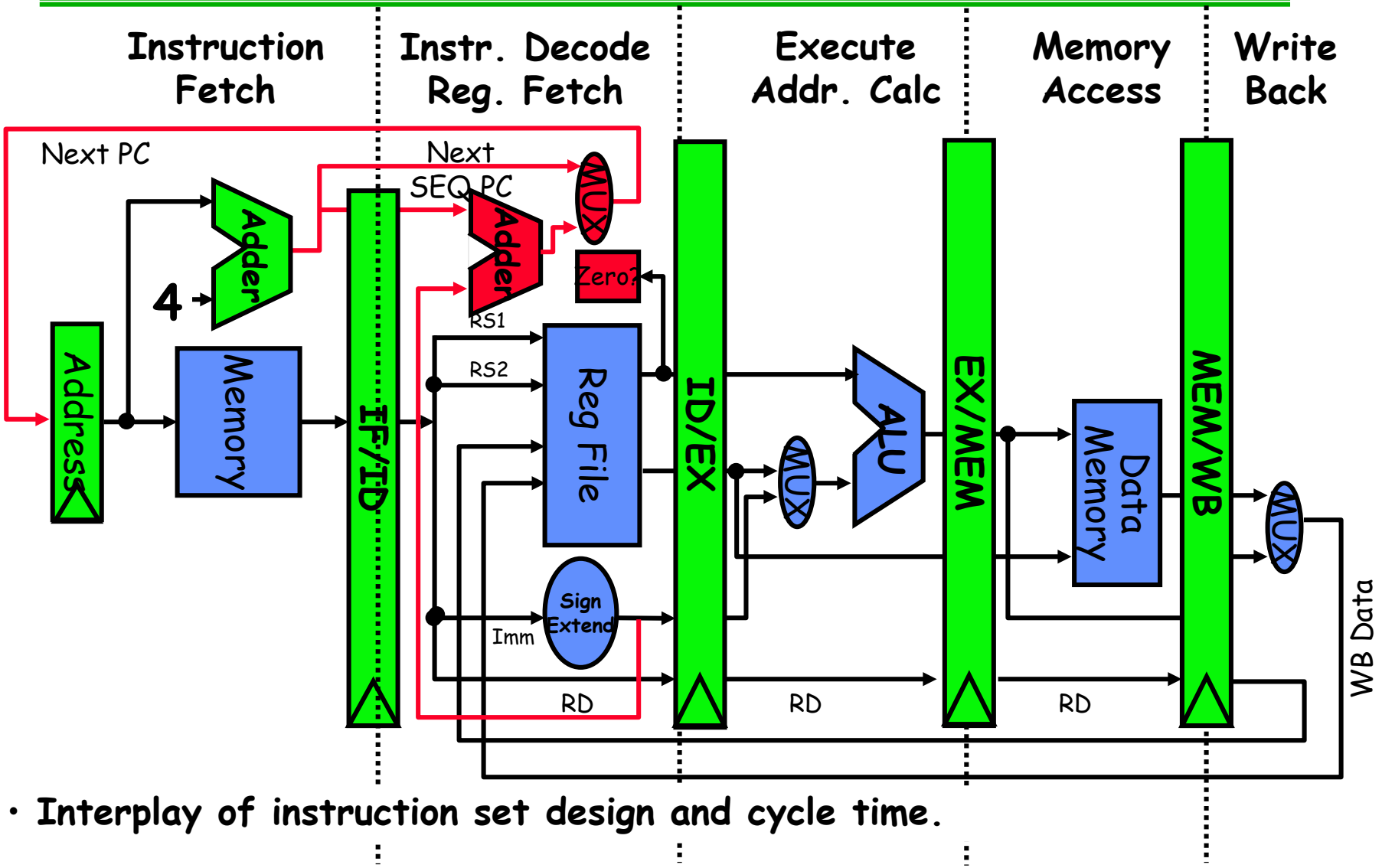
How do you do it?

Where is the “commit”?

Branch Stall Impact

- **If CPI = 1, 30% branch,**
  Stall 3 cycles => new CPI = 1.9!
- **Two part solution:**
  - **Determine branch taken or not sooner, AND**
  - **Compute taken branch address earlier**
- **MIPS branch tests if register = 0 or ≠ 0**
- **MIPS Solution:**
  - **Move Zero test to ID/RF stage**
  - **Adder to calculate new PC in ID/RF stage**
  - **1 clock cycle penalty for branch versus 3**

Branch result test at ID/RF stage



- Interplay of instruction set design and cycle time.

Branch Hazard Alternatives

#1: Stall until branch direction is clear

#2: Predict Branch Not Taken

- Execute successor instructions in sequence
- "Squash" instructions in pipeline if branch actually taken
- Advantage of late pipeline state update
- 47% MIPS branches not taken on average
- PC+4 already calculated, so use it to get next instruction

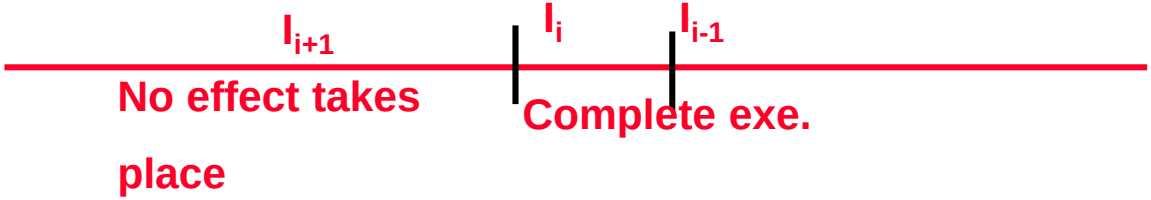
#3: Predict Branch Taken

- 53% MIPS branches taken on average
- But haven't calculated branch target address in MIPS

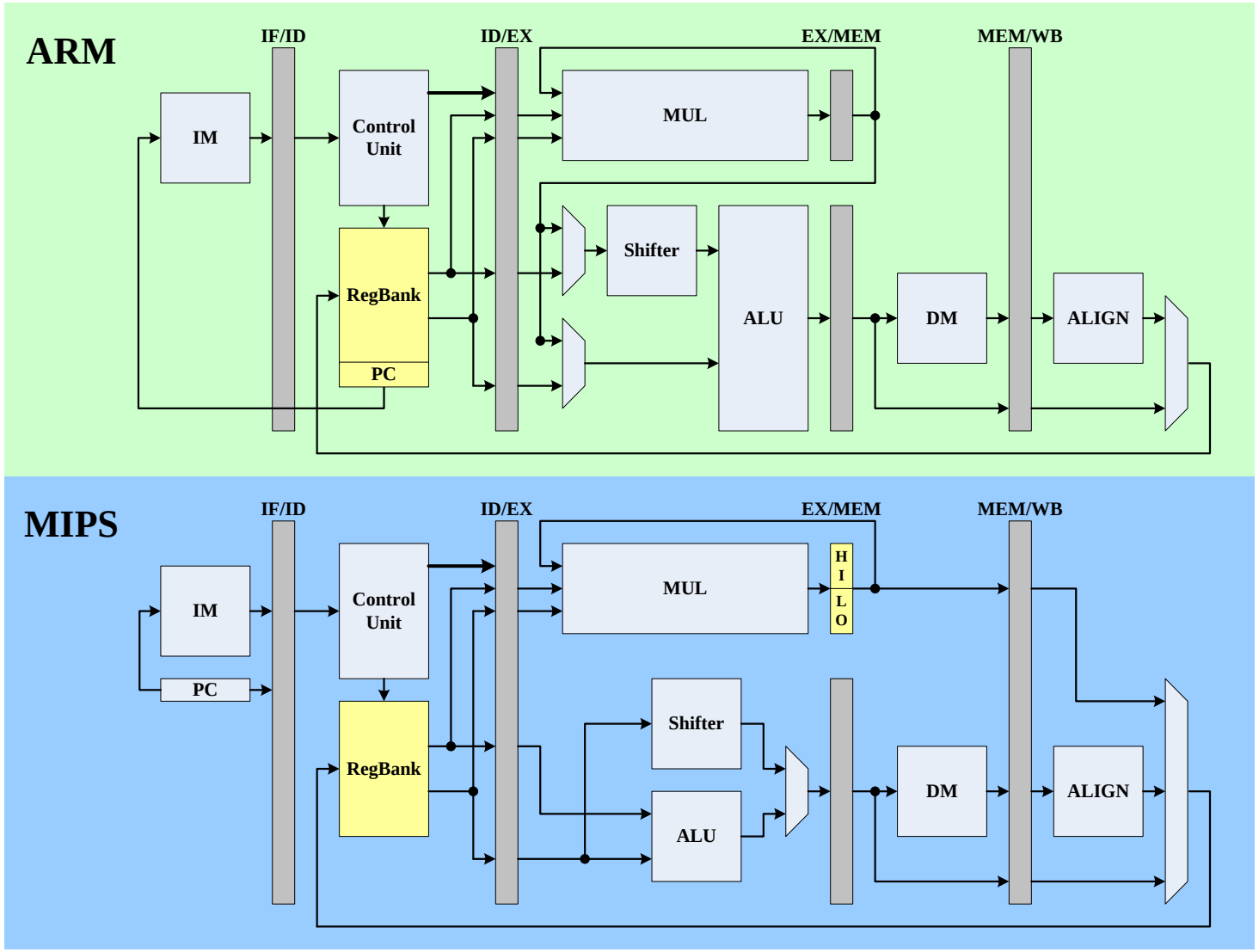
#4: Advanced processors use sophisticated predictors

Problems with Pipelining

- **Exception**: An unusual event happens to an instruction during its execution
  - Examples: divide by zero, undefined opcode
- **Interrupt**: Hardware signal to switch the processor to a new instruction stream
  - Example: a sound card interrupts when it needs more audio output samples (an audio "click" happens if it is left waiting)
- Problem: It must appear that the exception or interrupt must appear between 2 instructions (I_i and I_{i+1})
  - The effect of all instructions up to and including I_i is totally complete
  - No effect of any instruction after I_i can take place
- The interrupt (exception) handler either aborts program or restarts at instruction I_{i+1}



Pipeline


ARM
IF/ID
ID/EX
EX/MEM
MEM/WB
IM
Control Unit
MUL
Shifter
RegBank
PC
ALU
DM
ALIGN
MIPS
IF/ID
ID/EX
EX/MEM
MEM/WB
IM
Control Unit
MUL
H
I
L
O
PC
Shifter
RegBank
ALU
DM
ALIGN

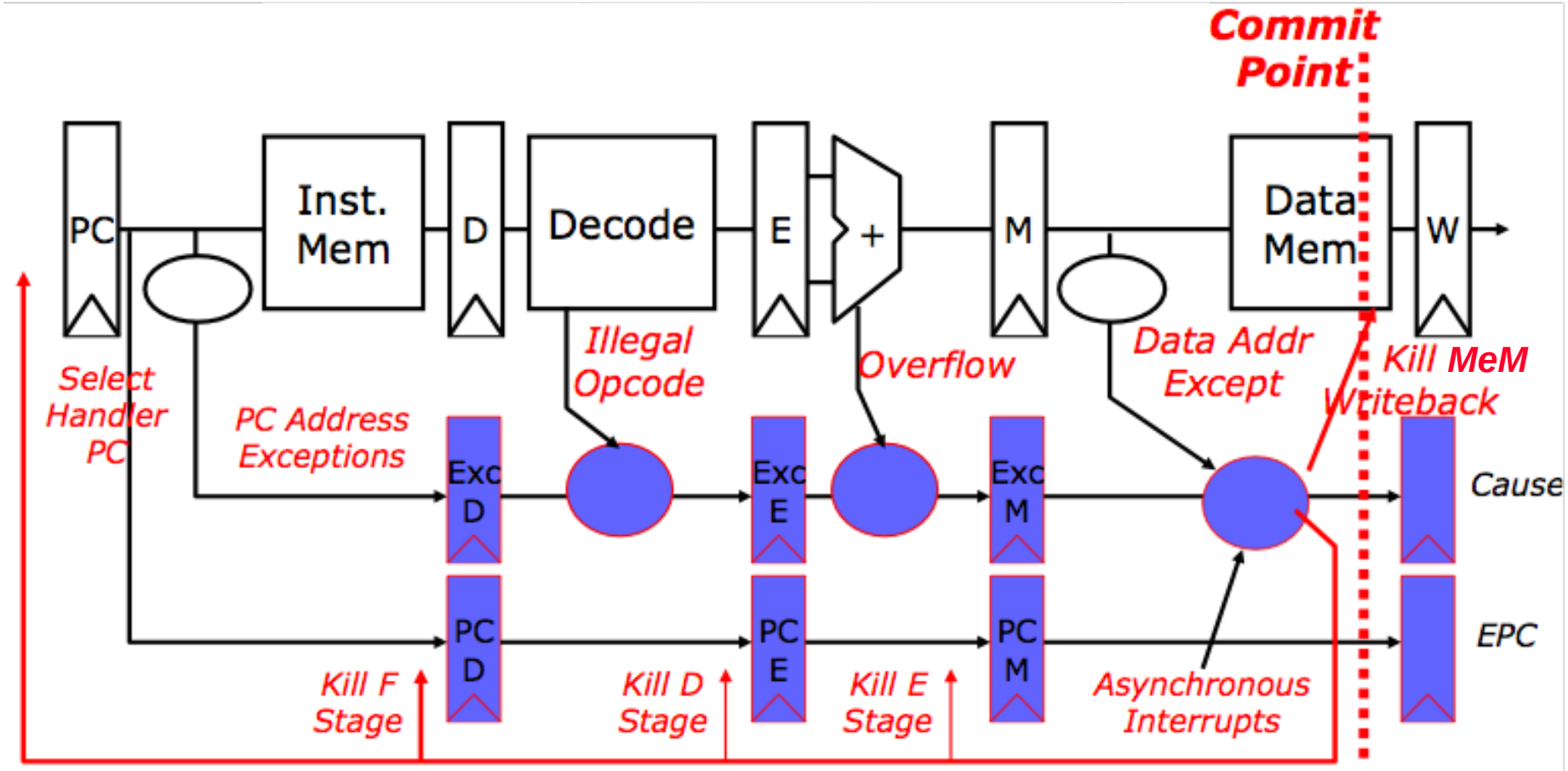
Processor Pipeline Model and Exception

Implement an in-order execution model :

a conventional five-stage pipeline

- **Instruction fetch (IF)**
  - **PC => ITLB => Instruction cache (physically addressed cache)**
- **Instruction decode (ID)**
- **Execute (EXE)**
- **Memory (MEM)**
  - **Load/store logical address => DTLB => Data cache**
- **Write back (WB).**

Precise Exceptions in Static Pipelines



Key observation: architected state only change in memory and register write stages.

Instruction with this PC is the faulting instru.

Exception Process in ARM (v4)

- **General process procedure**
  - » **When an exception occurs, the banked version of R14 and the SPSR for the exception mode are used to save the state.**

```
R14_<exception_mode> = return address;
SPSR_<exception mode> = CPSR;
CPSR[4:0] = exception mode number;
If <exception mode> == reset or FIQ then
    CPSR[6] = 1;              /* disable fast interrupt *
  /* Else CPSR[6] is unchanged *
    CPSR[7] = 1;              /* disable normal interrupt */
PC = exception vector address
```

Reset

- **Implementation**

  R14_svc = unpredictable value; SPSR_svc = unpredictable;

  CPSR[4:0] = 0b10011; /* Mode is supervisor. * /

  CPSR[I:F:T]=CPSR[7:6:5] = 110; /* Disable interrupts and execute ARM instructions */

  If high vectors configured then

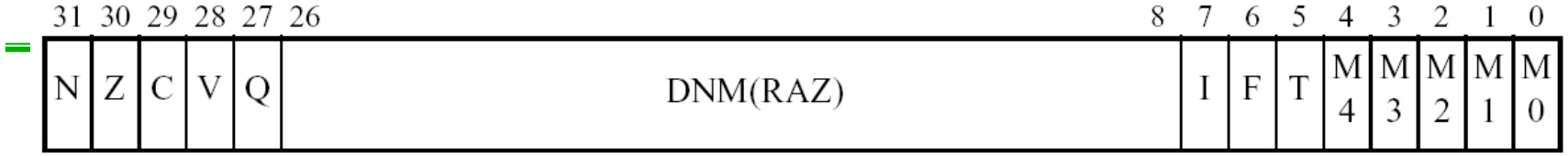
  PC = 0xFFFF0000

  Else

  PC = 0x00000000.

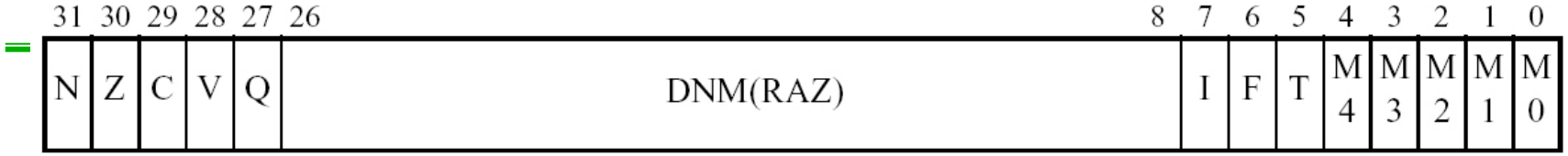
  – **The above actions are done in Hardware. This is how the processor fetches the first instruction. Set the program counter to one of the above values by checking the configuration pin.**

Program Status Register



- 4 condition code flags
  - (N, Z, C, V) flags：Negative, Zero, Carry, oVerflow
- 1 sticky overflow flag
  - Q bit：DSP instruction overflow bit.
  - In E variants of ARM architecture 5 and above.
- 2 interrupt disable bits
  - I bit：disable normal interrupt (IRQ)
  - F bit：disable fast interrupt (FIQ)
- 1 bit which encodes whether ARM or Thumb instructions are being executed.
  - T bit

Program Status Register (cont.)



- **5 bits that encode the current processor mode.**
  - **M[4:0] are the mode bits.**

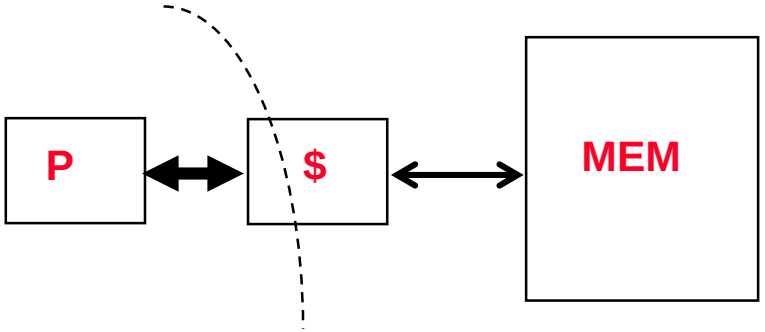
M[4:0]	Mode	Accessible registers
0b10000	User	PC, R14 to R0, CPSR
0b10001	FIQ	PC, R14_fiq to R8_fiq, R7 to R0, CPSR, SPSR_fiq
0b10010	IRQ	PC, R14_irq, R13_irq, R12 to R0, CPSR, SPSR_irq
0b10011	Supervisor	PC, R14_svc, R13_svc, R12 to R0, CPSR, SPSR_svc
0b10111	Abort	PC, R14_abt, R13_abt, R12 to R0, CPSR, SPSR_abt
0b11011	Undefined	PC, R14_und, R13_und, R12 to R0, CPSR, SPSR_und
0b11111	System	PC, R14 to R0, CPSR (ARM architecture v4 and above)

1) Taking Advantage of Parallelism

- **Increasing throughput of server computer via multiple processors or multiple disks**
- **Detailed HW design**
  - **Carry lookahead adders uses parallelism to speed up computing sums from linear to logarithmic in number of bits per operand**
  - **Multiple memory banks searched in parallel in set-associative caches**
- **Pipelining: overlap instruction execution to reduce the total time to complete an instruction sequence**
  - **Not every instruction depends on immediate predecessor ⇒ executing instructions completely/partially in parallel is possible**
  - **Classic 5-stage pipeline:**
    1) Instruction Fetch (Ifetch),
    2) Register Read (Reg),
    3) Execute (ALU),
    4) Data Memory Access (Dmem),
    5) Register Write (Reg)

2) The Principle of Locality

- **The Principle of Locality:**
  - **Program access a relatively small portion of the address space at any instant of time.**
- **Two Different Types of Locality:**
  - **Temporal Locality (Locality in Time): If an item is referenced, it will tend to be referenced again soon (e.g., loops, reuse)**
  - **Spatial Locality (Locality in Space): If an item is referenced, items whose addresses are close by tend to be referenced soon (e.g., straight-line code, array access)**
- **Last 30 years, HW relied on locality for memory perf.**



3) Focus on the Common Case

- **Common sense guides computer design**
  - **Since it's engineering, common sense is valuable**
- **In making a design trade-off, favor the frequent case over the infrequent case**
  - **E.g., Instruction fetch and decode unit used more frequently than multiplier, so optimize it 1st**
  - **E.g., If database server has 50 disks / processor, storage dependability dominates system dependability, so optimize it 1st**
- **Frequent case is often simpler and can be done faster than the infrequent case**
  - **E.g., overflow is rare when adding 2 numbers, so improve performance by optimizing more common case of no overflow**
  - **May slow down overflow, but overall performance improved by optimizing for the normal case**
- **What is frequent case and how much performance improved by making case faster => Amdahl's Law**

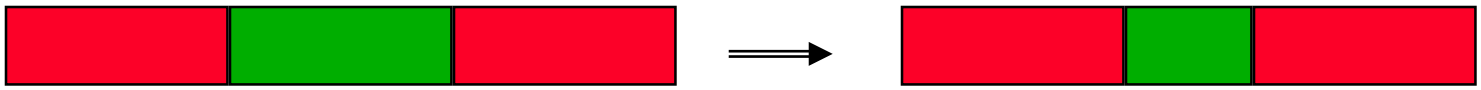
4) Amdahl's Law

$$ExTime_{new} = ExTime_{old} \times \left[\left(1 - Fraction_{enhanced}\right) + \frac{Fraction_{enhanced}}{Speedup_{enhanced}} \right]$$

$$Speedup_{overall} = \frac{ExTime_{old}}{ExTime_{new}} = \frac{1}{\left(1 - Fraction_{enhanced}\right) + \frac{Fraction_{enhanced}}{Speedup_{enhanced}}}$$

Best you could ever hope to do:

$$Speedup_{maximum} = \frac{1}{\left(1 - Fraction_{enhanced}\right)}$$



Amdahl's Law Example

- **New CPU 10X faster**
- **I/O bound server, so 60% time waiting for I/O**

$$\text{Speedup}_{\text{overall}} = \frac{1}{\left(1 - \text{Fraction}_{\text{enhanced}}\right) + \frac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}}}$$

$$= \frac{1}{\left(1 - 0.4\right) + \frac{0.4}{10}} = \frac{1}{0.64} = 1.56$$

- **Apparently, it's human nature to be attracted by 10X faster, vs. keeping in perspective it's just 1.6X faster**

And In Conclusion

- **Apply the simple and useful technology to high-end applications**
  1. **Take Advantage of Parallelism**
  2. **Principle of Locality**
  3. **Focus on the Common Case**
  4. **Amdahl's Law**
- **Hazards limit performance**
  - **Structural: need more HW resources**
  - **Data (RAW,WAR,WAW): need forwarding, compiler scheduling**
  - **Control: branch prediction (speculation execution, recovery technique)**
- **Exceptions and Interrupts add complexity**