



Computer Architecture

Lecturer: Mihai Negru

2nd Year, Computer Science

Lecture 1: Introduction

<http://users.utcluj.ro/~negrum/>



Course Objectives



- The lecture classes are **mandatory!**
- Provide the students with the necessary information
 - Understand: ISA, micro-architectures, CPU design methods, memory hierarchy, CPU performance improvement
 - Specification, design and implement CPUs, micro-architectures, data-paths and control units
 - To understand the new tendencies in computer architectures
- Prerequisites: Logic design, Digital System Design, VHDL Prog.

- $2C + 2L - 14$ weeks
- **Assessment:**
 - Written examination: E
 - Lab activity: L
 - Homework: H

$$Lab = \frac{L + H}{2}$$

$$Grade = 0.5 \cdot E + 0.5 \cdot Lab$$

Pass Condition

$$E \geq 4.5$$

$$Lab \geq 5.0$$



Course Content



- Introduction
- High Level Synthesis – HLS
- Instruction Set Architecture – ISA
- CPU Design – Single Cycle
- ALU Design
- CPU Design – Multi-Cycle
- CPU Design – Pipeline
- Advanced Pipelining – Static and Dynamic Scheduling of Execution
- Branch Prediction
- Superscalar Architectures
- Memory Hierarchy
- Modern CPU Architectures



Laboratory Objectives



- The laboratory classes and homework are **mandatory!**
- Teach students to operate with the concepts presented during the lectures
- Develop practical skills in machine language programming, design and implementation of micro-architectures using RTL and VHDL
- Design with Xilinx Development Tools and FPGA boards
- Design synthesizable VHDL hardware components → FPGA
- MIPS assembly language, running simple programs on the designed CPU
- Design and implementation (VHDL) of MIPS micro-architectures and testing on FPGA boards
- The homework helps students in improving their problem solving abilities



Laboratory Content



- Introduction to Xilinx ISE / **VIVADO** Design Suite
- **VHDL** programming
- Combinational Circuits
- Sequential Circuits
- Memories
- Single Cycle CPU Design
- Pipeline CPU Design
- UART Interface
- I/O Communication
- CPU testing
- CPU presentation



Bibliography



1. D. A. Patterson, J. L. Hennessy, "Computer Organization and Design: The Hardware/Software Interface", 3th edition, ed. Morgan–Kaufmann, 2005
2. D. A. Patterson, J. L. Hennessy, "Computer Organization and Design: The Hardware/Software Interface", 5th edition, ed. Morgan–Kaufmann, 2013
3. D. A. Patterson and J. L. Hennessy, "Computer Organization and Design: A Quantitative Approach", 5th edition, ed. Morgan-Kaufmann, 2011
4. D. M. Harris, S. L. Harris, *Digital Design and Computer Architecture*, Morgan Kaufmann, San Francisco, 2007
5. D. A. Patterson, J. L. Hennessy, "ORGANIZAREA SI PROIECTAREA CALCULATOARELOR. INTERFATA HARDWARE/SOFTWARE", Editura ALL, Romania, ISBN: 973-684-444-7
6. MIPS32™ Architecture for Programmers, Volume I: "Introduction to the MIPS32™ Architecture".
7. MIPS32™ Architecture for Programmers Volume II: "The MIPS32™ Instruction Set".
8. World Wide Web ...



Levels of abstraction of a computing system



| |
|----------------------|
| Application Software |
| Operating Systems |
| Architecture |
| Micro-architecture |
| Logic |
| Digital Circuits |
| Analog Circuits |
| Devices |
| Physics |

Programs
Device Drivers
Instructions
Registers
Datapaths
Controllers
Adders
Memories
AND gates
NOT gates
Amplifiers
Filters
Transistors
Diodes
Electrons

Applications that run on a computer

Digital Circuits, Logic Gates, Register Transfer Level (RTL), Micro-Architecture

OUR FOCUS IS HERE

Electronic Circuits and Devices



Basic Concepts



- Architecture – the interface between a user and an object
- Computer Architecture
 - Instruction Set Architecture (ISA)
 - Computer Organization – micro-architecture
- ISA: the interface between Hardware and low-level Software
- Micro-architecture: components and connections between them
 - Registers, ALU, Memory, Shifters, Logic Units, ...
- The same ISA can have different organizations:
 - MIPS single-cycle, multi-cycle, pipeline
- A specific architecture can be implemented by different micro-architectures with different price/performance/power constraints
- ISA Examples: IA-32, IA-64, MIPS, SPARC, ARM, etc.



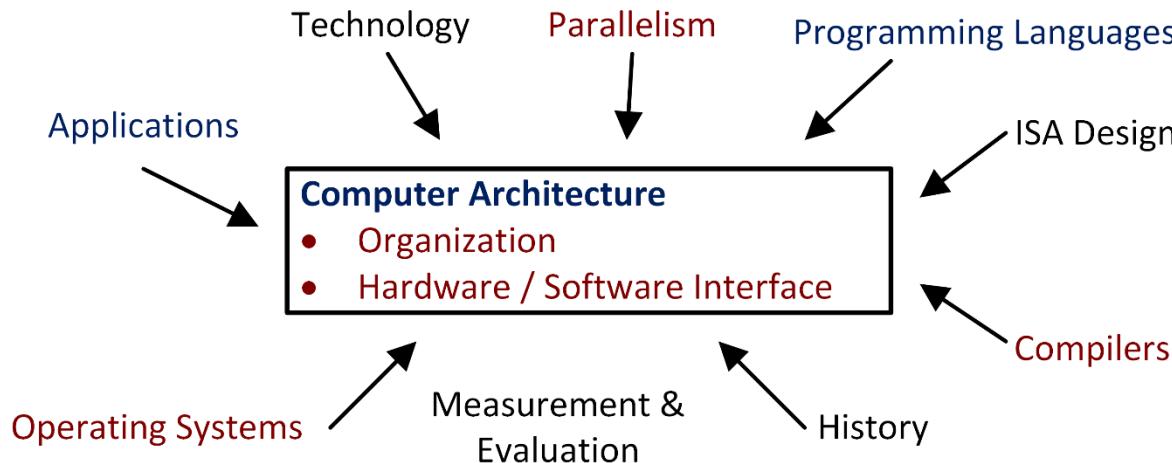
Basic Concepts



- Recommendations to **Manage Complexity**
 - **Hierarchy** – dividing the system into modules and sub-modules, until the pieces are easy to understand
 - **Modularity** – the modules must have well defined functions and interfaces, in order to be easily integrated
 - **Regularity** – uniformity among modules → reusable modules, in order to reduce the number of modules that must be designed
- A computer architect designs a computer that must fulfill
 - Functional requirements
 - Price/Power/Performance/Availability constraints

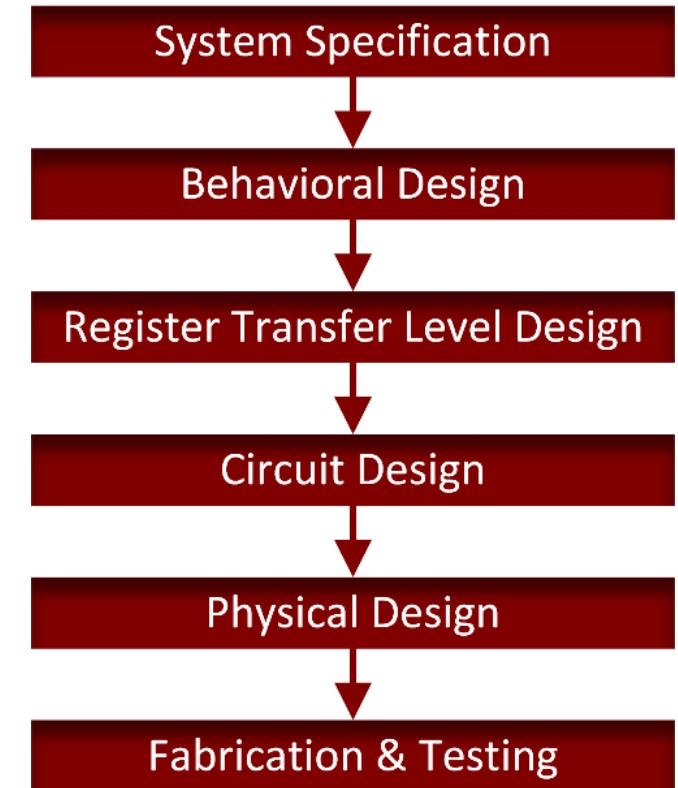


High Level Synthesis → Logic Synthesis → Layout Synthesis



Factors that influence the design process

- Synthesis is the automatic mapping from a high-level description to a low-level description
- High Level Synthesis or Architectural Synthesis
 - having a description of circuit behavior, create a Register Transfer Level (RTL) architecture that implements the circuit



Design on levels of abstraction
Top-down



Parallelism Types



- 2 types of parallelism (application specific point of view)
 - Data Level Parallelism (DLP) – data that can be processed in the same time
 - Task Level Parallelism (TLP) – independent tasks
- Parallelism classes
 - Instruction Level Parallelism (ILP) – exploits data level parallelism
 - Pipelining, Speculative execution
 - Thread Level Parallelism – exploits DLP & TLP in a hardware model that permits interaction between parallel threads
 - Request Level Parallelism – exploits TLP in de-coupled tasks, specified by the programmer or the OS
- Parallel Architectures
 - Uni-processor systems
 - Multi-processors systems – Multi-Core CPUs
 - Vector Architectures and GPUs – exploits DLP by applying a single instruction to a collection of data, in parallel



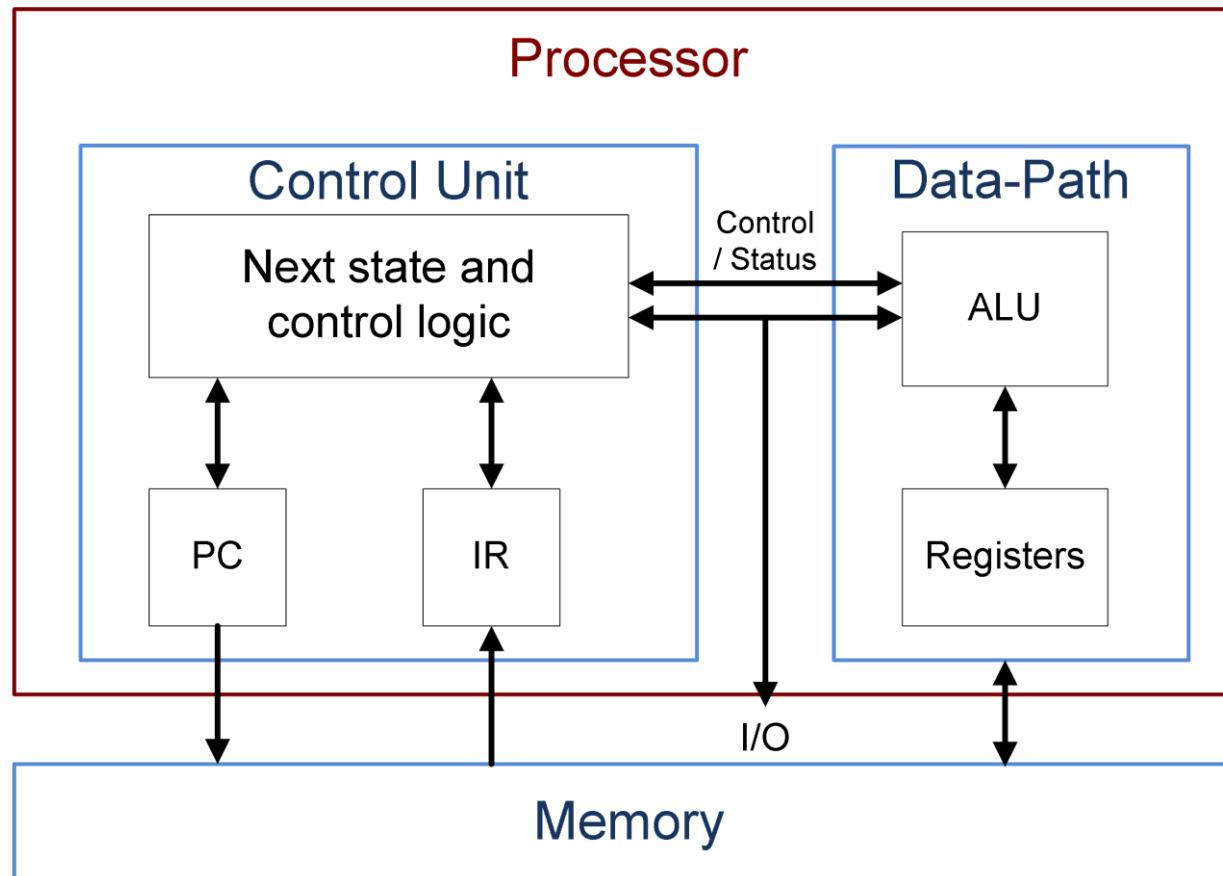
Flynn's Taxonomy

- Simple classification of multi-processing architectures – 1966

| Flynn's Taxonomy | | |
|------------------|--------------------|----------------------|
| | Single Instruction | Multiple Instruction |
| Single Data | <u>SISD</u> | <u>MISD</u> |
| Multiple Data | <u>SIMD</u> | <u>MIMD</u> |

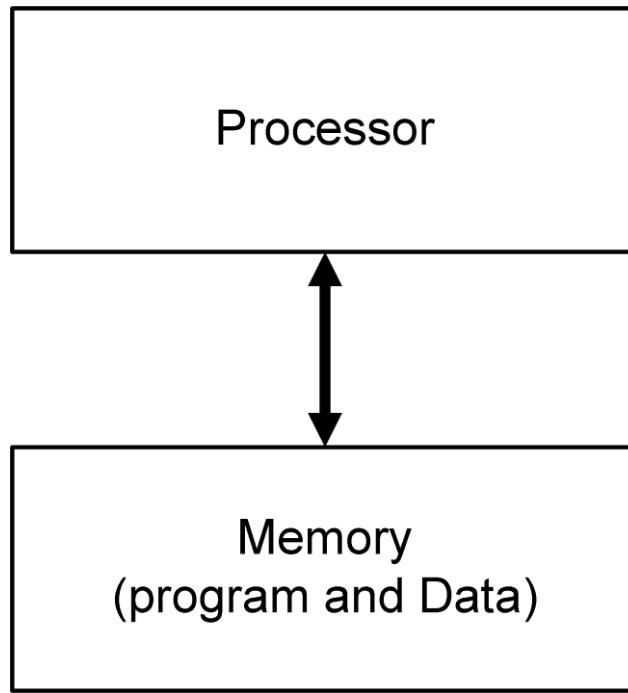
- SISD – Conventional uni-processor systems, can exploit ILP
- SIMD – The same instruction is executed by many processors on different data: Vector Architectures
- MISD – very rare, offers the advantage of redundancy
- MIMD – every processor operates on its own data and instructions, exploits task level parallelism
- **A system with N cores is effective when it runs N or more threads concurrently!**

General Processor Architecture



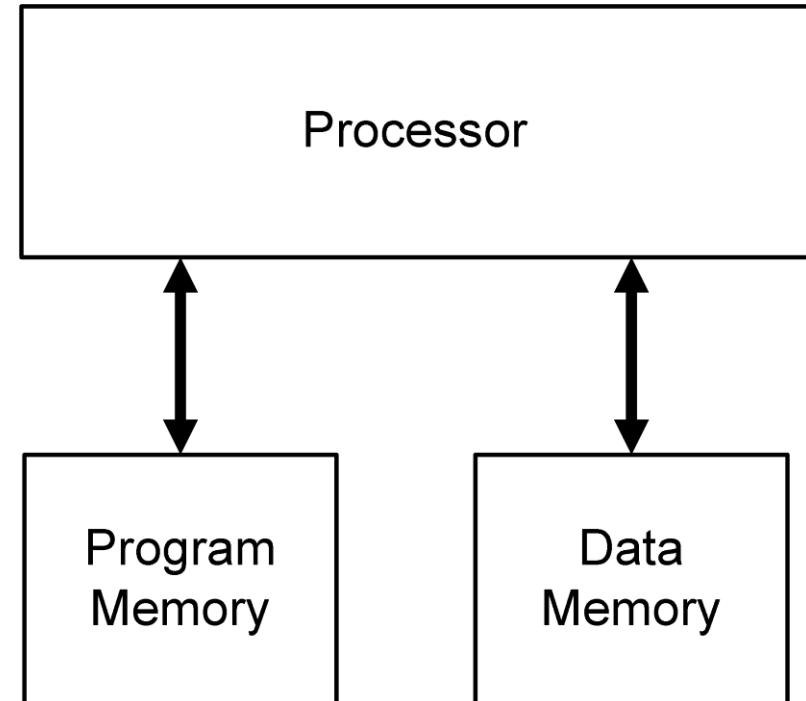


Uni-processor Classical Architectures



Van Neumann /
Princeton Architecture

A single memory for both Instruction and Data
Stored program computer



Harvard Architecture

Separate memories for Instruction and Data



- CPU types
 - Complex Instruction Set Computer (CISC)
 - Complex set of instructions, hard to pipeline, reduced number of registers, ALU operations with memory
 - Memory accesses through many different instructions
 - Many addressing modes
 - Instructions have variable width
 - Reduced Instruction Set Computer (RISC)
 - Reduced set of instructions, easy to pipeline, larger number of registers, ALU operations only with registers
 - Memory accesses only through load / store instructions
 - Reduced number of addressing modes
 - Instructions have fixed width
- Other architectures
 - DSP – digital signal processors
 - Embedded – SoC (system on chip)
 - Reconfigurable – FPGA (field programmable gate arrays)



- The interface between Hardware and low-level Software
- Core ISA elements
 - Memory models (alignment, linear, split address space)
 - Registers (special, general, mixed, kernel), Register model
 - Data types (numeric, non-numeric)
 - Instruction (format, size, types, and set)
 - Operations provided in the instruction set
 - Number of operands for each instruction, type and size of operands
 - Address specification (registers, implicit, ACC, stack)
 - Addressing modes (immediate, direct, register, indexed, stack,...)
 - Flow of Control
 - Input/Output, Interrupts
 - ...



Instruction Set Architecture



- ISA design issues
 - Which operation and data types should be supported?
 - Operands: how many, how big?
 - Where do operands reside?
 - How many registers?
 - How important are immediates and how big are they?
 - Which addressing modes dominate usage?
 - How are memory addresses computed?
 - Which control instructions should be supported?
 - How big a branch displacement is needed?
 - How should the instruction format be like, which bits designate what?
 - Instruction length: are all instructions the same length?
 - Can you add contents of memory to a register?
 - ...



Instruction Set Architecture



- ISA Classes
 - Most modern ISAs are general purpose register (GPR). ALU operands are registers or memory locations
 - 2 types
 - Register-Memory ISA: x86, x64. ALU operations: reg-reg or reg-mem
 - Register-Register, Load/Store ISA: ARM, MIPS. ALU operations: reg-reg, only Load and Store instructions access memory
- Memory addressing
 - 80x86, ARM, MIPS use byte addressing
 - ARM, MIPS – instructions must be aligned in memory
 - To access an **s-byte** object at address **A** is aligned if **A mod s = 0**
 - 80x86 does not require memory alignment, but the access is faster to aligned operands



ISA – Addressing Modes

| Addressing mode | Example Instruction | Meaning |
|--------------------|---------------------|---|
| Register | Add R4, R3 | $\text{Regs[R4]} \leftarrow \text{Regs[R4]} + \text{Regs[R3]}$ |
| Immediate | Add R4, #3 | $\text{Regs[R4]} \leftarrow \text{Regs[R4]} + 3$ |
| Displacement | Add R4, 100(R1) | $\text{Regs[R4]} \leftarrow \text{Regs[R4]} + \text{Mem[100 + Regs[R1]]}$ |
| Register Indirect | Add R4, (R1) | $\text{Regs[R4]} \leftarrow \text{Regs[R4]} + \text{Mem[Regs[R1]]}$ |
| Indexed | Add R4, (R1+R2) | $\text{Regs[R4]} \leftarrow \text{Regs[R4]} + \text{Mem[Regs[R1] + Regs[R2]]}$ |
| Direct or Absolute | Add R4, (1001) | $\text{Regs[R4]} \leftarrow \text{Regs[R4]} + \text{Mem[1001]}$ |
| Memory indirect | Add R4, @(R3) | $\text{Regs[R4]} \leftarrow \text{Regs[R4]} + \text{Mem[Regs[R3]]}$ |
| Auto-increment | Add R4, (R3)+ | $\text{Regs[R4]} \leftarrow \text{Regs[R4]} + \text{Mem[Regs[R3]]}$ $\text{Regs[R3]} \leftarrow \text{Regs[R3]} + d \text{ (size of element)}$ |
| Auto-decrement | Add R4, -(R3) | $\text{Regs[R3]} \leftarrow \text{Regs[R3]} - d \text{ (size of element)}$ $\text{Regs[R4]} \leftarrow \text{Regs[R4]} + \text{Mem[Regs[R3]]}$ |
| Scaled | Add R4, 100(R2)[R3] | $\text{Regs[R4]} \leftarrow \text{Regs[R4]} + \text{Mem[100 + Regs[R2] + Regs[R3] * d]}$ |



Instruction Set Architecture



- Endianness



Bytes in register

| | | | | |
|---------|------|------|------|------|
| Address | 0003 | 0002 | 0001 | 0000 |
| Byte # | 3 | 2 | 1 | 0 |

Little Endian
LSB byte at lower address

| | | | | |
|---------|------|------|------|------|
| Address | 0003 | 0002 | 0001 | 0000 |
| Byte # | 0 | 1 | 2 | 3 |

Big Endian
MSB byte at lower address

- Type and dimension of operands

- 80x86, ARM, MIPS support:
 - 8-bit (ASCII character)
 - 16-bit (Unicode character or half word)
 - 32-bit (integer or word)
 - 64-bit (double word or long integer)
 - IEEE 754 floating point: 32-bit (single precision) and 64-bit (double precision)
- 80x86 also supports 80-bit floating point (extended double precision)



Instruction Set Architecture



- Instruction operations

| Operation type | Examples |
|------------------------|--|
| Arithmetic and logical | Integer arithmetic and logical operations: add, sub, and, or, multiply, divide |
| Data transfer | Load, stores, move instructions (on computers with memory addressing) |
| Control | Branch, jump, procedure call and return, traps |
| System | Operating system call, virtual memory management instructions |
| Floating Point | Floating-point operations: add, multiply, divide, compare |
| Decimal | Decimal add, multiply, decimal to character conversion |
| String | String move, compare, search |
| Graphics | Pixel and vertex operations, compression/decompression operations |



Instruction Set Architecture



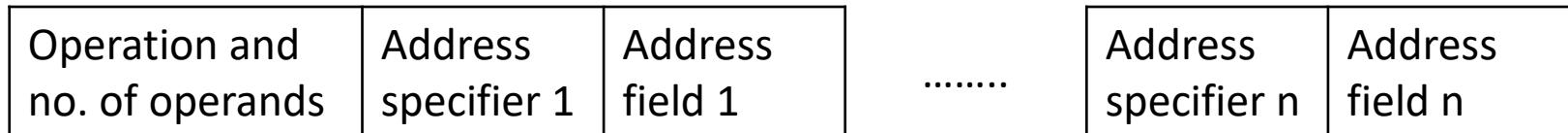
- Flow Control Instructions
 - Conditional jumps, unconditional jumps, procedure calls and returns
 - PC relative addressing: next address is an offset added to the PC
 - MIPS (BEQ, BNE, etc.): test the content of a register
 - 80x86, ARM: test the bits of the FLAG register that are affected by arithmetic / logic operations
 - ARM, MIPS procedure call: sets the return address in a register
 - 80x86 procedure call: sets the return address in memory or stack
- Instruction formats – 2 main types: fixed and variable length
 - ARM, MIPS: 32-bit instructions, simple decoding
 - 80x86: variable length instructions (1 – 18 bytes)
 - Variable length instructions occupy less space
 - The number of registers and used addressing modes influence instruction length
 - ARM, MIPS extensions: 16-bit instructions Thumb and MIPS16



Instruction Set Architecture



- Variable (Intel 80x86, VAX)



- Fixed (Alpha, ARM, MIPS, PowerPC, SPARC)

| | | | |
|-----------|-----------------|-----------------|-----------------|
| Operation | Address field 1 | Address field 2 | Address field 3 |
|-----------|-----------------|-----------------|-----------------|

- Hybrid (IBM 360/370, MIPS16, Thumb)

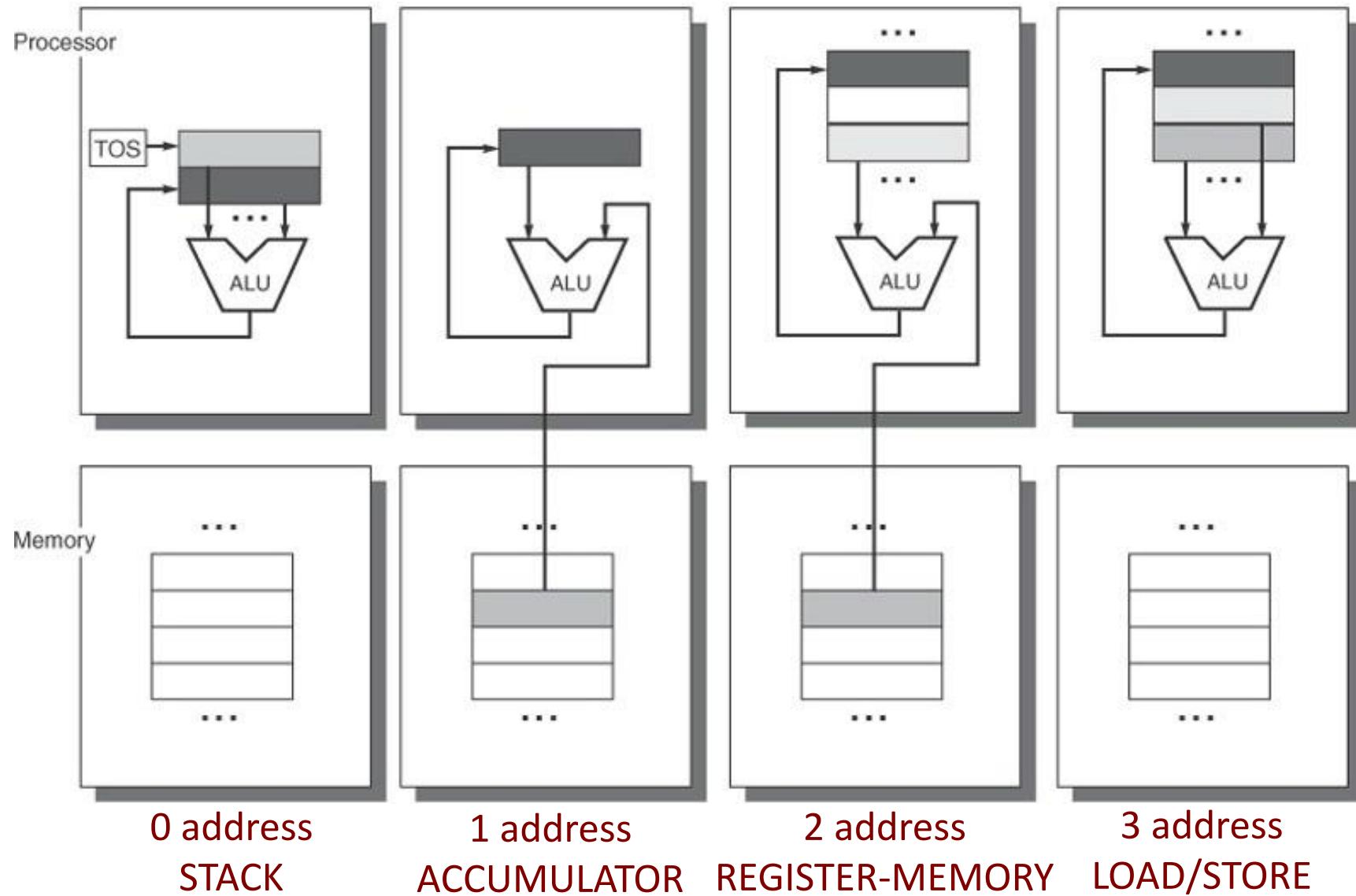
| | | |
|-----------|-------------------|---------------|
| Operation | Address specifier | Address field |
|-----------|-------------------|---------------|

| | | | |
|-----------|---------------------|---------------------|---------------|
| Operation | Address specifier 1 | Address specifier 2 | Address field |
|-----------|---------------------|---------------------|---------------|

| | | | |
|-----------|-------------------|-----------------|-----------------|
| Operation | Address specifier | Address field 1 | Address field 2 |
|-----------|-------------------|-----------------|-----------------|



Basic ISA Classes

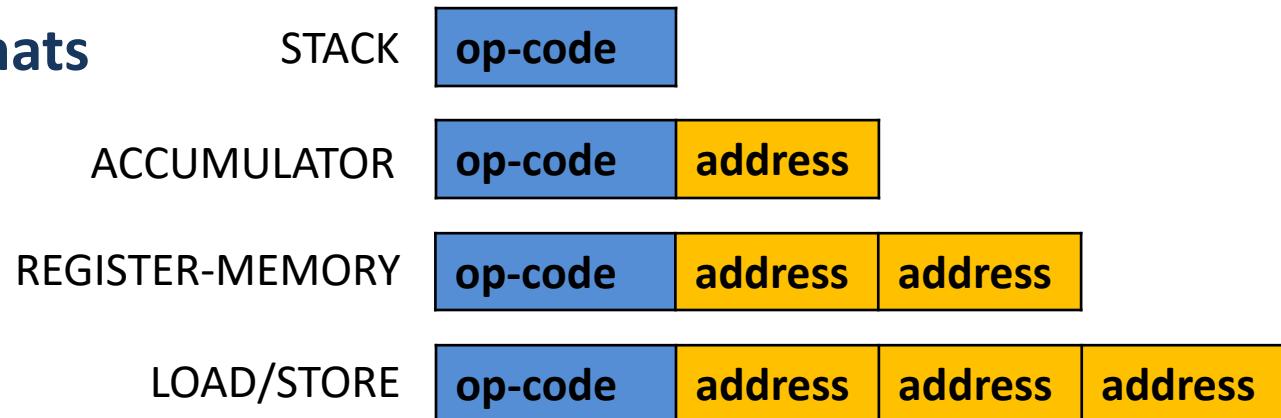




Basic ISA Classes



Instruction formats



| STACK | ACCUMULATOR | REGISTER-MEMORY | LOAD/STORE |
|--------|-------------|-----------------|----------------|
| Push A | Load A | Load R1, A | Load R1, A |
| Push B | Add B | Add R1, B | Load R2, B |
| Add | Store C | Store R1, C | Add R3, R2, R1 |
| Pop C | | | Store R3, C |

Assembly for $C = A + B$. Operands A, B, C are in memory

The add instruction has implicit operands for stack and ACC, explicit for GPR



Basic ISA Classes

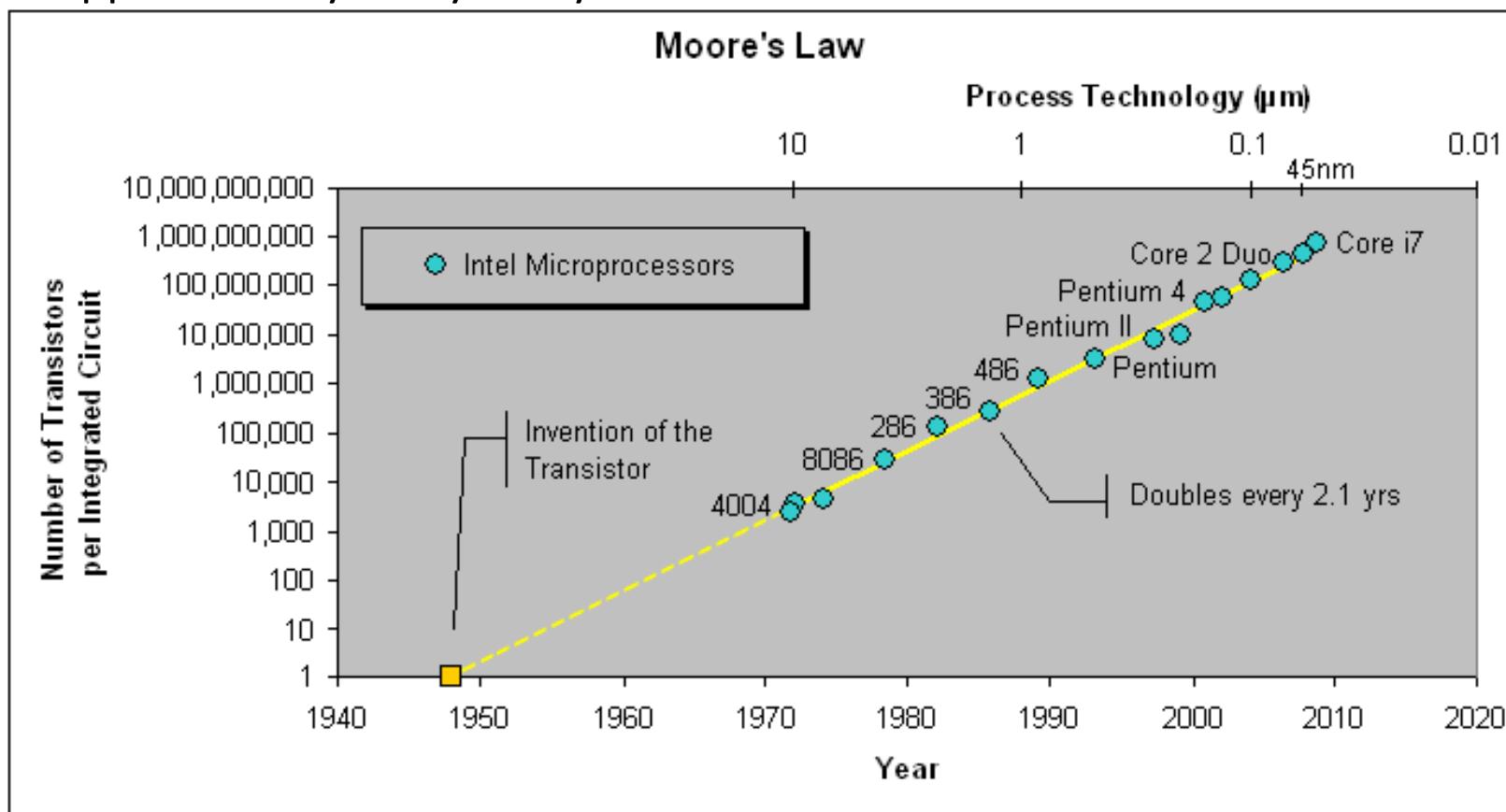


- Location of operands
 - STACK (0 Address)
 - both operands are implicit TOS (top of stack) and SOS (second on stack)
 - the result goes to TOS
 - Special instructions for memory transfers: PUSH and POP
 - ACCUMULATOR (1 address)
 - one operand is the accumulator register
 - the other operand is given explicit
 - REGISTER-MEMORY (2 address)
 - the operands are registers or memory locations
 - the result is one of the source registers
 - LOAD/STORE (3 address)
 - all operands are registers
 - special instructions for accessing memory locations (load and store)



Technology – Moore's Law

- Moore's Law
 - Gordon Moore (1965): the number of transistors on a chip will double approximately every two years.



<http://forums.anandtech.com/showthread.php?t=2173027&page=2>



Technology – Power Consumption



- **Dynamic Power (Watts)**

- in CMOS chips (switching transistors)

$$Power_{dynamic} = \frac{1}{2} \times CapacitiveLoad \times Voltage^2 \times FrequencySwitched$$

- Slowing clock rate for a task reduces power consumption
 - Dynamic Power can be reduced by lowering the voltage
 - Voltages dropped from 5V to almost 1V in 20 years
 - Microprocessors stop the clock for inactive modules → energy saving
- **Static Power (Watts)**
 - Important due to leakage current (even if the transistor is inactive)

$$Power_{static} = Current_{static} \times Voltage$$

- Proportional to the number of devices on a chip
 - Leak current increases as transistor size decreases



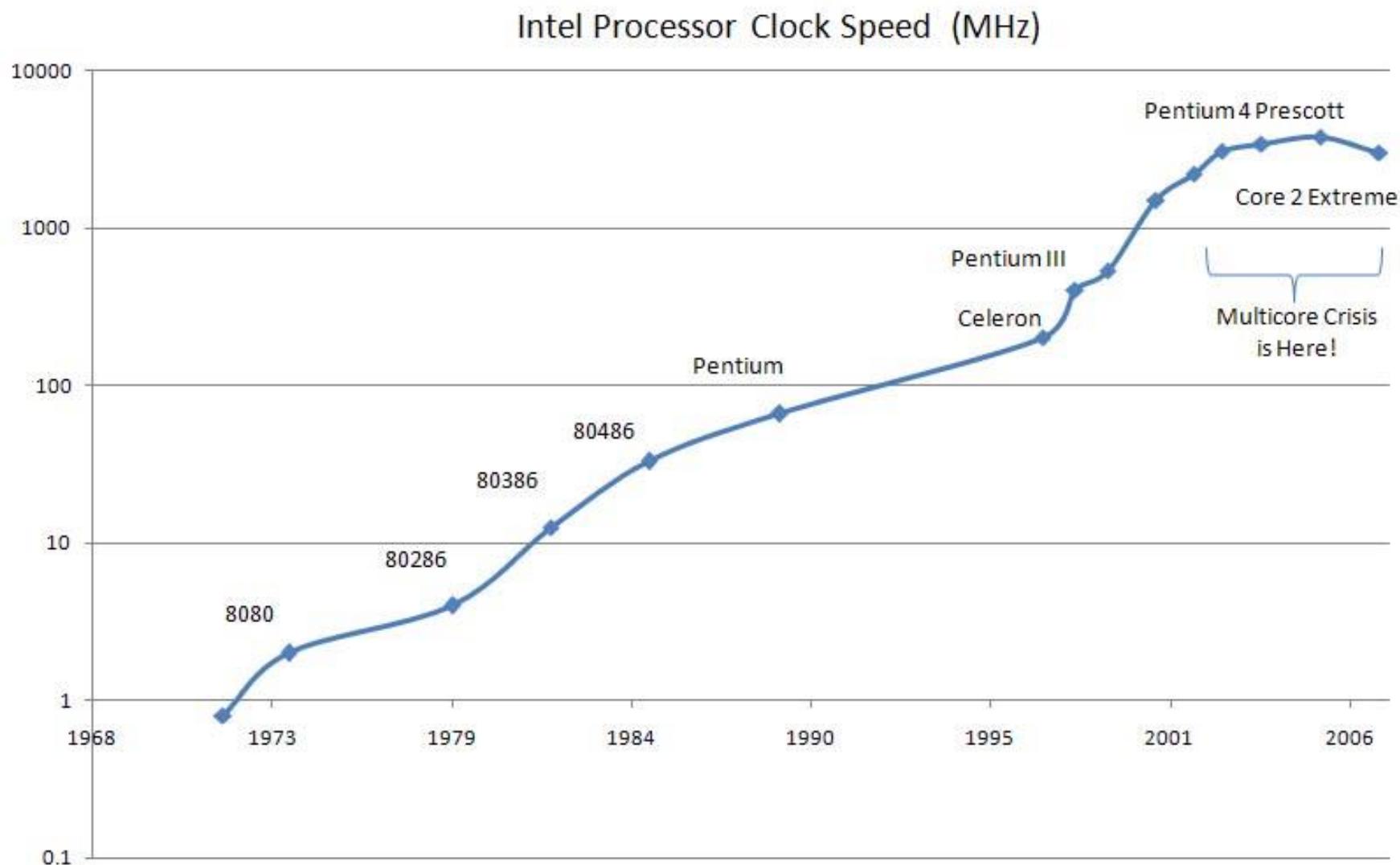
Technology – Power Consumption



- Systems with reduce power consumption
 - Temperature diodes to reduce activity if the chip get's to hot
 - Reduce voltage and clock frequency or the issue rate of instructions
- In 2011, the target for leaks – 25% of the total power consumption
- First 32-bit microprocessors (Intel 80386) ~ 2 Watts
- Now, 3.3 GHz Intel Core i7 ~ 130 Watts
 - The heat from a chip (1.5 cm) must be dissipated → reach the limits of what can be cooled by air
- Design for power:
 - Sleep modes
 - Partially or totally reduce the clock frequency
 - Maximum operating temperatures → Low
 - The limits of air cooling have led to multiple processors on a chip running at lower voltages and clock rates

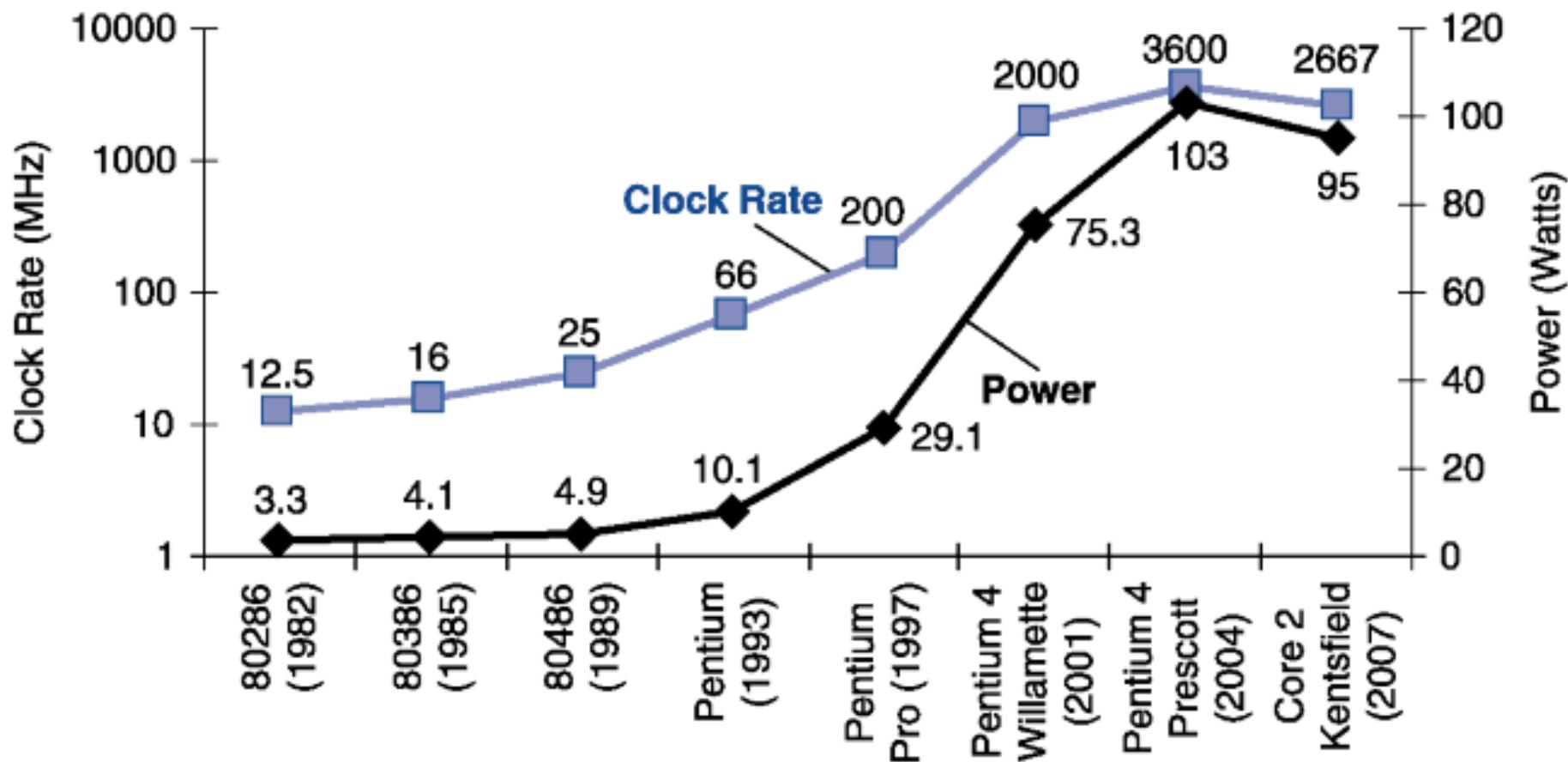


Clock Frequency Evolution





Frequency vs. Power Consumption



http://www.edwardbosworth.com/My5155_Slides/Chapter01/ThePowerWall.htm



Computer Performance – Metrics



- Bandwidth over Latency
 - Bandwidth or throughput
 - Total amount of work in a given time
 - Number of tasks completed per unit time
 - Important when we run several tasks
 - Latency or execution time or response time (delay)
 - The time period to complete a task
 - Important if we have to run a time critical task
- Processor Performance Equation
 - IC – instruction count
 - CPI – average number of clock cycles per instruction
 - CCT – clock cycle time

$$CPU_{time} = \frac{CPU \text{ clock cycles for a program}}{Clock \text{ rate}} \quad CPI = \frac{CPU \text{ clock cycles for a program}}{Instruction \text{ count}}$$

$$CPU_{time} = IC \cdot CPI \cdot CCT = \frac{Instructions}{Program} \cdot \frac{Cycles}{Instruction} \cdot \frac{Seconds}{Cycle} = \frac{Seconds}{Program}$$



Computer Performance – Metrics



- Computer Performance depends on
 - CCT → hardware and organization
 - CPI → organization and ISA
 - IC → ISA and compiler
- ISA influences the three components of computer performance
- Performance equation

$$\text{Performance}_x = \frac{1}{\text{Execution time}_x}$$

- Running speed of a program: MIPS (millions instructions per second)

$$MIPS = \frac{\text{Instruction count}}{\text{Execution time} \times 10^6} = \frac{\text{Instruction count}}{\frac{\text{Instruction count} \times \text{CPI}}{\text{Clock rate}} \times 10^6} = \frac{\text{Clock rate}}{\text{CPI} \times 10^6}$$



Amdahl's Law



- “the performance improvement to be gained from using some faster mode of execution is limited by the fraction of the time the faster mode can be used”
- Speedup

$$\text{Speedup} = \frac{\text{Performance for entire task using the enhancement when possible}}{\text{Performance for entire task without using the enhancement}}$$

$$\text{Speedup} = \frac{\text{Execution time for entire task without using the enhancement}}{\text{Execution time for entire task using the enhancement when possible}}$$

- Speedup depends on 2 factors:
 - The fraction of time that can benefit from enhancement $\text{Fraction}_{\text{enhanced}} = f_x$
 - The gain obtained by using the enhancement $\text{Speedup}_{\text{enhanced}} = S_x$

$$\text{Execution time}_{\text{new}} = \text{Execution time}_{\text{old}} \times \left((1 - f_x) + \frac{f_x}{S_x} \right)$$

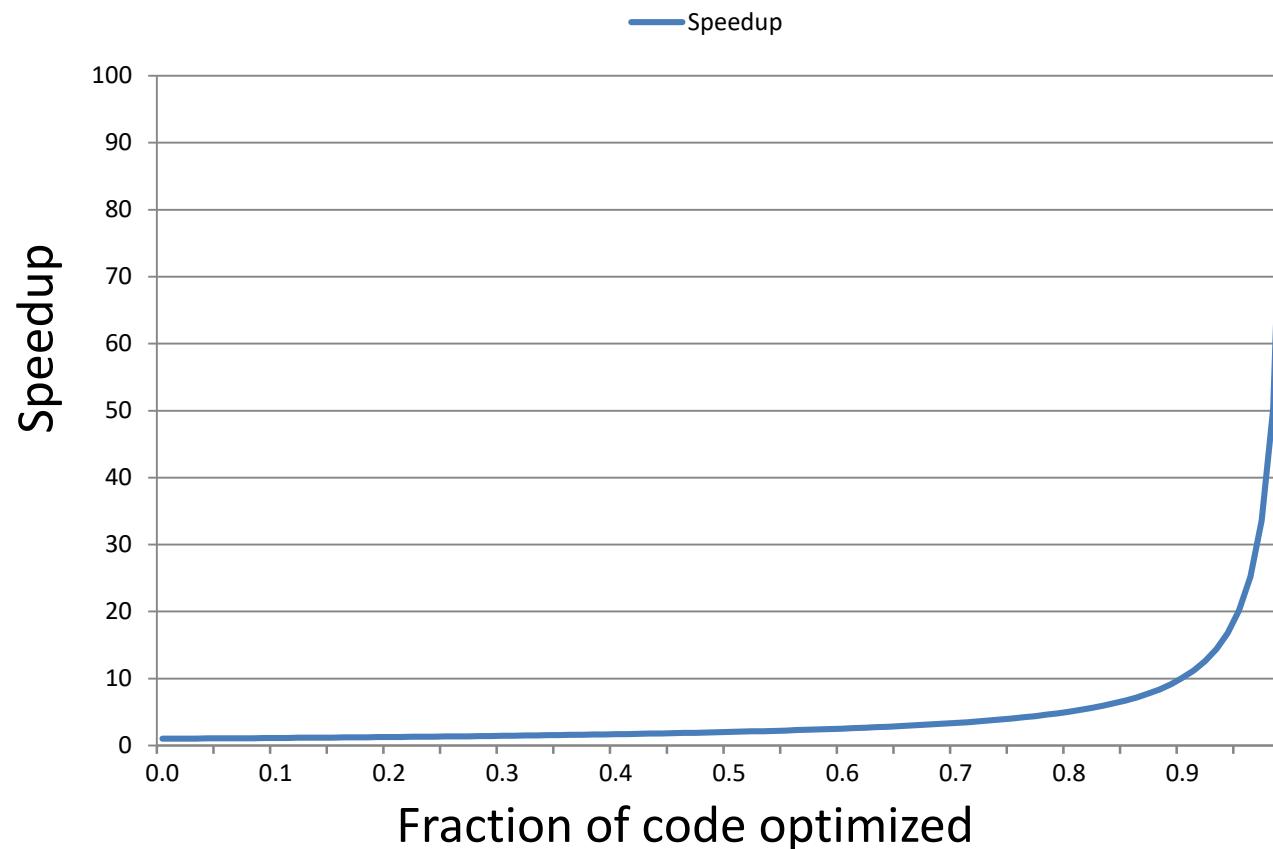


Amdahl's Law



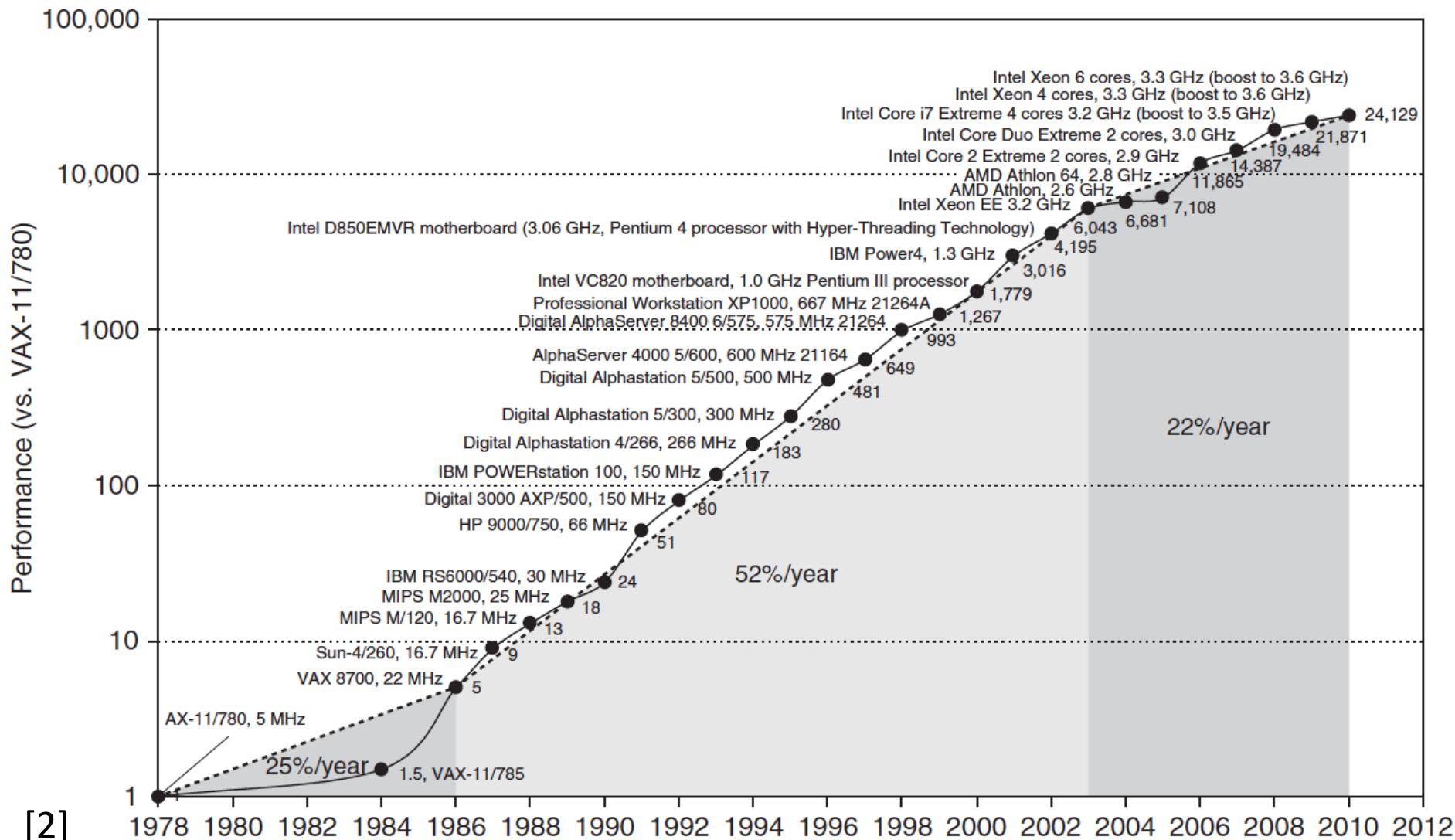
$$Speedup_{overall} = \frac{\text{Execution time}_{old}}{\text{Execution time}_{new}} = \frac{1}{(1 - f_x) + \frac{f_x}{S_x}}$$

If $S_x=100$, what is the overall speedup as a function of f_x





Computer Performance – SPEC benchmarks





Computer Performance – History Table



| Microprocessor | 16-bit address/ bus, microcoded | 32-bit address/ bus, microcoded | 5-stage pipeline, on-chip I & D caches, FPU | 2-way superscalar, 64-bit bus | Out-of-order 3-way superscalar | Out-of-order superpipelined, on-chip L2 cache | Multicore OOO 4-way on chip L3 cache, Turbo |
|-----------------------------|---------------------------------------|---------------------------------------|---|----------------------------------|-----------------------------------|--|--|
| Product | Intel 80286 | Intel 80386 | Intel 80486 | Intel Pentium | Intel Pentium Pro | Intel Pentium 4 | Intel Core i7 |
| Year | 1982 | 1985 | 1989 | 1993 | 1997 | 2001 | 2010 |
| Die size (mm ²) | 47 | 43 | 81 | 90 | 308 | 217 | 240 |
| Transistors | 134,000 | 275,000 | 1,200,000 | 3,100,000 | 5,500,000 | 42,000,000 | 1,170,000,000 |
| Processors/chip | 1 | 1 | 1 | 1 | 1 | 1 | 4 |
| Pins | 68 | 132 | 168 | 273 | 387 | 423 | 1366 |
| Latency (clocks) | 6 | 5 | 5 | 5 | 10 | 22 | 14 |
| Bus width (bits) | 16 | 32 | 32 | 64 | 64 | 64 | 196 |
| Clock rate (MHz) | 12.5 | 16 | 25 | 66 | 200 | 1500 | 3333 |
| Bandwidth (MIPS) | 2 | 6 | 25 | 132 | 600 | 4500 | 50,000 |
| Latency (ns) | 320 | 313 | 200 | 76 | 50 | 15 | 4 |
| Memory module | DRAM | Page mode DRAM | Fast page mode DRAM | Fast page mode DRAM | Synchronous DRAM | Double data rate SDRAM | DDR3 SDRAM |
| Module width (bits) | 16 | 16 | 32 | 64 | 64 | 64 | 64 |
| Year | 1980 | 1983 | 1986 | 1993 | 1997 | 2000 | 2010 |
| Mbits/DRAM chip | 0.06 | 0.25 | 1 | 16 | 64 | 256 | 2048 |
| Die size (mm ²) | 35 | 45 | 70 | 130 | 170 | 204 | 50 |
| Pins/DRAM chip | 16 | 16 | 18 | 20 | 54 | 66 | 134 |
| Bandwidth (MBytes/s) | 13 | 40 | 160 | 267 | 640 | 1600 | 16,000 |
| Latency (ns) | 225 | 170 | 125 | 75 | 62 | 52 | 37 |
| Local area network | Ethernet | Fast Ethernet | Gigabit Ethernet | 10 Gigabit Ethernet | 100 Gigabit Ethernet | | |
| IEEE standard | 802.3 | 803.3u | 802.3ab | 802.3ac | 802.3ba | | |
| Year | 1978 | 1995 | 1999 | 2003 | 2010 | | |
| Bandwidth (Mbits/sec) | 10 | 100 | 1000 | 10,000 | 100,000 | | |
| Latency (usec) | 3000 | 500 | 340 | 190 | 100 | | |
| Hard disk | 3600 RPM | 5400 RPM | 7200 RPM | 10,000 RPM | 15,000 RPM | 15,000 RPM | |
| Product | CDC WrenI 94145-36 | Seagate ST41600 | Seagate ST15150 | Seagate ST39102 | Seagate ST373453 | Seagate ST3600057 | |
| Year | 1983 | 1990 | 1994 | 1998 | 2003 | 2010 | |
| Capacity (GB) | 0.03 | 1.4 | 4.3 | 9.1 | 73.4 | 600 | |
| Disk form factor | 5.25 inch | 5.25 inch | 3.5 inch | 3.5 inch | 3.5 inch | 3.5 inch | |
| Media diameter | 5.25 inch | 5.25 inch | 3.5 inch | 3.0 inch | 2.5 inch | 2.5 inch | |
| Interface | ST-412 | SCSI | SCSI | SCSI | SCSI | SAS | |
| Bandwidth (MBytes/s) | 0.6 | 4 | 9 | 24 | 86 | 204 | |
| Latency (ms) | 48.3 | 17.1 | 12.7 | 8.8 | 5.7 | 3.6 | |

[2]



Conclusions



- In 2004 Intel has canceled its uni-processor projects and has declared, together with IBM and SUN, that higher performances can be obtained by using more processors on a chip instead of making uni-processor systems more faster
- This is a historical turnaround from instruction level parallelism to thread and data level parallelism
- The compiler and the hardware exploit ILP implicitly
- For exploiting TLP and DLP the programmer is involved in developing faster codes
- Next: Multiprocessors, Multi-cores, Many-cores, etc.
- Processor market 2010:
 - 1.8 billion PMDs (90% cell phones), 350 mil. desktop PCs, 20 mil. servers
 - 19 billion embedded processors
 - ARM (RISC) ~ 6.1 billion caps, ~ 20 times more than x86



Problems – Homework



- Write a program using instructions defined by you for the 0, 1, 2 and 3 addresses processors to implement the following expression: $e = a \cdot b \cdot c + d$. The operands a, b, c, d and the result e are memory locations.
- For the 0, 1, 2 and 3 addresses machines write a program to evaluate the following expression: $e = a \cdot b + c \cdot d$.
- Describe the differences between big endian and little endian.
- ...



References



1. D. A. Patterson, J. L. Hennessy, “Computer Organization and Design: The Hardware/Software Interface”, 5th edition, ed. Morgan–Kaufmann, 2013
2. D. A. Patterson and J. L. Hennessy, “Computer Organization and Design: A Quantitative Approach”, 5th edition, ed. Morgan-Kaufmann, 2011
3. D. M. Harris, S. L. Harris, *Digital Design and Computer Architecture*, Morgan Kaufmann, San Francisco, 2007
4. D. A. Patterson, J. L. Hennessy, “ORGANIZAREA SI PROIECTAREA CALCULATOARELOR. INTERFATA HARDWARE/SOFTWARE”, Editura ALL, Romania, ISBN: 973-684-444-7
- ...



VHDL – Remember



- Types of Circuits
 - Combinational Circuits
 - Sequential circuits
- Basic building blocks
 - Logic Gates
 - Multiplexers
 - Decoders
 - D-Latches and D-Flip-Flops
 - Counters
 - Memories



VHDL – To Remember



- **Rules of VHDL coding!!!**
 - Not EVERYTHING is a component. Do not create components for basic building blocks like: logic gates, latches, flip-flops, tri-state buffers, counters, decoders, etc.
 - Do not abuse of structural design at the logic gate granularity!
 - You will generally use the behavioral type of describing your design.
 - You will create a new component **only when a part of your design has meaning** (or when the TA explicitly tells you to do so).



VHDL – Remember



- 1-bit signal declaration

```
signal sig_name : std_logic := '0';
```

- N-bit signal declaration

```
signal sig_name: std_logic_vector(N-1 downto 0) := "00....0";
```

- Initialization

16-bit signal "0000000000000000";

16-bit signal x"0000";

16-bit signal (others => '0');



VHDL – Remember



- Logic Gates – A & B – inputs, O – output



NOT



AND



OR

$O \leqslant \text{not } A;$

$O \leqslant A \text{ and } B;$

$O \leqslant A \text{ or } B;$



NAND



NOR



XOR

$O \leqslant A \text{ nand } B;$

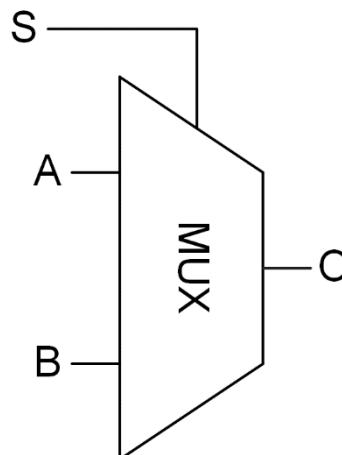
$O \leqslant A \text{ nor } B;$

$O \leqslant A \text{ xor } B;$

- Do not declare an entity, only signals if needed!



- 2:1 Multiplexer

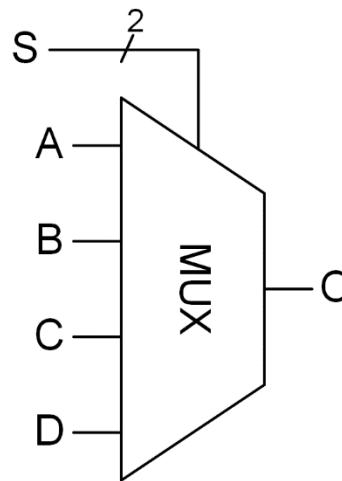


Do not declare an entity,
only signals if needed!

$O \leq A$ when $S = '0'$ else B ;

```
process(S, A, B)
begin
  if(S = '0') then
    O <= A;
  else
    O <= B;
  end if;
end process;
```

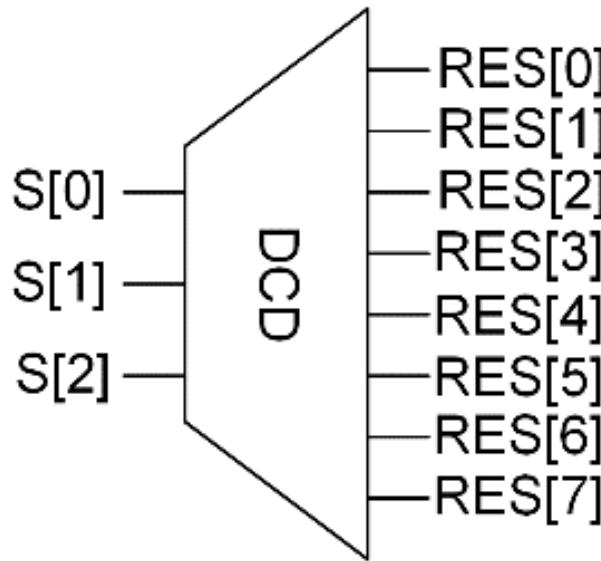
- 4:1 Multiplexer



```
process(S, A, B, C, D)
begin
  case S is
    when "00" => O <= A;
    when "01" => O <= B;
    when "10" => O <= C;
    when others => O <= D;
  end case;
end process;
```



- 3:8 Decoder



```
process(S)
begin
    case S is
        when "000" => RES <= "00000001";
        when "001" => RES <= "00000010";
        when "010" => RES <= "00000100";
        when "011" => RES <= "00001000";
        when "100" => RES <= "00010000";
        when "101" => RES <= "00100000";
        when "110" => RES <= "01000000";
        when others => RES <= "10000000";
    end case;
end process;
```

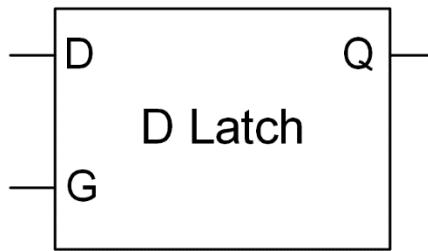
- Do not declare an entity, only signals if needed!



VHDL – Remember

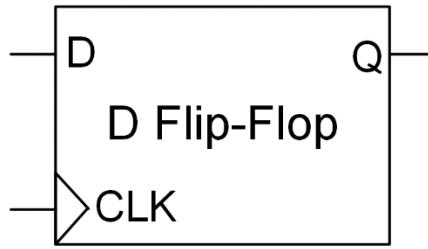


- D-Latch



```
process(G, D)
begin
  if(G = '1') then
    Q <= D;
  end if;
end process;
```

- D-Flip-Flop



```
process(clk)
begin
  if rising_edge(clk) then
    Q <= D;
  end if;
end process;
```

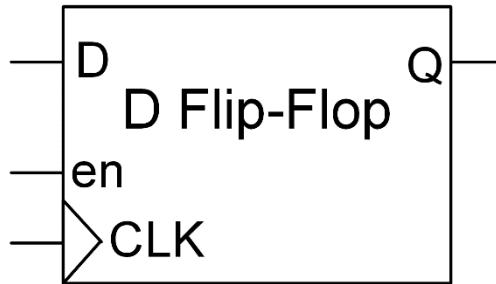
- Do not declare an entity, only signals if needed!



VHDL – Remember



- D-Flip-Flop with enable

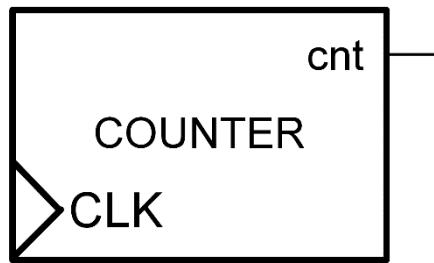


```
process(clk, en)
begin
    if rising_edge(clk) then
        if en = '1' then
            Q <= D;
        end if;
    end if;
end process;
```

- **rising_edge(clk)** is equivalent to **clk'event** and **clk = '1'** but shorter
- NEVER use **rising_edge(clk)** and **en = '1'** a.k.a. Gated Clock!
- Do not declare an entity, only signals if needed!

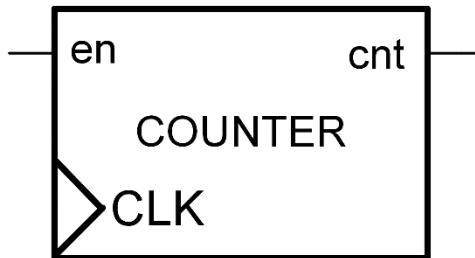


- Up Counter



```
process(clk)
begin
    if rising_edge(clk) then
        cnt <= cnt + 1;
    end if;
end process;
```

- Up Counter with enable signal



```
process(clk, en)
begin
    if rising_edge(clk) then
        if en = '1' then
            cnt <= cnt + 1;
        end if;
    end if;
end process;
```

- Do not declare an entity, only signals if needed!



Computer Architecture

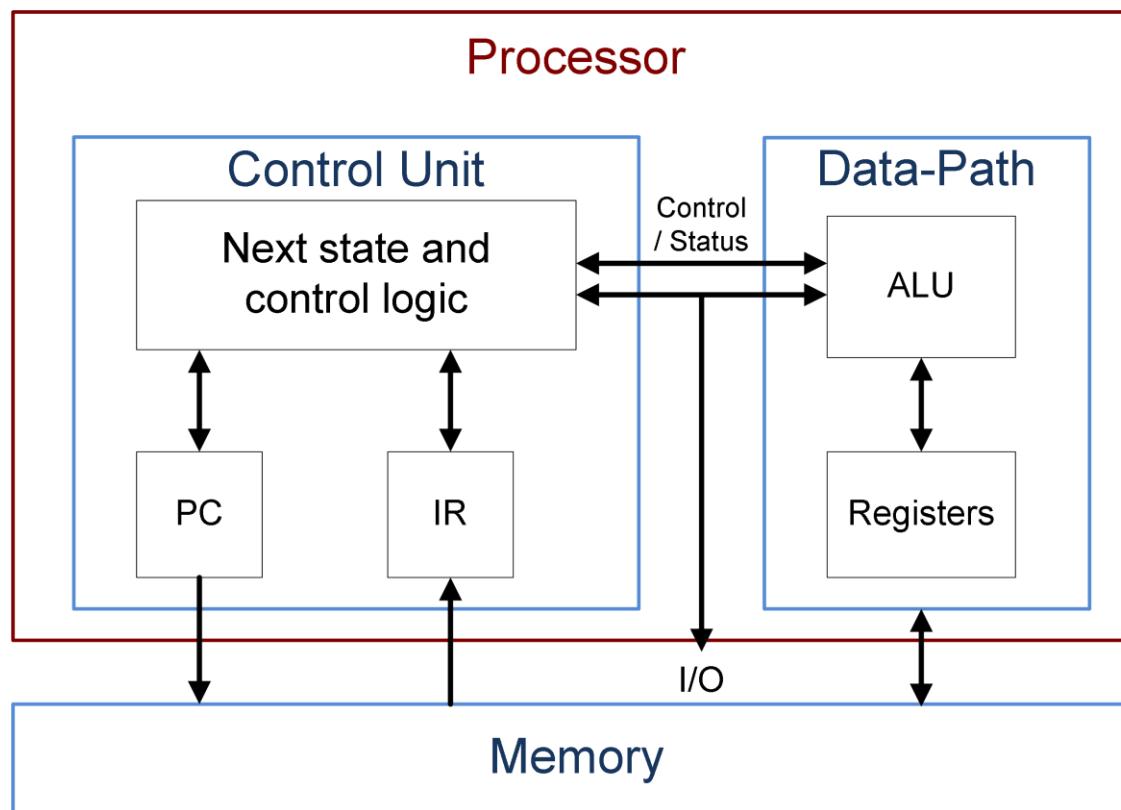
Lecturer: Mihai Negru

2nd Year, Computer Science

Lecture 2: High Level Synthesis

<http://users.utcluj.ro/~negrum/>

- Digital System
 - Data-Path – registers, busses, processing logic
 - Control Unit – determines the sequence of data-processing operations of the Data-Path





Register Transfer Level (RTL)



- A system is described at RTL level by the transfer of information between the memory elements of the system
$$\text{RT operation: } \text{Rdest} \leftarrow f(\text{Rsrc}_1, \text{Rsrc}_2, \dots, \text{Rsrc}_n)$$
- Abstract RTL – a behavioral specification
 - Does not take into account the structure of the digital system
 - Not related to timing or resources
- Physical or Concrete RTL – provides an implementation of the behavioral specification based on a selected structure at clock period granularity
 - Related to resources and timing constraints
- RTL Design
 - Converts a behavioral specification (Abstract RTL) into a structural description (Concrete RTL)



Register Transfer Level (RTL)



- A digital system specified at Concrete RTL level includes the following 3 components:
 - The set of registers/memories in the system.
 - The functional units, which perform the required operations.
 - The control that supervises the sequence of operations in the system.
- **Register Transfer Level (RTL) / Register Transfer Notation (RTN)**
 - RTL represents an algebraic notation used to define machine-level operations
 - Not executed by a computer – used to explain/describe how the computer works.
 - Micro-operation: single register transfer operation

$$X \leftarrow Y + Z$$

- RTL statement: (condition) \rightarrow {micro-operation,..., micro-operation};

$$(c > 0) \rightarrow X \leftarrow Y + Z$$



Register Transfer Notation (RTN)



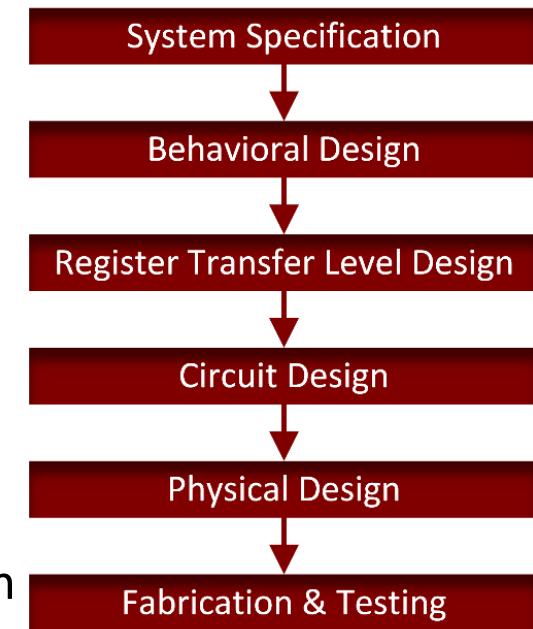
| | |
|--|--|
| \leftarrow | Assignment |
| $=, \neq$ | Tests for equality and inequality |
| $ $ | Bit string concatenation |
| $X \leftarrow Y$ | Data transfer of contents of regY to regX |
| $X \leftarrow 0$ | Clears regX |
| $X \leftarrow Y + Z$ | Adds contents of regY with regZ, load into regX |
| $X \leftarrow Y \vee Z$ | ORs contents of regY with regZ, load into regX |
| $DR \leftarrow M[AR]$ | Load into DR the contents of memory pointed to by AR |
| $R1 \leftarrow R1 >> 1$ | Register R1 one bit right shift, with 0 into left-most bit |
| $R2 \leftarrow R2 << 1$ | Register R2 one bit left shift, with 0 into right-most bit |
| $X \leftarrow Y, A \leftarrow B$ | Parallel transfers |
| $(cond) \rightarrow A \leftarrow B$ | If cond = 1 then transfer contents of regB into regA |
| $S0 \rightarrow A \leftarrow B$ | When in state S0, load regA with contents of regB |
| $P \cdot (a \cdot b) \rightarrow R2 \leftarrow R3$ | When in state P, if a AND b is true then load R2 with contents of R3; a, b are signals |



High Level Synthesis (HLS)

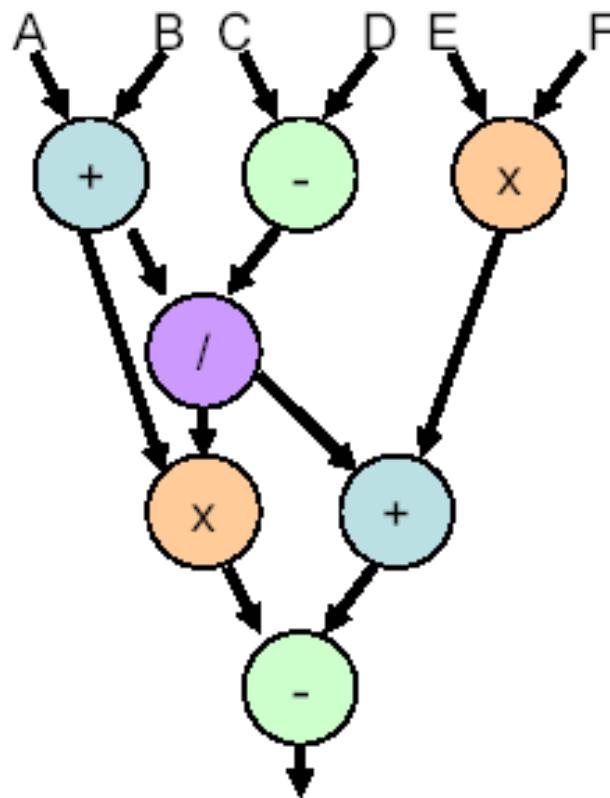


- High Level Synthesis (Structural Synthesis)
 - Starts from an abstract behavioral description
 - Generates a Concrete RTL structural description
 - Functional units (+,-,*), Memories, Interconnections
 - Obeys constraints (timing/size/performance)
- VHDL Synthesis (comparison)
 - Starts from an RTL description
 - Uses logic synthesis techniques to optimize the design
 - Generates a standard-cell net list → FPGA
- HLS basic operations
 - Resource allocation – the number and types of hardware components
 - Scheduling – assigning operations to time slots (clock cycles)
 - Module Binding – assigning operations to allocated hardware components
 - Controller Synthesis – design on control style and clocking scheme





- Data-Flow graphs (DFGs) or Data-Dependency Graphs (DDGs)
 - Represent parallelism in computation and precedency of operations
 - Nodes: represent computations
 - Edges: represent precedence relations



Data-Flow Graph – Example

Sequential Computation

$\text{tmp_1} = A+B$

$\text{tmp_2} = C-D$

$\text{tmp_3} = E*F$

$\text{tmp_4} = \text{tmp_1}/\text{tmp_2}$

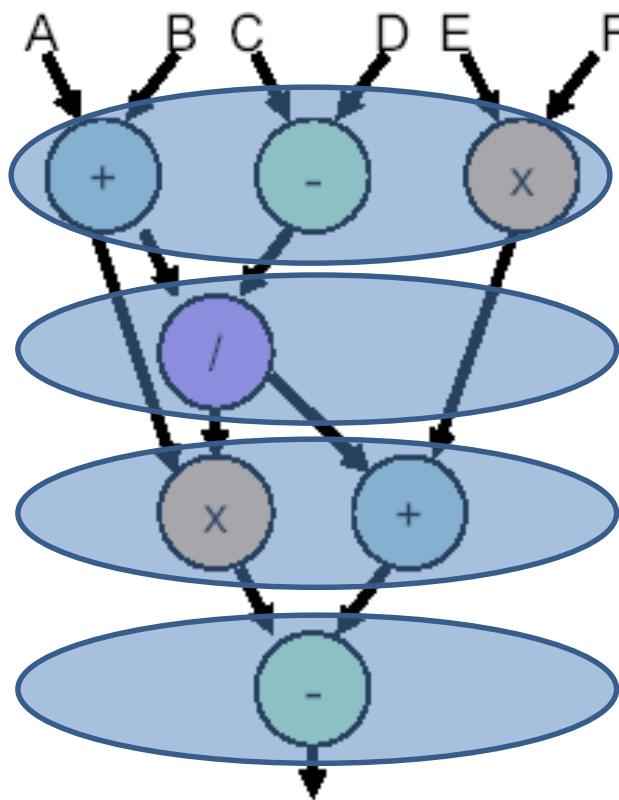
$\text{tmp_5} = \text{tmp_1}*\text{tmp_4}$

$\text{tmp_6} = \text{tmp_4}+\text{tmp_3}$

$\text{Result} = \text{tmp_5}-\text{tmp_6}$



- Data-Flow graphs (DFGs) or Data-Dependency Graphs (DDGs)
 - Represent parallelism in computation and precedency of operations
 - Nodes: represent computations
 - Edges: represent precedence relations



Data-Flow Graph – Example

Concurrent Computation

$\text{tmp_1}=A+B, \text{tmp_2}=C-D, \text{tmp_3}=E*F$

$\text{tmp_4}=\text{tmp_1}/\text{tmp_2}$

$\text{tmp_5}=\text{tmp_1}*\text{tmp_4}, \text{tmp_6}=\text{tmp_4}+\text{tmp_3}$

$\text{Result}=\text{tmp_5}-\text{tmp_6}$



Compilation / Optimization Techniques



- Behavioral optimization
 - no knowledge about the circuit implementation is required

| Tree Height Reduction | |
|--|---|
| $x = a + b + c + d;$ | can be split |
| $x = a + b; \quad x = x + c; \quad x = x + d;$ | → three additions in series. |
| $p = a + b; \quad q = c + d; \quad x = p + q;$ | → the first two additions can be done in parallel |
| Constant Propagation | Variable Propagation |
| $a = 0; b = a + 1; c = 2 \cdot b$ | $a = x; b = a + 1; c = 2 \cdot a$ |
| $a = 0; b = 1; c = 2$ | $a = x; b = x + 1; c = 2 \cdot x;$ |
| Operator strength reduction | Dead code elimination |
| $b = 3 \cdot x;$ | $a = x; b = x + 1; c = 2 \cdot x;$ |
| $t = x << 1; b = x + t;$ | Removed if not referenced in subsequent code |
| Code motion | Common Sub-expression elimination |
| $\text{for } (i = 1; i \leq a \cdot b) \{ \}$ | $a = x + y; b = a + 1; c = x + y;$ |
| $t = a \cdot b; \text{for } (i = 1; i \leq t) \{ \}$ | $a = x + y; b = a + 1; c = a$ |



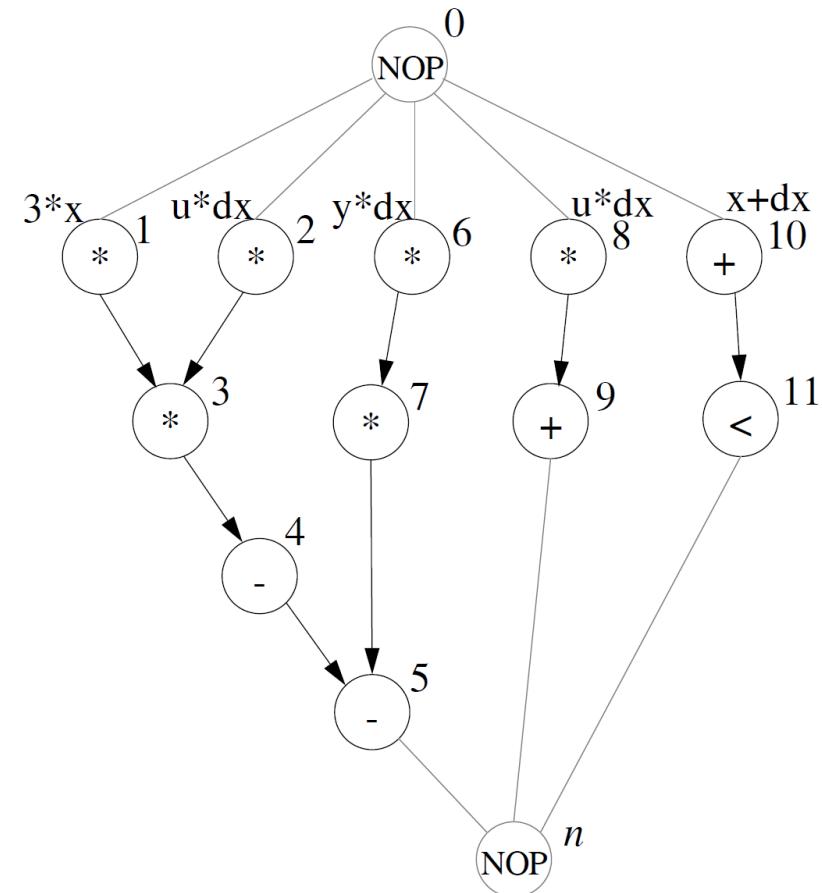
- Consider the following program [1]:

```

repeat
    xl = x + dx;
    ul = u - (3 * x * u * dx) - (3 * y * dx);
    yl = y + u * dx;
    c = xl < a;
    x = xl; u = ul; y = yl;
until (c);

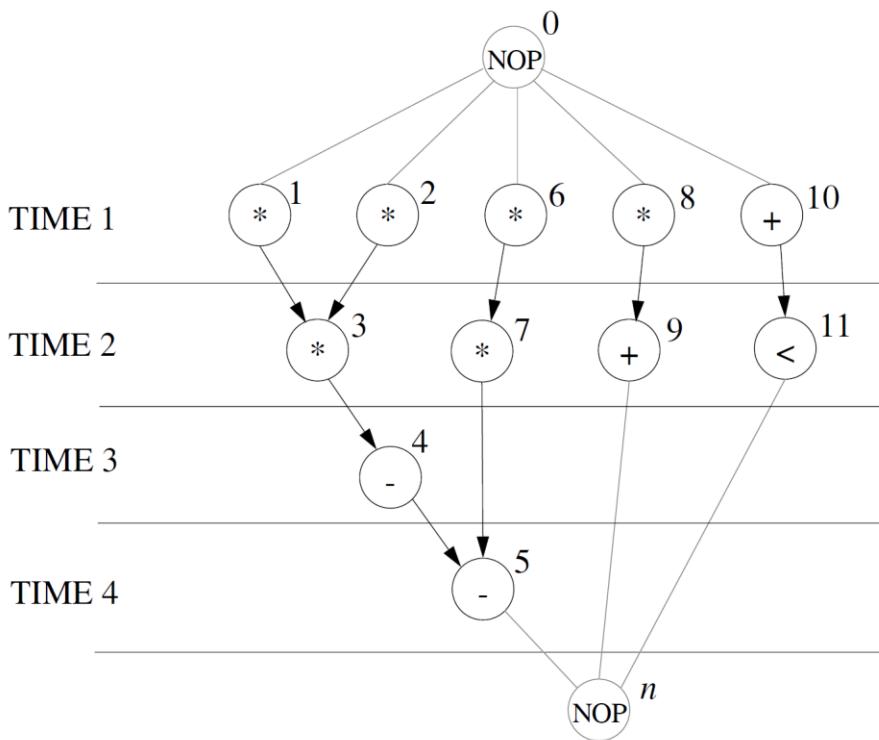
```

| | |
|--|--------|
| $xl = x + dx$ | v10 |
| $ul = u - (3 \cdot x \cdot u \cdot dx) - (3 \cdot y \cdot dx)$ | v1-v7 |
| $yl = y + u \cdot dx$ | v8, v9 |
| $c = x < a$ | v11 |

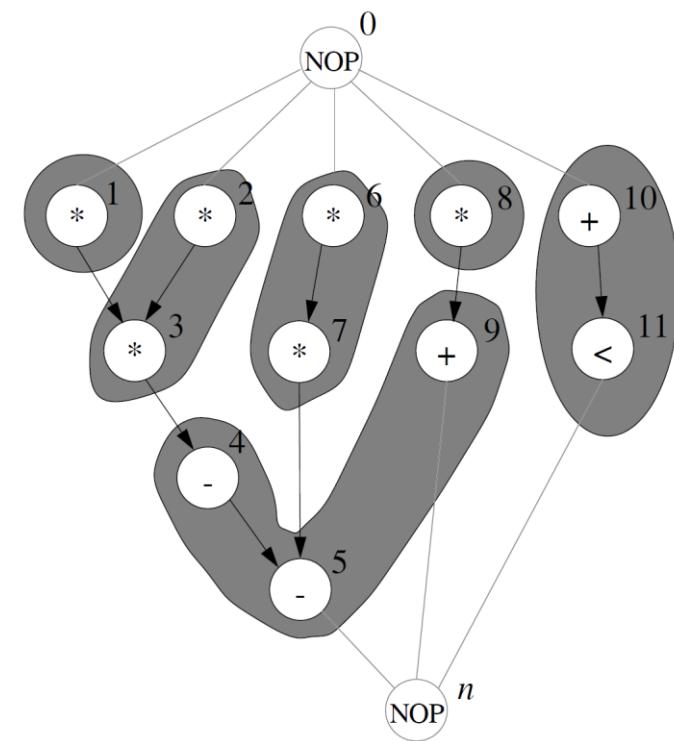




Scheduling and Binding



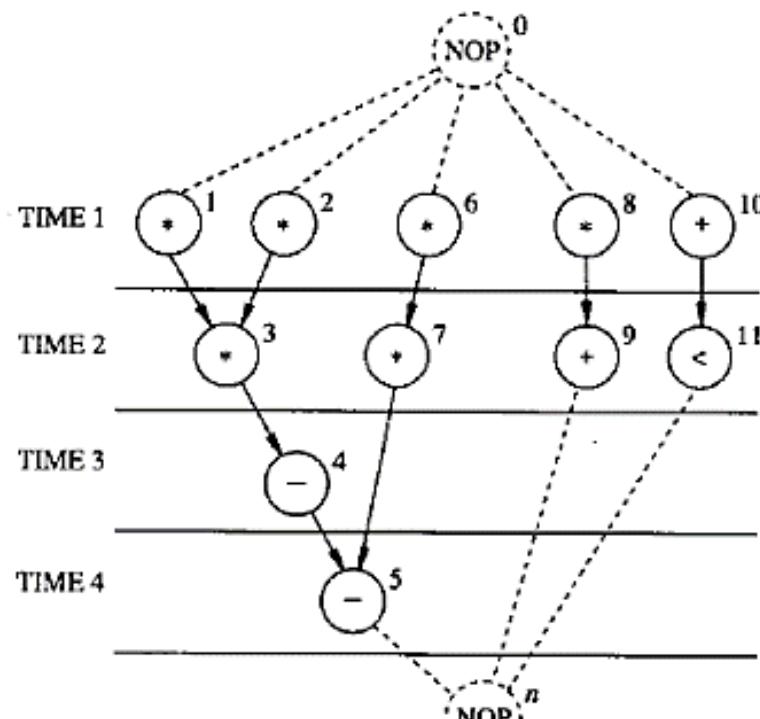
Sequencing Graph – 6 Multipliers, 5 ALUs



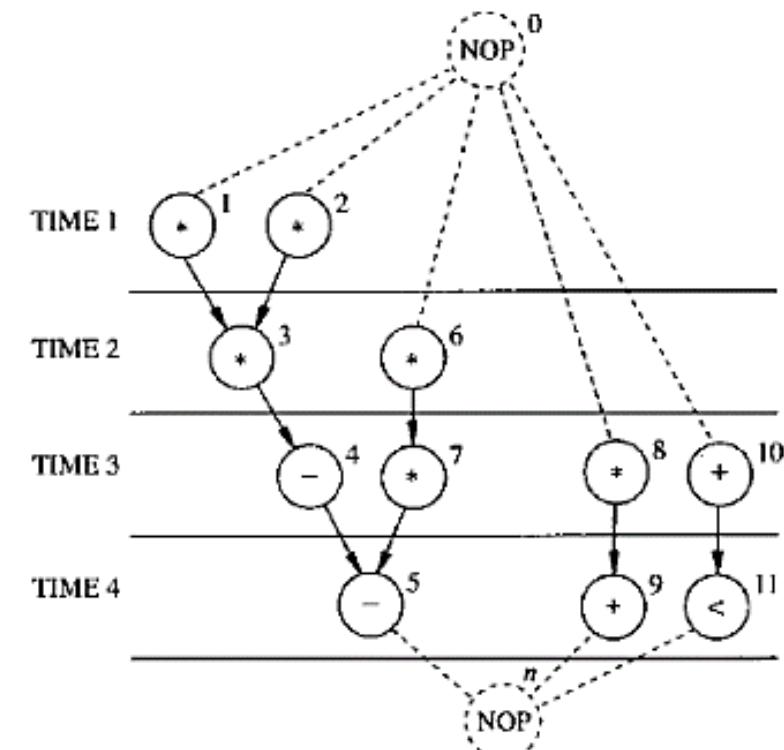
Allocation and Binding – 4 Multipliers, 2 ALUs

| | |
|-------------------|---|
| Scheduling | Associate a start time to each operation |
| Resources Sharing | A component may be used for several operations → sequential scheduling |
| Binding | Relates operations to available resources Specifies which resource implements an operation |

- **ASAP** – as soon as possible: the start time for each operation is the minimum allowed by all dependencies; yields the minimum start-time values
- **ALAP** – as late as possible, provides the maximum corresponding values



ASAP Scheduling



ALAP Scheduling



Mobility



- Mobility defines the start time span of an operation
 - Use ASAP and ALAP scheduling
 - The difference in scheduling determines the mobility of the operations
 - Mobility can be exploited for a more efficient scheduling
- Zero Mobility
 - implies that an operation can be started only at the given time step in order to meet the overall latency constraints. When the mobility is larger than zero, then it measures the span of the time interval in which it may be started
- Mobility example:
 - Operations 1 – 5: mobility = 0
 - Operations 6 – 7: mobility = 1
 - Operations 8 – 11: mobility = 2



Data-Path Synthesis and Optimization

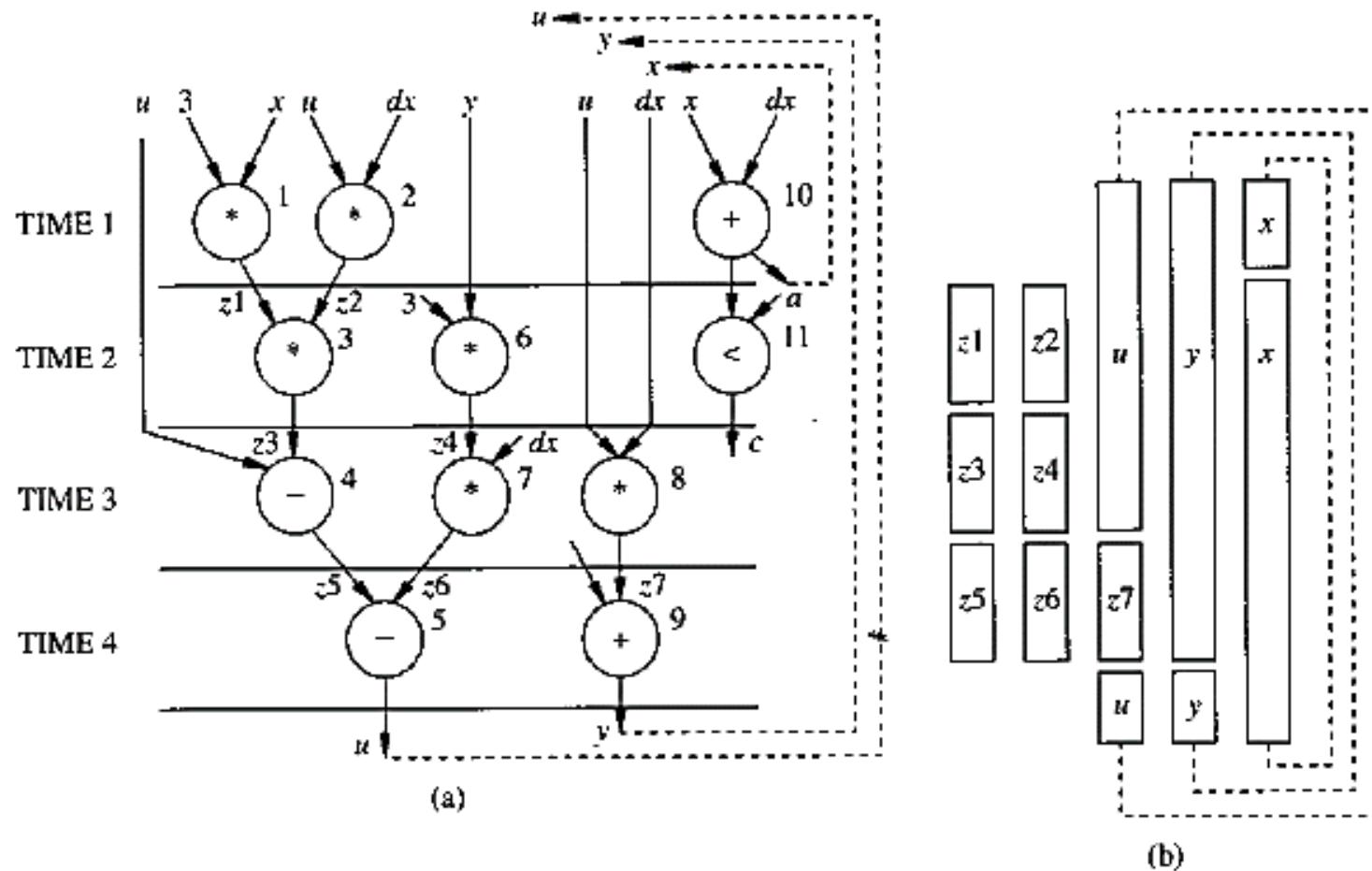


- Hardware models:
 - Functional Units
 - Registers
 - Register-File
 - Multiplexers
 - Tri-state buffers
 - Buses
- Parameters defining the hardware model:
 - Clocking strategy: single or multiple phase clocks
 - Interconnect: MUX and/or BUS based
 - Clocking of functional units
 - Single-cycle
 - Multi-cycle
 - Chaining
 - Pipeline



- **Data-Path synthesis**
 - Generate structural data-path realization from scheduled DFG
 - Two steps: allocation and binding
- **Allocation – chooses FUs and registers**
 - select the one that best matches the design constraints
 - Example: If area is more important than speed, adder is implemented using ripple-carry, if speed is more important than area, adder is implemented using look-ahead
- **Binding – assigns operations to FUs, variables to registers**
 - the aim is to connect FUs such that the cost of interconnection (number of mux, BUS) is minimized
- **Register-Binding Algorithm**
 - Lifetime of a variable – number of cycle times in which that variable is alive
 - Analyses the lifetimes of all variables
 - Establish the required number of registers

Register Allocation and Binding

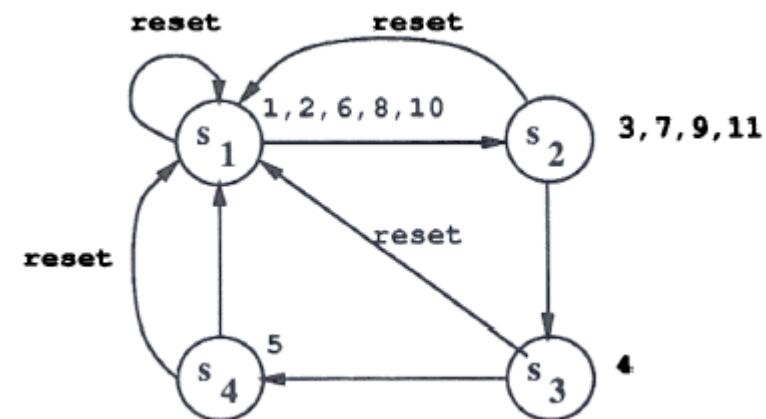
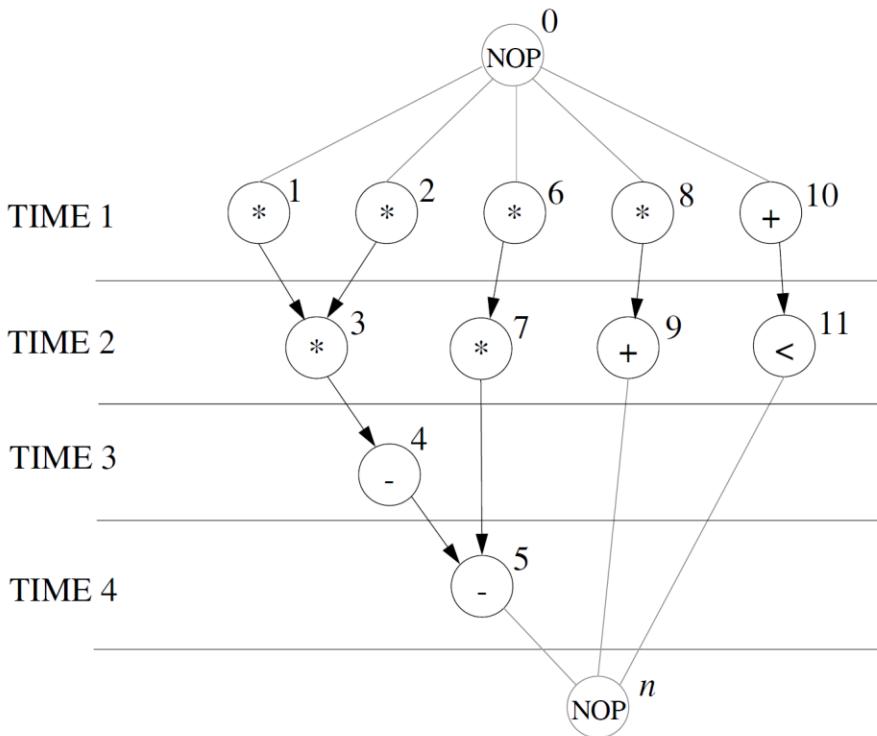


Variable lifetime → register allocation

a. Sequencing graph, b. Variable lifetimes → 5 registers required



- The control flow described by the schedule is implemented using a state machine, one state for each control step
- The control signals generated from each state activates the Functional Units, Registers, MUX, BUS selects, etc.

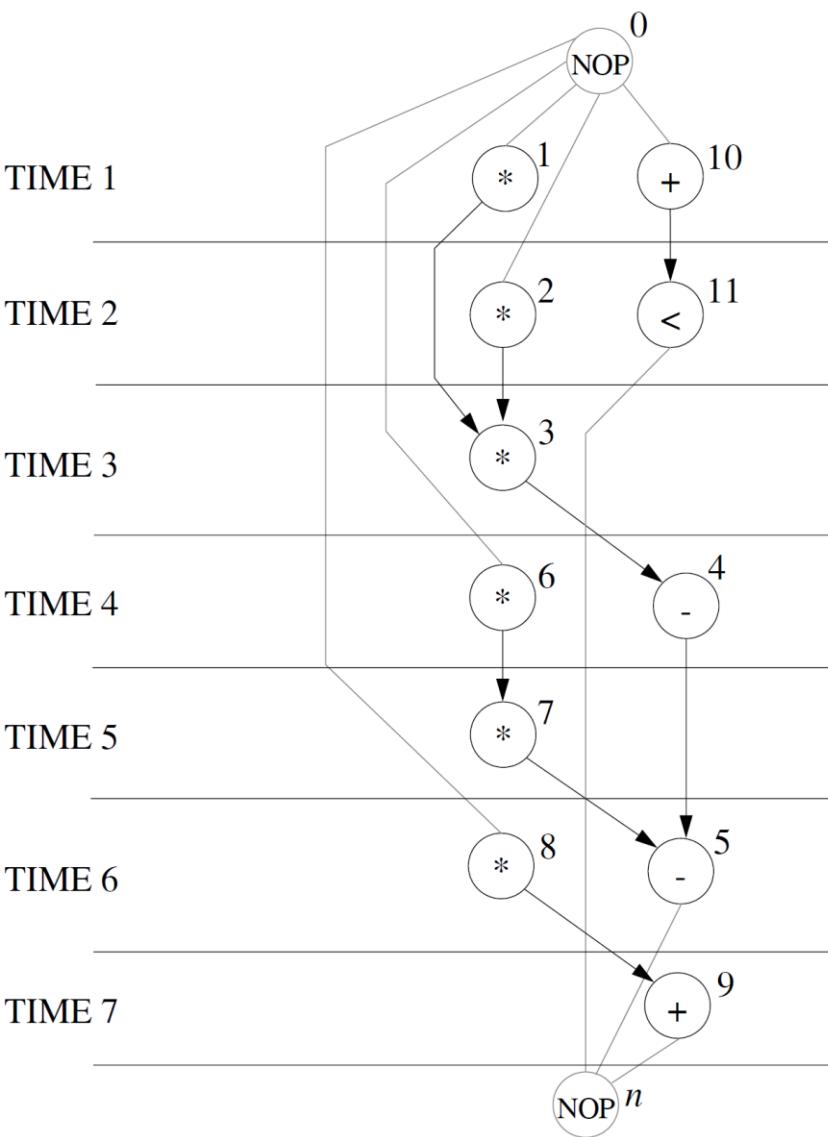


ASAP Scheduling FSM state transition diagram

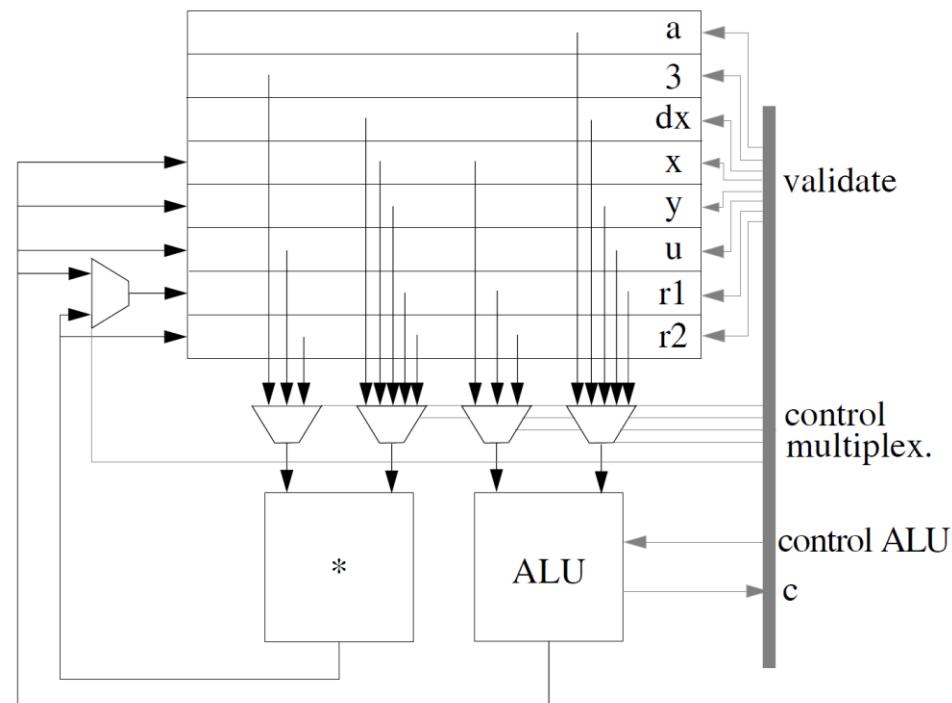
The numbers by the vertices of the diagram are the reference to the activation signals



Scheduling With Resource Constraints



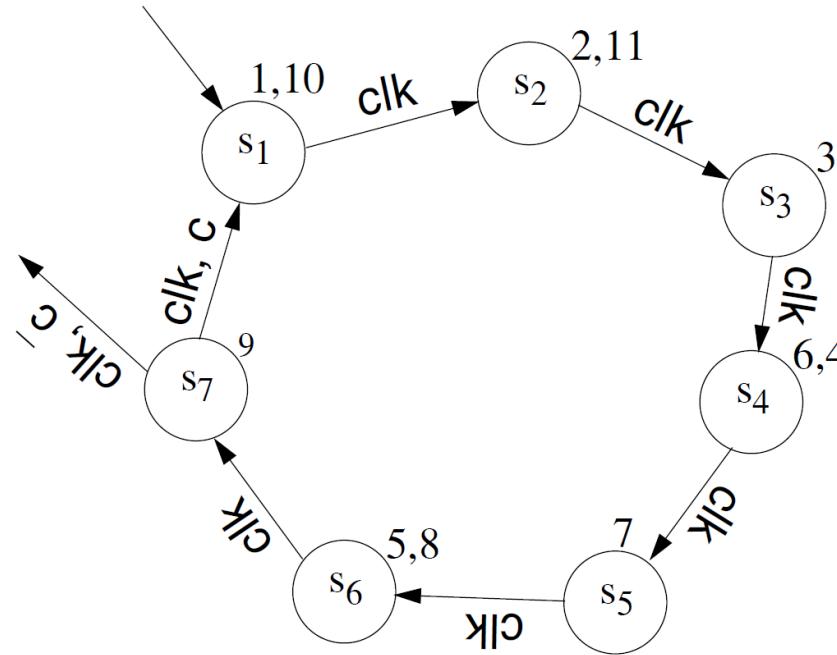
1 Multiplier and 1 ALU



MUX-based data-path



- 1 Multiplier and 1 ALU
 - One state for each clock cycle.
 - In each state the generated signals activate the needed operations



FSM state transition diagram



Synthesis of Pipeline Circuits



- **Pipelining** – a common technique for enhancing the performance of a digital circuit or processor
- The circuit is partitioned in a linear array of stages each concurrently executing a task on a different set of data and feeding its results to the following stage
- Pipelining increases **throughput**.
- No resource constraints
- Operations have unit execution delays



Design Constraints and Optimizations



- **Time** → speed; latency, throughput, clock frequency
- **Space** → cost; multilevel parallel combinational logic, sequential component sharing, pipelining
- **Power** → battery life; sleep modes
- **Testability** → crash; BIST (Built In Self Test)
- **Design styles:**
 - **Latency** (response delay time) optimization: single-cycle, multilevel parallel combinational logic data-path, no resource constraints, and slow clock
 - **Throughput** (frequency of result generation) optimization: pipelined data path, fast clock.
 - **Space** (resource) optimization: sharing, reuse of components, multi-cycle data-path.
 - **Space** (communication) optimization: MUX-based or 1, 2, 3 BUS multi-cycle data-path.

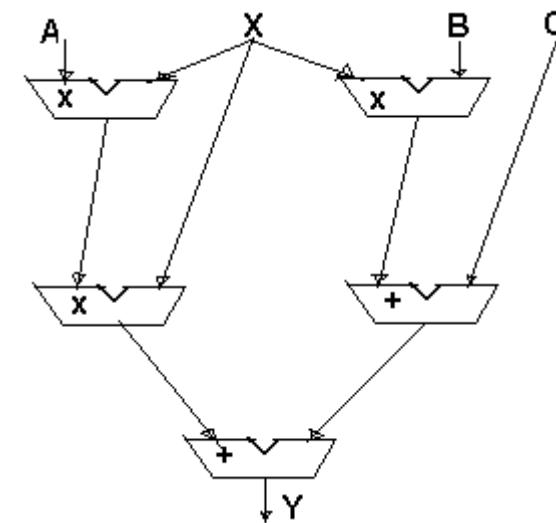
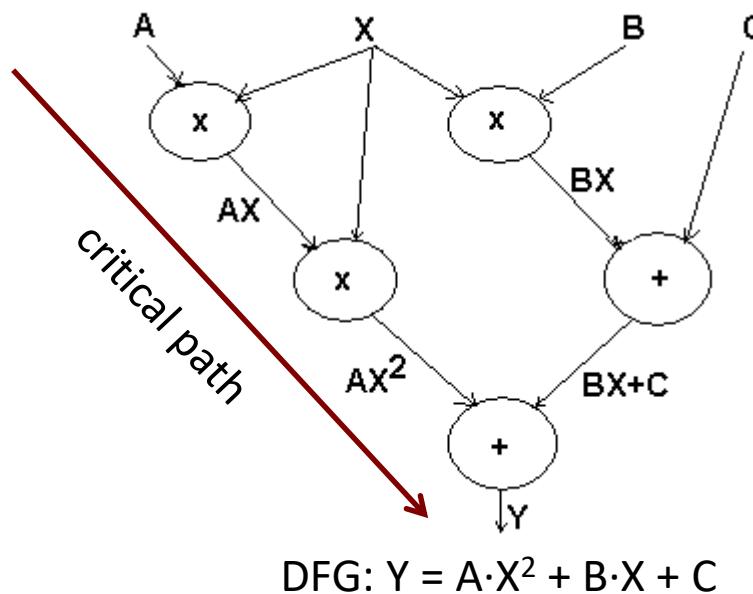


Design Constraints and Optimizations



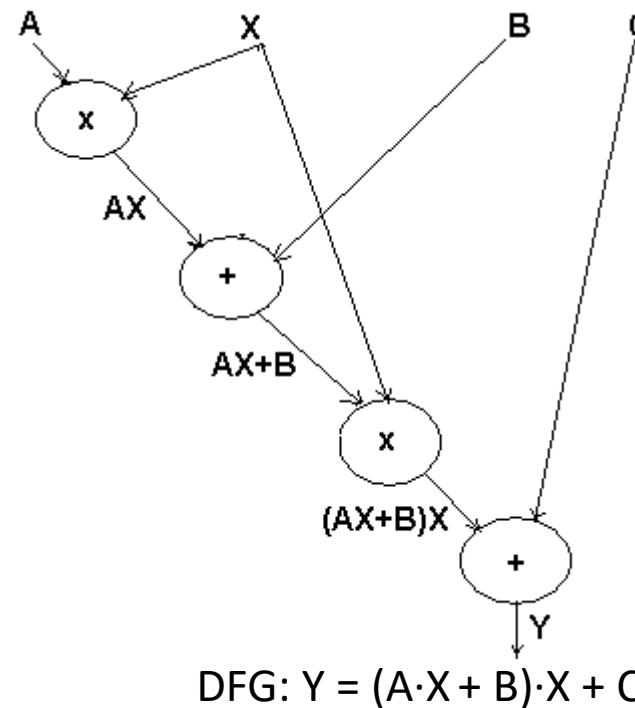
- **Single-cycle design**
 - Design for: optimal latency.
 - Resources: combinational functional units (FU), no resource constraints.
 - Communication: direct, point-to-point connections between FUs.
 - Clock frequency: defined by the critical path, the longest composed delay of the serial connected FUs
- **Pipeline design**
 - Design for: optimal throughput.
 - Resources: combinational FUs, inter stage registers, possible resource constraints.
 - Communication: through inter stage pipeline registers. The combinational FUs are isolated through inter stage pipeline registers and can work in parallel on different execution phases.
 - Clock frequency: defined by the slowest functional component and register overhead.
- **Multi-cycle design**
 - Design for: high clock frequency with spatial constraints. Resource sharing – reuse.
 - Resources: limited number of combinational FUs, temporary registers.
 - Communication: through temporary registers, MUX-es and BUS-es. The intermediate results of combinational FUs are registered for further use.
 - Clock frequency: defined by the slowest functional component and register overhead.

- Design of application specific FUs, based on standard components
- **Solution 1 – Single-cycle**
- We have to rewrite the equation in terms of standard, 2 inputs, 1 - output operations: $A \cdot X$, $B \cdot X$, $(A \cdot X) \cdot X$, $(B \cdot X) + C$, $((A \cdot X) \cdot X + (B \cdot X)) + C$
- The DFG shows the data dependencies and precedence of operations



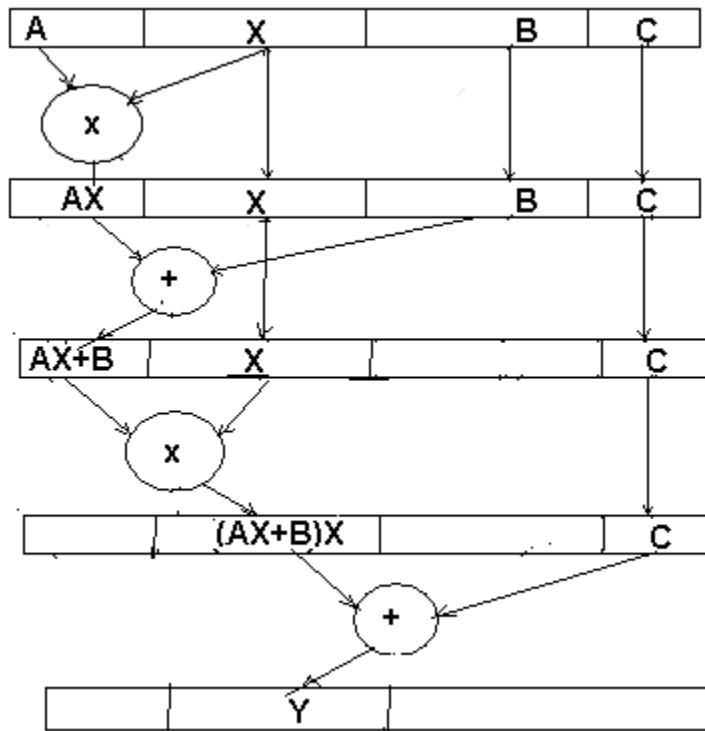
- Resources: 3 Multipliers, 2 ALUs. Critical Path defines the response time ($2 \cdot x + 1 \cdot +$)
- Data Flow driven execution: NO conditions, NO control signals

- Solution 2 – Single-cycle
- Rewrite the equation: $Y = (A \cdot X + B) \cdot X + C$



- Resources: 2 Multipliers, 2 ALUs. Critical Path defines the response time ($2 \cdot x + 2 \cdot +$)
- Data Flow driven execution: NO conditions, NO control signals
- 2 solutions with different response times and resource utilization

- Solution 3 – Pipeline (based on the second single-cycle solution)

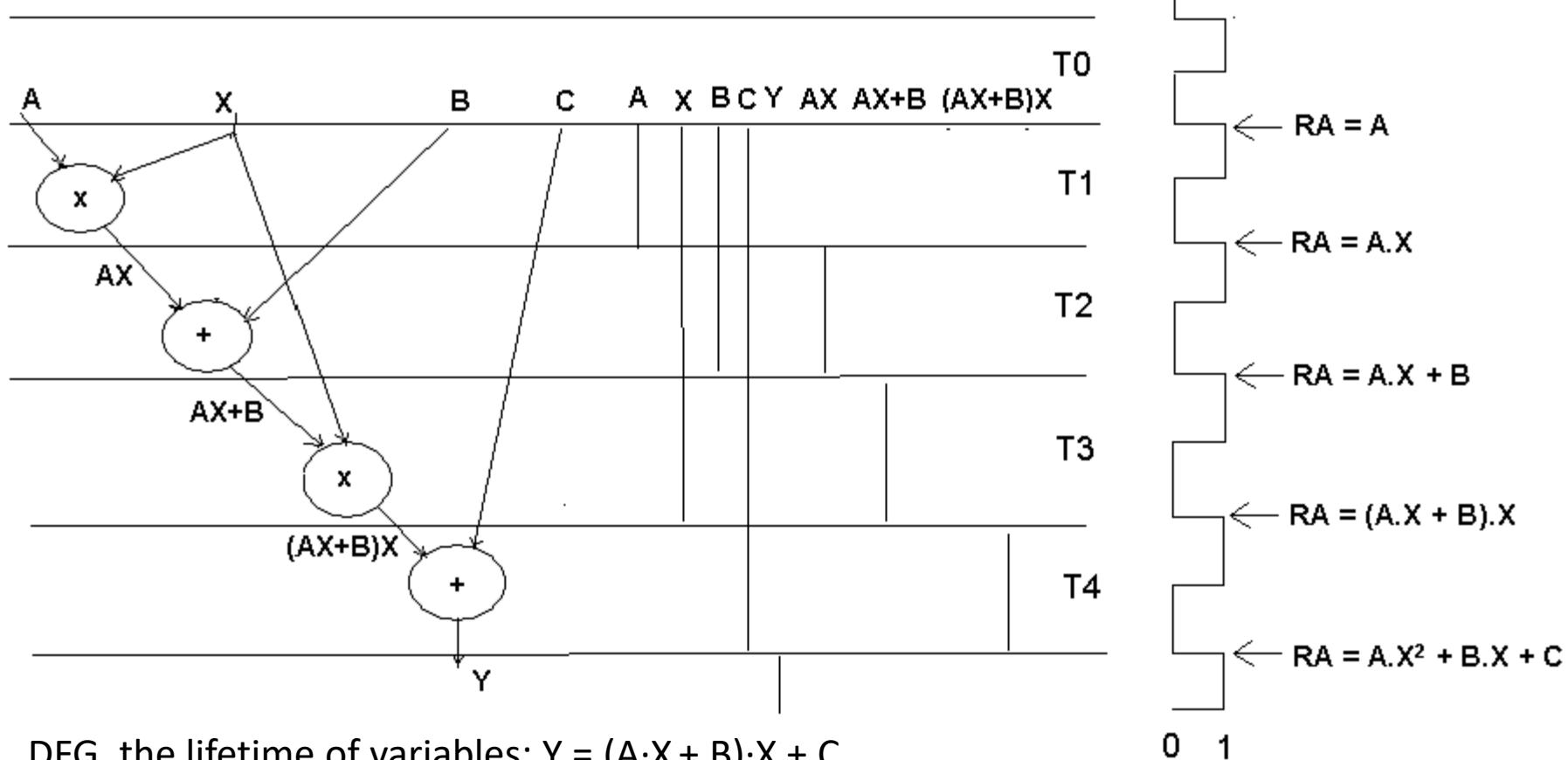


4 Pipeline stages: $Y = (A \cdot X + B) \cdot X + C$

- Suppose the Adder and Multiplier delays are equal
- Introduce 4 pipeline registers after every combinational circuit
→ 4 balanced pipeline stages
- At every clock cycle a new set of variables A, X, B, C can enter the pipeline
- The pipeline stages are working simultaneously**
- The latency of the circuit (the propagation delay through the complete pipeline) is composed from the delay of the combinational and register components of the stages.
 - Latency > Single-cycle design.
- The pipelined design needs more resources than single-cycle
- After a pipeline filling period, in every clock cycle we obtain a new value of Y.
- The throughput is 1 result /clock period.**



- Solution 4 – Multi-cycle MUX-Based, 1 Multiplier and 1 ALU

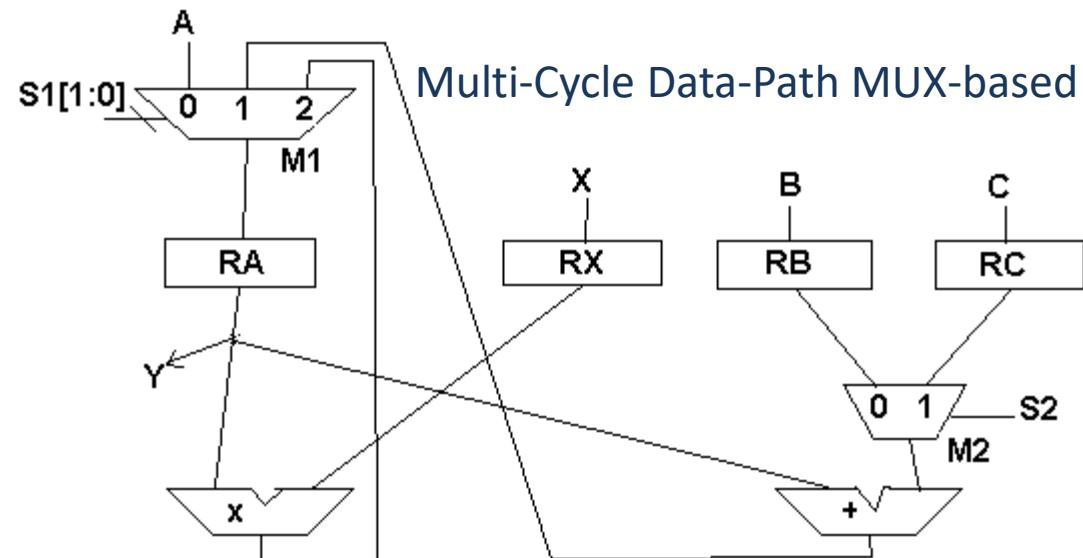


- Use 4 temporary registers for the initial A, X, B, C values: RA, RB, RC, RX
- RA can be used as for A, A·X, A·X+B, (A·X+B)·X and Y values – 5 different values

Problem: $Y = A \cdot X^2 + B \cdot X + C$

| | | Clock1 | Clock2 | Clock3 | Clock4 |
|------------|------|---------------------------|--------------------------|---|---------------------------|
| Multiplier | in 0 | A | | $A \cdot X + B$ | |
| Multiplier | in 1 | X | | X | |
| Multiplier | Out | $A \cdot X$ | | $(A \cdot X + B) \cdot X$ | |
| Adder | in 0 | | $A \cdot X$ | | $(A \cdot X + B) \cdot X$ |
| Adder | in 1 | | B | | C |
| Adder | Out | | $A \cdot X + B$ | | Y |
| RA in | | $A \cdot X$ Multiplier | $A \cdot X + B$ Adder | $(A \cdot X + B) \cdot X$ Multiplier | Y Adder |

Connection between FUs and RA





Problem: $Y = A \cdot X^2 + B \cdot X + C$



- Concrete RTL description for $Y = (A \cdot X + B) \cdot X + C$ and control signals

T0: The initial values of the variables have been loaded

T1: $RA \leftarrow RA \times RX; \quad S2 \leftarrow X; \quad S1 \leftarrow 2; \quad // RA = A \cdot X$

T2: $RA \leftarrow RA + RB; \quad S2 \leftarrow 0; \quad S1 \leftarrow 1; \quad // RA = A \cdot X + B$

T3: $RA \leftarrow RA \times RX \quad S2 \leftarrow X; \quad S1 \leftarrow 2; \quad // RA = (A \cdot X + B) \cdot X$

T4: $RA \leftarrow RA + RC; \quad S2 \leftarrow 1; \quad S1 \leftarrow 1; \quad // RA = A \cdot X^2 + B \cdot X + C$

RTL description

MUX control

Tracing

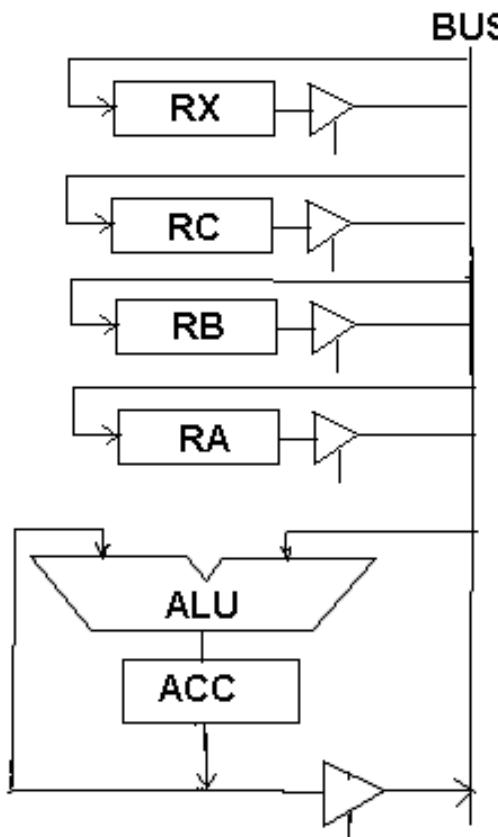
The control unit can be implemented as a 4-state FSM (5 by includings T0)

Pipeline vs. Multi-cycle throughput difference:

- Pipeline:** at every clock cycle a new set of variables A, X, B, C can enter the pipeline and a new result is produced, after the filling time.
- Multi-cycle:** at every 5th clock cycle a new set of variables can enter and a result is produced.
- The clock cycles can have the same values.



- Solution 5 – Multi-cycle 1-BUS, Temporary ACC register, 1 ALU



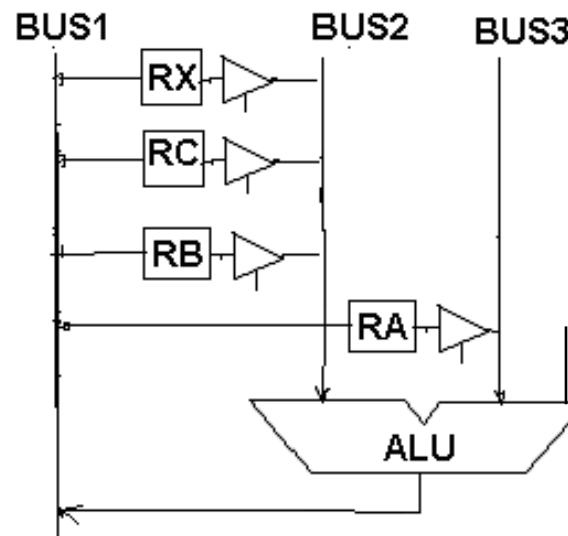
Multi-Cycle Data-Path
1-BUS

| RTL description | Control Definition | Tracing |
|--|---|---------|
| T0: Suppose, that the initial values of the variables have been loaded | | |
| T1: $ACC \leftarrow ALUtransfer\ RA;$ | rdRA, wrACC, ALUop=transfer; // ACC = A | |
| T2: $ACC \leftarrow ACC \times RX;$ | rdRX, wrACC, ALUop= *; // ACC = A · X | |
| T3: $ACC \leftarrow ACC + RB;$ | rdRB, wrACC, ALUop= +; // ACC = A · X + B | |
| T4: $ACC \leftarrow ACC \times RX;$ | rdRX, wrACC, ALUop= *; // ACC = (A · X + B) · X | |
| T5: $ACC \leftarrow ACC + RC;$ | rdRC, wrACC, ALUop= +; // ACC = A · X ² + B · X + C | |

Tri-state buffers for all writes to the bus.
The Control Unit can be implemented as a 5 state FSM.



- Solution 5 – Multi-cycle 3-BUS, 1 ALU



Multi-Cycle Data-Path
3-BUS

| | | |
|--|-----------------------------|---------------------------------------|
| T0: Suppose, that the initial values of the variables have been loaded | | |
| T1: $RA \leftarrow RA \times RX;$ | rdRA, rdRX, ALUop= *, wrRA; | // $RA = A \cdot X$ |
| T2: $RA \leftarrow RA + RB;$ | rdRA, rdRB, ALUop= +, wrRA; | // $RA = A \cdot X + B$ |
| T3: $RA \leftarrow RA \times RX;$ | rdRA, rdRX, ALUop= *, wrRA; | // $RA = (A \cdot X + B) \cdot X$ |
| T4: $RA \leftarrow RA + RC;$ | rdRA, rdRC, ALUop= +, wrRA | // $RA = A \cdot X^2 + B \cdot X + C$ |

RTL description

Control Definition

Tracing

BUS3 can be used for initial loading of the registers, through the ALU
The Control Unit can be implemented as a 4 state FSM.



Problems – Homework



- For the equation: $a \cdot x^2 + b \cdot x + c$, draw the dataflow graph for:
 - a. ASAP, ALAP scheduled execution
 - b. Pipelined execution
 - Compare the latencies, throughputs and costs of the solutions.
 - a. Implement the multi-cycle 2-bus based solution

- For the equation: $w = a \cdot (b \cdot x + z) - c \cdot (a \cdot y + x)$ assume:
 - a. Implementation without constraints.
 - b. Implementation with 1 Add/Subtract and 1 Multiplier unit; Latencies = 1 clk.
 - For the implementations a and b show the scheduled DFGs, the lifetime of the variables, the minimal number of necessary registers and the corresponding data paths for a multi-cycle design.



References



1. P. Eles, Z. Peng, “System Synthesis of Digital Systems”, Lecture slides, March 2000,
<http://www.ida.liu.se/~petel/SysSyn/lect1.frm.pdf>
2. P. Coussy, D.Gajski, M.Meredith, and A.Takash, „An introduction to high-level synthesis”, *IEEE Design Test of Computers*, 26(4):8 – 17, jul. 2009.
3. Egon Boerger and Robert Staerk, “A Method for High-Level System Design and Analysis”, (Abstract State Machines chapter), Springer-Verlag 2003.



Computer Architecture

Lecturer: Mihai Negru

2nd Year, Computer Science

Lecture 3: MIPS ISA

<http://users.utcluj.ro/~negrum/>



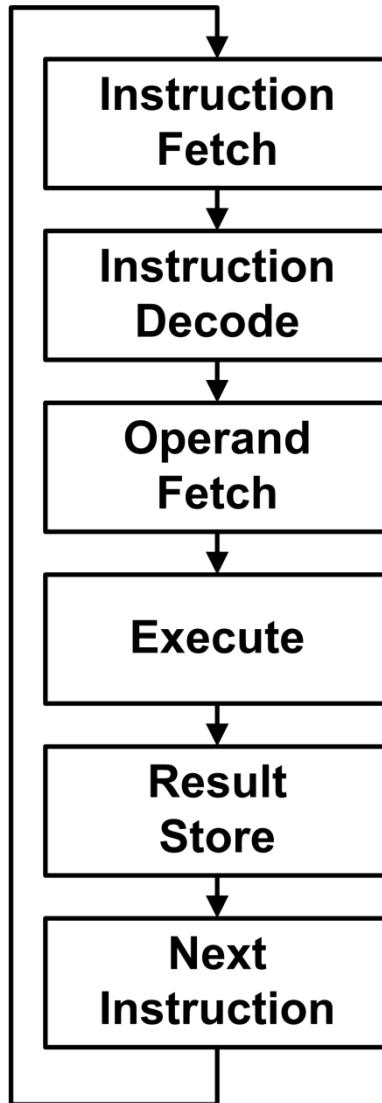
MIPS ISA



- MIPS – Microprocessor without Interlocked Pipeline Stages
- ISA – Instruction Set Architecture
 - The interface between Hardware and Software
 - ISA Components:
 - Memory organization
 - Registers
 - Data Types and Data Structures
 - Instruction Formats
 - Instruction Set
 - Addressing modes
 - Flow of Control
 - Input/Output
 - Interrupts
 - ...



Instruction Execution Cycle



Obtain/Read Instruction from Program Memory/Storage

Determine the instruction format or encoding

Locate and obtain the operands of the instruction and the location of the result

Compute result of the instruction:

- value, status or address

Write/Save the result of the instruction:

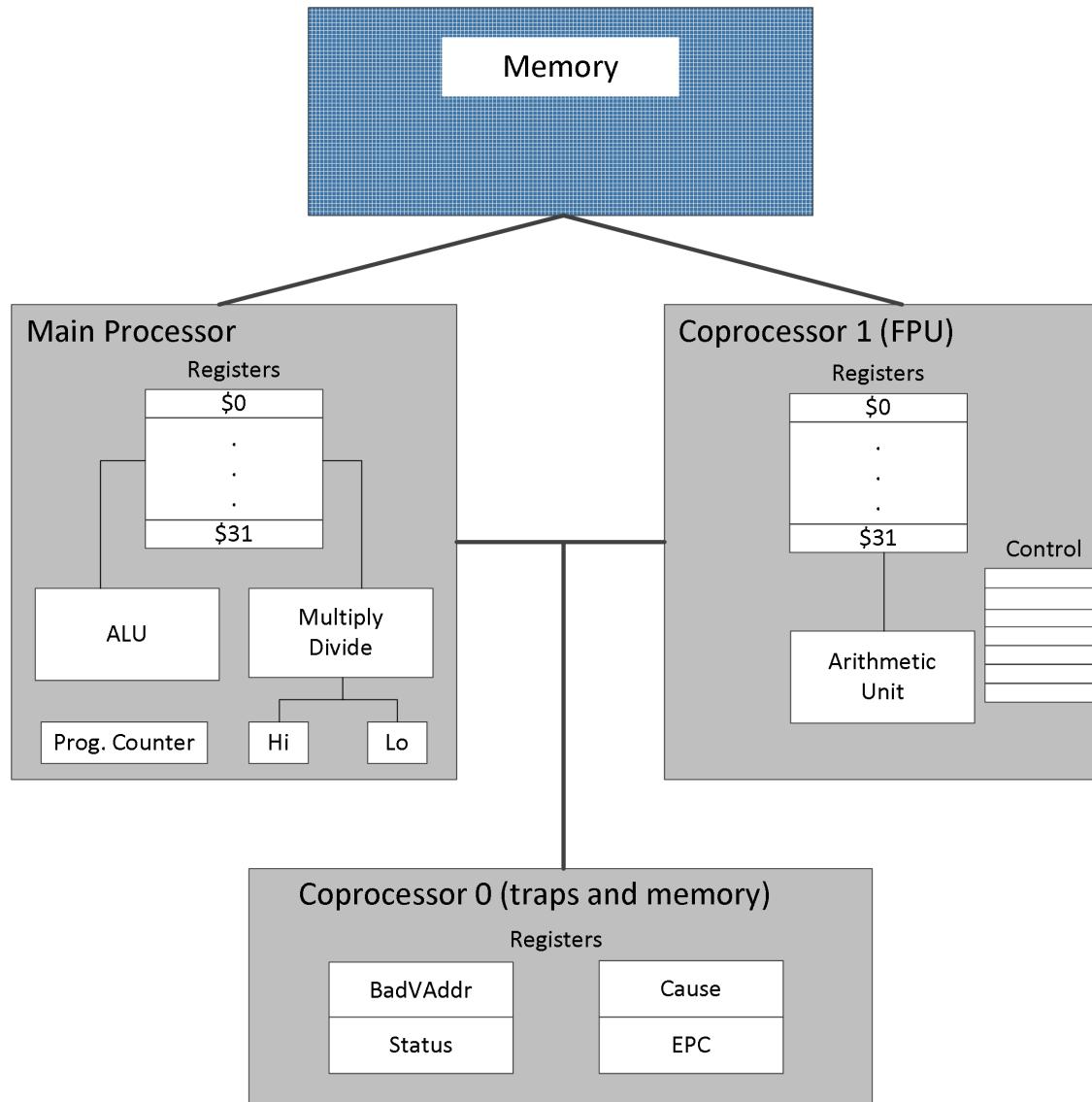
- register or memory

Determine the next instruction (address) to be executed:

- Normal operation
- Jumps, Conditions, Branches



MIPS Architecture – Block Diagram



MIPS – LOAD/STORE ISA

- 3 Address Machine
- General Purpose Register
- 32 bit µprocessor
- Main Processor – CPU
- Coprocessor 0 – OS Support
- Coprocessor 1 – FPU
- Memory Unit



MIPS CPU Registers



- 32 x 32-bit General purpose registers
 - R0 – R31
 - R0 has fixed value of zero.
 - Attempt to writing into R0 is not illegal, but it is ineffective
 - R31 holds the return address of a procedure call
- PC – Program Counter
 - contains the address of the next instruction to be fetched
- Hi, Lo
 - store partial result of multiplication and division operations



MIPS General Purpose Registers



| Name | Register Number | Usage |
|-------------|-----------------|---|
| \$zero | 0 | Always 0 |
| \$at | 1 | Reserved for assembler |
| \$v0 - \$v1 | 2-3 | Expression evaluation and function results |
| \$a0 - \$a3 | 4-7 | Arguments |
| \$t0 - \$t7 | 8-15 | Temporary, saved by caller |
| \$s0 - \$s7 | 16-23 | Temporary, saved by called function |
| \$t8 - \$t9 | 24-25 | Temporary, saved by caller |
| \$k0 - \$k1 | 26-27 | Reserved for kernel (OS) |
| \$gp | 28 | Global Pointer |
| \$sp | 29 | Stack Pointer (top of stack) |
| \$fp | 30 | Frame Pointer (beginning of current frame) |
| \$ra | 31 | Return Address (pushed by call instruction) |



MIPS Coprocessors



- Coprocessor 0 – CP0
 - Provides support for the operating system (OS)
 - Exception handling
 - Memory management
 - Scheduling and control of critical resources
 - CP0 registers – 16 registers
 - Status Register (CP0, Reg12) – processor status and control, interrupt bits
 - Cause Register (CP0, Reg13) – cause of the most recent interrupt
 - EPC Register (CP0, Reg14) – program counter at the last interrupt
 - BadVAddr Register (CP0, Reg08) – virtual address for the most recent address related exception
 - Instructions for reading and writing the CP0 Registers:
 - `mfc0 $k0, $13 # Move from CP0, Reg13 (Cause Register) in $k0`
 - `mtc0 $0, $12 # Move value 0 in CP0, Reg12 (Status Register)`
- Coprocessor 1 – CP1, Floating Point Unit (FPU)
 - 32x32-bit Floating Point Registers (f0 – f31)
 - Five control registers



MIPS Memory Organization



- A large, single-dimension array
 - A memory address – an index in the array
- **Byte addressing** – the index points to a byte of memory
- Word – 32 bits (4 bytes)
- 32-bit addresses
 - 2^{32} bytes $\rightarrow 2^{30}$ words:
0, 1, 2, ... to $2^{32}-1 \rightarrow 0, 4, 8, \dots$ to $2^{32}-4$
- Endianness – can be selected
 - Little / Big endian
- **Data Alignment: YES**
 - 32 bit word starts at a multiple of 4 bytes address
 - 16 bit word starts at multiple of 2 bytes address

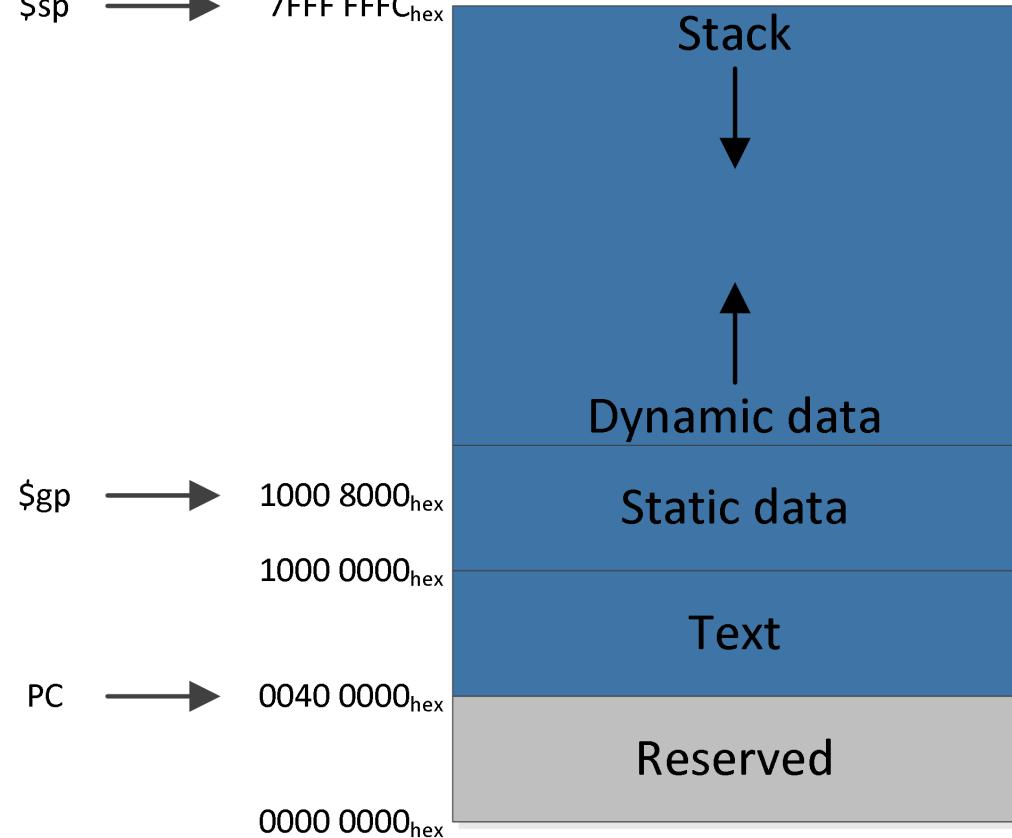
\$sp \longrightarrow 7FFF FFFC_{hex}

\$gp \longrightarrow 1000 8000_{hex}

1000 0000_{hex}

PC \longrightarrow 0040 0000_{hex}

0000 0000_{hex}





MIPS Data Types



- MIPS operates on:
 - 32-bit (unsigned or 2's complement) integers
 - 32-bit (single precision floating point) real numbers
 - 64-bit (double precision floating point) real numbers
- Locations in General Purpose Registers
 - 32-bit words, bytes (8-bits) and half words (16-bits) can be loaded into GPRs
 - When loading into GPRs, bytes and half words are **Zero or Sign Extended** to fill the 32 bits
- Locations in Floating Point Registers
 - Only 32-bit units can be loaded into FPRs
 - 32-bit real numbers are stored in FPRs
 - 64-bit real numbers are stored in two consecutive FPRs, starting with even numbered register



MIPS Instruction Formats



- 32-bit length instructions
- 3 instruction formats
 - R-type instructions – used for arithmetical/logical operations
 - I-type instructions – used for arithmetical/logical operations with immediate values, memory data transfers and conditional jumps or branches
 - J-type instructions – used for unconditional jumps

| | | | | | | | |
|--------|--------|-------|-------|-------|---------------------|----------|---|
| R-type | 31 | 26 25 | 21 20 | 16 15 | 11 10 | 6 5 | 0 |
| | opcode | rs | rt | rd | sa | function | |
| I-type | 31 | 26 25 | 21 20 | 16 15 | address / immediate | | 0 |
| | opcode | rs | rt | | | | |
| J-type | 31 | 26 25 | | | target address | | 0 |
| | opcode | | | | | | |

Detailed description: The table illustrates the three MIPS instruction formats. R-type instructions have fields for opcode (6 bits), rs (5 bits), rt (5 bits), rd (5 bits), sa (5 bits), and function (6 bits). I-type instructions have fields for opcode (6 bits), rs (5 bits), rt (5 bits), and a combined address/immediate field (16 bits). J-type instructions have fields for opcode (6 bits) and a target address field (26 bits).



MIPS Instruction Formats



| Field | Description |
|---------------------|--|
| opcode | 6-bit primary operation code |
| rs | 5-bit source register specifier |
| rt | 5-bit target (source/destination) register specifier or used to specify functions within the primary opcode value REGIMM |
| rd | 5-bit destination register specifier |
| sa | 5-bit shift amount |
| function | 6-bit function field used to specify functions within the primary opcode SPECIAL (000000) |
| address / immediate | 16-bit immediate used for: logical operands, arithmetic signed operands |
| | load/store address byte offsets |
| | PC-relative branch signed instruction displacement |
| target address | 26-bit index shifted left two bits to supply the low-order 28 bits of the jump target address |

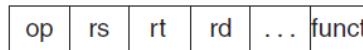


MIPS Addressing Modes

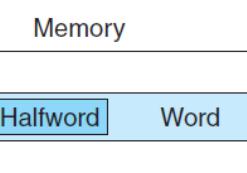
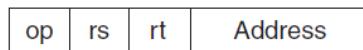
1. Immediate addressing



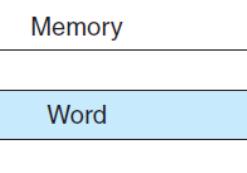
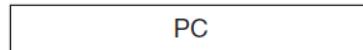
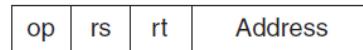
2. Register addressing



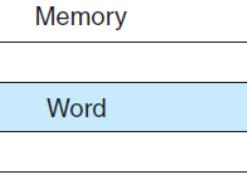
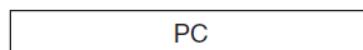
3. Base addressing



4. PC-relative addressing



5. Pseudodirect addressing



- Immediate addressing

- Register addressing

- Base addressing (indexed)

- The only **memory data addressing**
- Register content + address
- $r_0 + \text{Address} \rightarrow \text{absolute addressing}$
- $r_i + \text{Address } (=0) \rightarrow \text{register indirect}$

- PC-relative addressing

- Branch instructions
- Branch address: $\text{PC} + 4 + 4 * \text{Address}$

- Pseudo-direct addressing

- Jump Instructions
- Jump address: $\text{PC}[31:28] | 4 * \text{Address}$
- Jumps in 256 MB regions

[1]



MIPS Core Instruction Set



| Category | Instruction | Example | Meaning | Comments |
|--------------------|----------------------------------|---------------------|--|---|
| Arithmetic | add | add \$s1,\$s2,\$s3 | $\$s1 = \$s2 + \$s3$ | Three operands |
| | subtract | sub \$s1,\$s2,\$s3 | $\$s1 = \$s2 - \$s3$ | Three operands |
| | add immediate | addi \$s1,\$s2,100 | $\$s1 = \$s2 + 100$ | + constant |
| Data transfer | load word | lw \$s1,100(\$s2) | $\$s1 = \text{Memory}[\$s2 + 100]$ | Word from memory to register |
| | store word | sw \$s1,100(\$s2) | $\text{Memory}[\$s2 + 100] = \$s1$ | Word from register to memory |
| | load half unsigned | lhu \$s1,100(\$s2) | $\$s1 = \text{Memory}[\$s2 + 100]$ | Halfword memory to register |
| | store half | sh \$s1,100(\$s2) | $\text{Memory}[\$s2 + 100] = \$s1$ | Halfword register to memory |
| | load byte unsigned | lbu \$s1,100(\$s2) | $\$s1 = \text{Memory}[\$s2 + 100]$ | Byte from memory to register |
| | store byte | sb \$s1,100(\$s2) | $\text{Memory}[\$s2 + 100] = \$s1$ | Byte from register to memory |
| Logical | load upper immediate | lui \$s1,100 | $\$s1 = 100 * 2^{16}$ | Loads constant in upper 16 bits |
| | and | and \$s1,\$s2,\$s3 | $\$s1 = \$s2 \& \$s3$ | Three reg. operands; bit-by-bit AND |
| | or | or \$s1,\$s2,\$s3 | $\$s1 = \$s2 \$s3$ | Three reg. operands; bit-by-bit OR |
| | nor | nor \$s1,\$s2,\$s3 | $\$s1 = \sim (\$s2 \$s3)$ | Three reg. operands; bit-by-bit NOR |
| | and immediate | andi \$s1,\$s2,100 | $\$s1 = \$s2 \& 100$ | Bit-by-bit AND with constant |
| | or immediate | ori \$s1,\$s2,100 | $\$s1 = \$s2 100$ | Bit-by-bit OR with constant |
| | shift left logical | sll \$s1,\$s2,10 | $\$s1 = \$s2 \ll 10$ | Shift left by constant |
| Conditional branch | shift right logical | srl \$s1,\$s2,10 | $\$s1 = \$s2 \gg 10$ | Shift right by constant |
| | branch on equal | beq \$s1,\$s2,25 | if ($\$s1 == \$s2$) go to PC + 4 + 100 | Equal test; PC-relative branch |
| | branch on not equal | bne \$s1,\$s2,25 | if ($\$s1 != \$s2$) go to PC + 4 + 100 | Not equal test; PC-relative |
| | set on less than | slt \$s1,\$s2,\$s3 | if ($\$s2 < \$s3$) $\$s1 = 1$; else $\$s1 = 0$ | Compare less than; two's complement |
| | set less than immediate | slti \$s1,\$s2,100 | if ($\$s2 < 100$) $\$s1 = 1$; else $\$s1 = 0$ | Compare < constant; two's complement |
| | set less than unsigned | sltu \$s1,\$s2,\$s3 | if ($\$s2 < \$s3$) $\$s1 = 1$; else $\$s1 = 0$ | Compare less than; unsigned numbers |
| Unconditional jump | set less than immediate unsigned | sltiu \$s1,\$s2,100 | if ($\$s2 < 100$) $\$s1 = 1$; else $\$s1 = 0$ | Compare < constant; unsigned numbers |
| | jump | j 2500 | go to 10000 | Jump to target address |
| | jump register | jr \$ra | go to $\$ra$ | For switch, procedure return |
| | jump and link | jal 2500 | $\$ra = \text{PC} + 4$; go to 10000 | For procedure call |

[1]



MIPS Instructions



| Assembly | Type | RTL Abstract | Instruction |
|-----------------------|------|---|----------------------------|
| add \$rd, \$rs, \$rt | R | $RF[rd] \leftarrow RF[rs] + RF[rt]$ | Add |
| sub \$rd, \$rs, \$rt | R | $RF[rd] \leftarrow RF[rs] - RF[rt]$ | Subtract |
| addi \$rt, \$rs, imm | I | $RF[rt] \leftarrow RF[rs] + S_Ext(imm)$ | Add Immediate |
| addu \$rd, \$rs, \$rt | R | $RF[rd] \leftarrow RF[rs] + RF[rt]$ | Add Unsigned (no overflow) |
| subu \$rd, \$rs, \$rt | R | $RF[rd] \leftarrow RF[rs] - RF[rt]$ | Subtract Unsigned |
| mult \$rs, \$rt | R | $Hi, Lo \leftarrow RF[rs] * RF[rt]$ | Signed Multiplication |
| multu \$rs, \$rt | R | $Hi, Lo \leftarrow RF[rs] * RF[rt]$ | Unsigned Multiplication |
| div \$rs, \$rt | R | $Lo \leftarrow RF[rs] / RF[rt],$ $Hi \leftarrow RF[rs] \bmod RF[rt]$ | Signed Division |
| divu \$rs, \$rt | R | $Lo \leftarrow RF[rs] / RF[rt],$ $Hi \leftarrow RF[rs] \bmod RF[rt]$ | Unsigned Division |
| mfhi \$rd | R | $RF[rd] \leftarrow Hi$ | Move from Hi |
| mflo \$rd | R | $RF[rd] \leftarrow Lo$ | Move from Lo |
| | | ... | |



MIPS Instructions



| Assembly | Type | RTL Abstract | Instruction |
|-----------------------|------|---|---------------------------------|
| and \$rd, \$rs, \$rt | R | $RF[rd] \leftarrow RF[rs] \& RF[rt]$ | Logical AND |
| or \$rd, \$rs, \$rt | R | $RF[rd] \leftarrow RF[rs] RF[rt]$ | Logical OR |
| xor \$rd, \$rs, \$rt | R | $RF[rd] \leftarrow RF[rs] ^ RF[rt]$ | Exclusive OR |
| andi \$rt, \$rs, imm | I | $RF[rt] \leftarrow RF[rs] \& Z_Ext(imm)$ | Logical AND unsigned constant |
| ori \$rt, \$rs, imm | I | $RF[rt] \leftarrow RF[rs] Z_Ext(imm)$ | Logical OR unsigned constant |
| xori \$rt, \$rs, imm | I | $RF[rt] \leftarrow RF[rs] ^ Z_Ext(imm)$ | Exclusive OR unsigned constant |
| sll \$rd, \$rt, sa | R | $RF[rd] \leftarrow RF[rt] << sa$ | Shift Left Logical |
| srl \$rd, \$rt, sa | R | $RF[rd] \leftarrow RF[rt] >> sa$ | Shift Right Logical |
| sra \$rd, \$rt, sa | R | $RF[rd] \leftarrow RF[rt] >> sa$ | Shift Right Arithmetic |
| sllv \$rd, \$rs, \$rt | R | $RF[rd] \leftarrow RF[rs] << RF[rt]$ | Shift Left Logical Variable |
| srlv \$rd, \$rs, \$rt | R | $RF[rd] \leftarrow RF[rs] >> RF[rt]$ | Shift Right Logical Variable |
| sraw \$rd, \$rs, \$rt | R | $RF[rd] \leftarrow RF[rs] >> RF[rt]$ | Shift Right Arithmetic Variable |
| | | ... | |



MIPS Instructions



| Assembly | Type | RTL Abstract | Instruction |
|---------------------|------|---|---|
| lw \$rt, imm(\$rs) | I | $RF[rt] \leftarrow M[RF[rs] + S_Ext(imm)]$ | Load Word |
| sw \$rt, imm(\$rs) | I | $M[RF[rs] + S_Ext(imm)] \leftarrow RF[rt]$ | Store Word |
| lb \$rt, imm(\$rs) | I | $RF[rt] \leftarrow S_Ext(M[RF[rs] + S_Ext(imm)])$ | Load Byte |
| sb \$rt, imm(\$rs) | I | $M[RF[rs] + S_Ext(imm)] \leftarrow S_Ext(RF[rt])$ | Store Byte |
| lui \$rt, imm | I | $RF[rt] \leftarrow imm \mid\mid 0x0000$ | Load Upper Immediate |
| beq \$rt, \$rs, imm | I | If($RF[rs] == RF[rt]$) then $PC \leftarrow PC + 4 + S_Ext(imm) \ll 2$ | Branch on equal |
| bne \$rs, \$rt, imm | I | If($RF[rs] != RF[rt]$) then $PC \leftarrow PC + 4 + S_Ext(imm) \ll 2$ | Branch on not equal |
| bgez \$rs, imm | I | If($RF[rs] \geq 0$) then $PC \leftarrow PC + 4 + S_Ext(imm) \ll 2$ | Branch on Greater Than or Equal to Zero |
| bltz \$rs, imm | I | If($RF[rs] < 0$) then $PC \leftarrow PC + 4 + S_Ext(imm) \ll 2$ | Branch on Less Than Zero |
| | | ... | |



MIPS Instructions



| Assembly | Type | RTL Abstract | Instruction |
|-----------------------|------|--|--|
| slt \$rd, \$rs, \$rt | R | If($RF[rs] < RF[rt]$) then $RF[rd] \leftarrow 1$ else $RF[rd] \leftarrow 0$ | Set on Less Than |
| slti \$rt, \$rs, imm | I | If($RF[rs] < S_Ext(imm)$) then $RF[rt] \leftarrow 1$ else $RF[rt] \leftarrow 0$ | Set on Less Than Immediate |
| sltu \$rd, \$rs, \$rt | R | If($RF[rs] < RF[rt]$) then $RF[rd] \leftarrow 1$ else $RF[rd] \leftarrow 0$ | Set on Less Than Unsigned |
| sltiu \$rt, \$rs, imm | I | If($RF[rs] < Z_Ext(imm)$) then $RF[rt] \leftarrow 1$ else $RF[rt] \leftarrow 0$ | Set on Less Than Immediate Unsigned |
| j target_address | J | $PC \leftarrow PC[31:28] \parallel target_address \parallel 00$ | Jump |
| jal target_address | J | $RF[31] \leftarrow PC + 4$ $PC \leftarrow PC[31:28] \parallel target_address \parallel 00$ | Jump And Link $RF[31]$ – return address |
| jr \$rs | R | $PC \leftarrow RF[rs]$ | Jump Register |
| | | ... | |

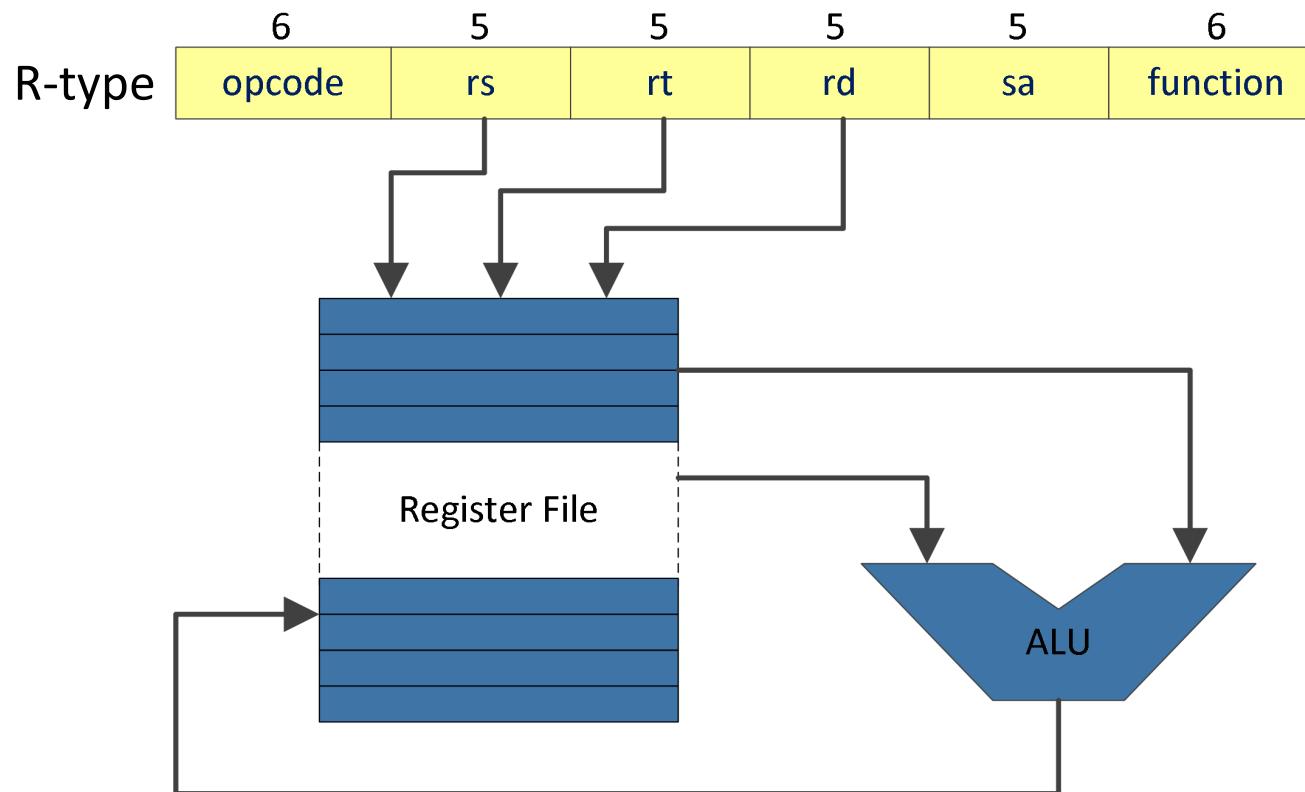


MIPS Instructions



- Arithmetic / Logical R-type Instructions

add \$rd, \$rs, \$rt → $RF[rd] \leftarrow RF[rs] + RF[rt]$



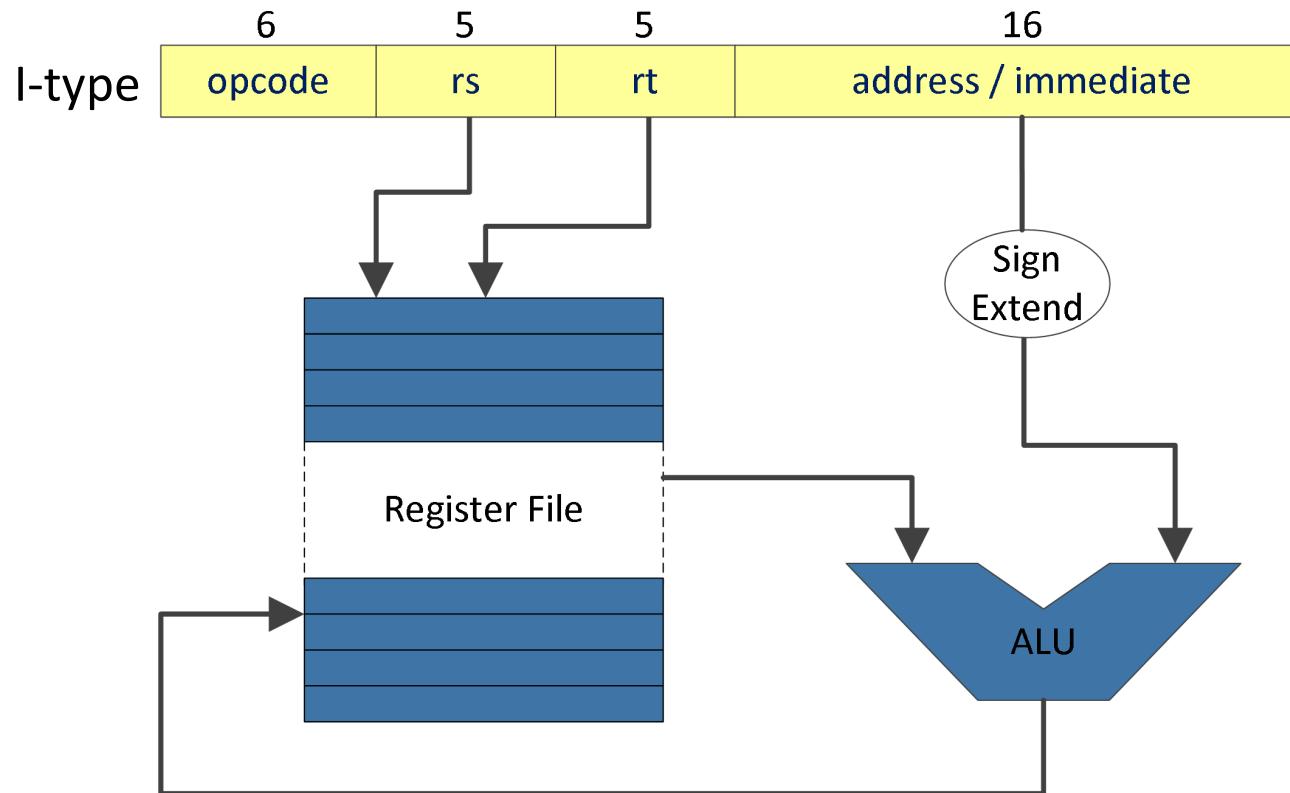


MIPS Instructions



- Immediate Arithmetic Instructions

$\text{addi } \$rt, \$rs, \text{imm} \rightarrow RF[rt] \leftarrow RF[rs] + S_Ext(\text{imm})$



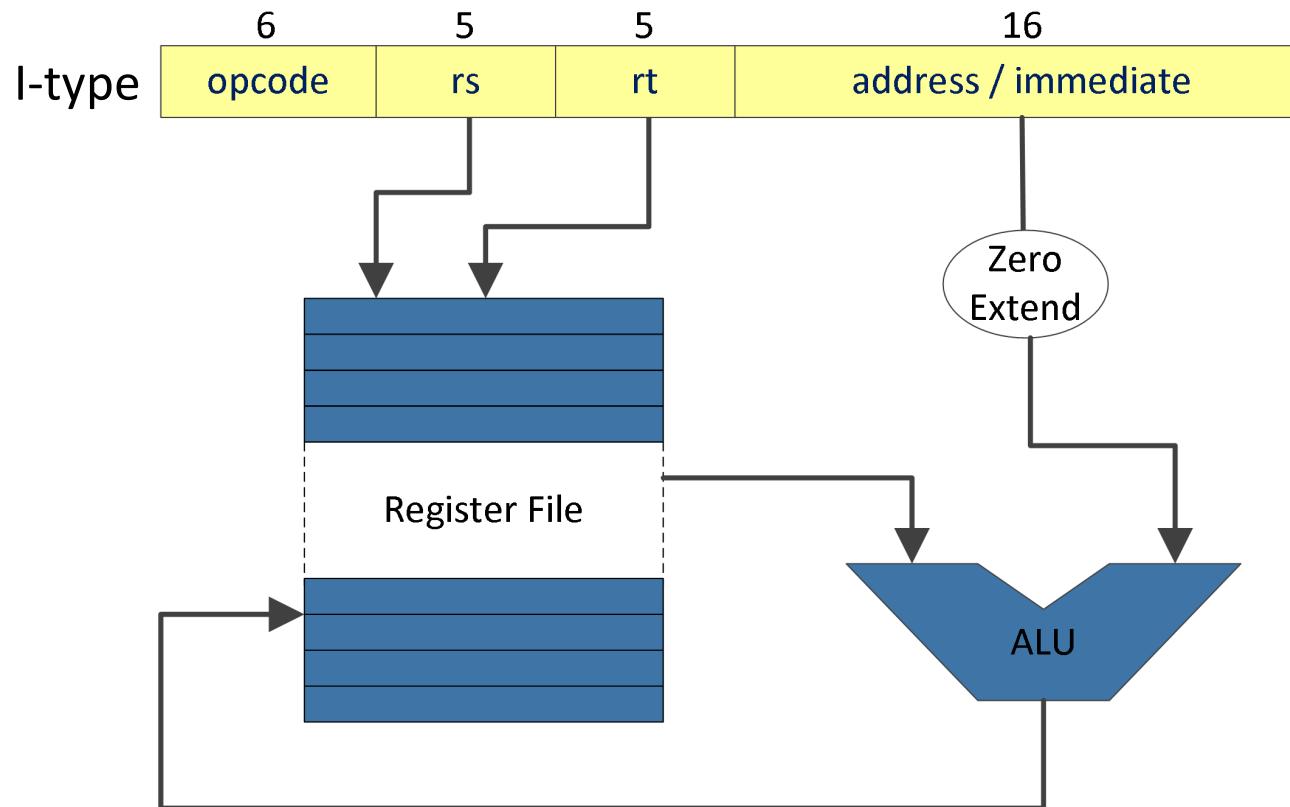


MIPS Instructions



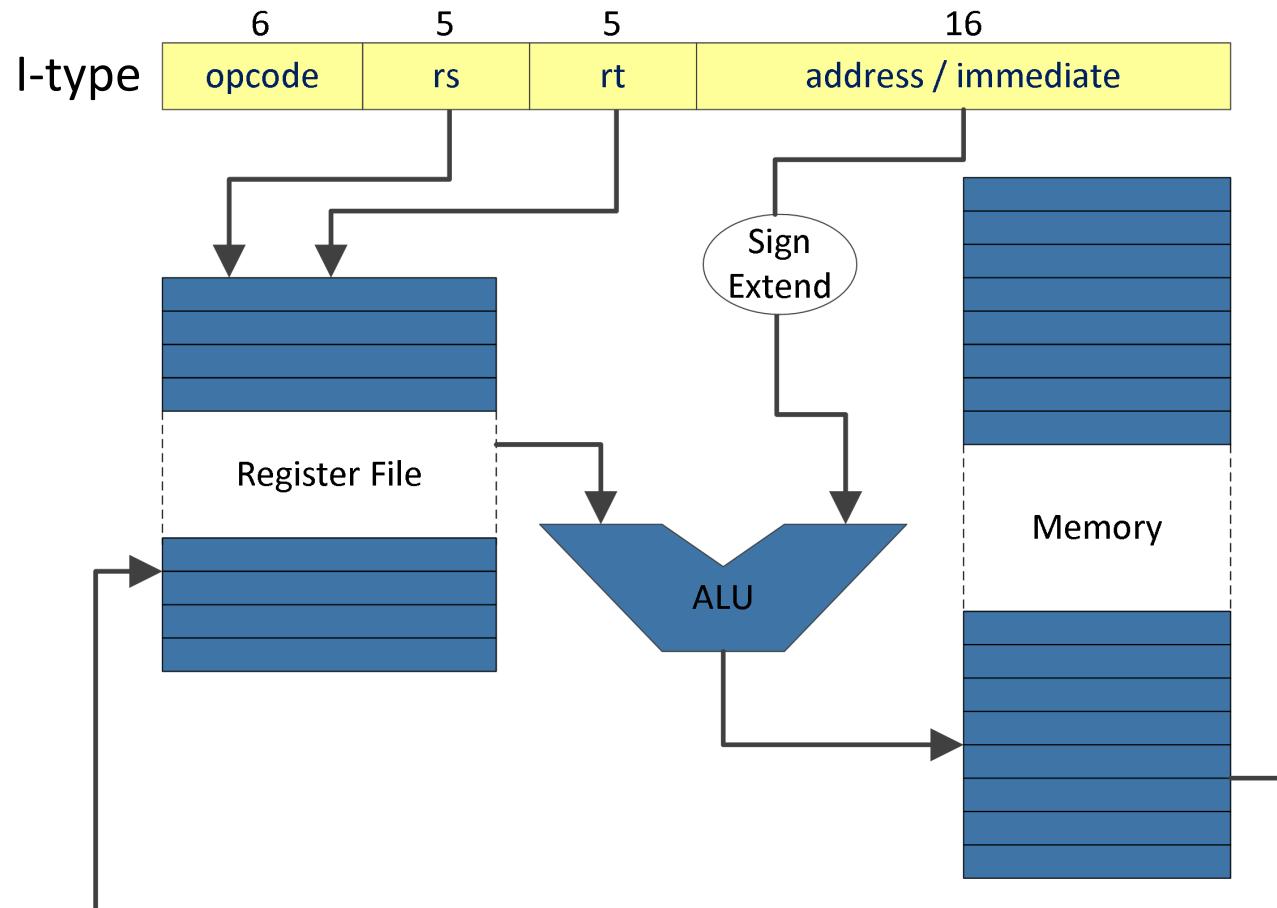
- Immediate Logical Instructions

andi \$rt, \$rs, imm → $RF[rt] \leftarrow RF[rs] \& Z_Ext(imm)$



- Load Word Instruction

$\text{lw } \$rt, \text{ imm}(\$rs) \rightarrow RF[rt] \leftarrow M[RF[rs] + S_Ext(imm)]$



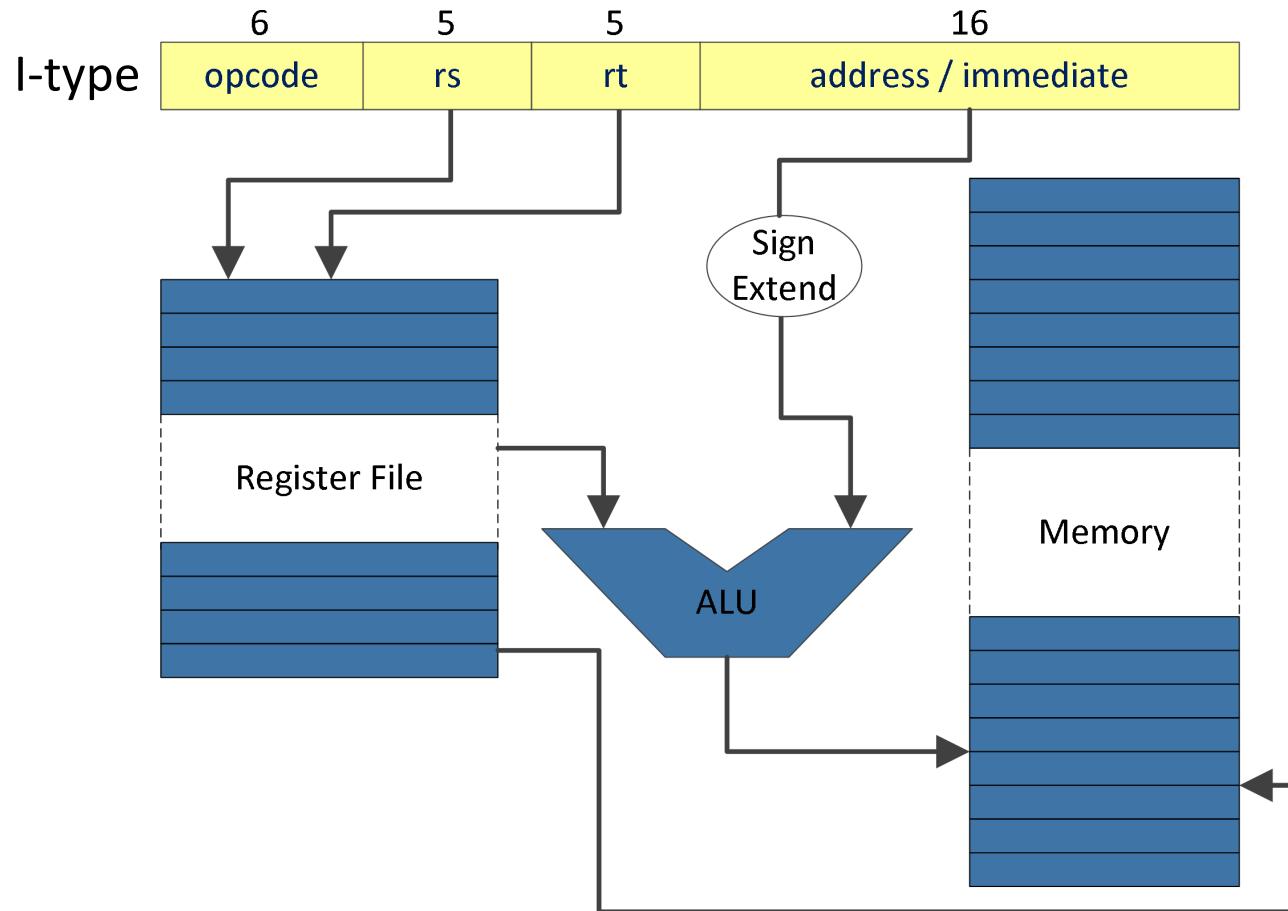


MIPS Instructions



- Store Word Instruction

$sw \$rt, imm(\$rs)$ → $M[RF[rs] + S_Ext(imm)] \leftarrow RF[rt]$



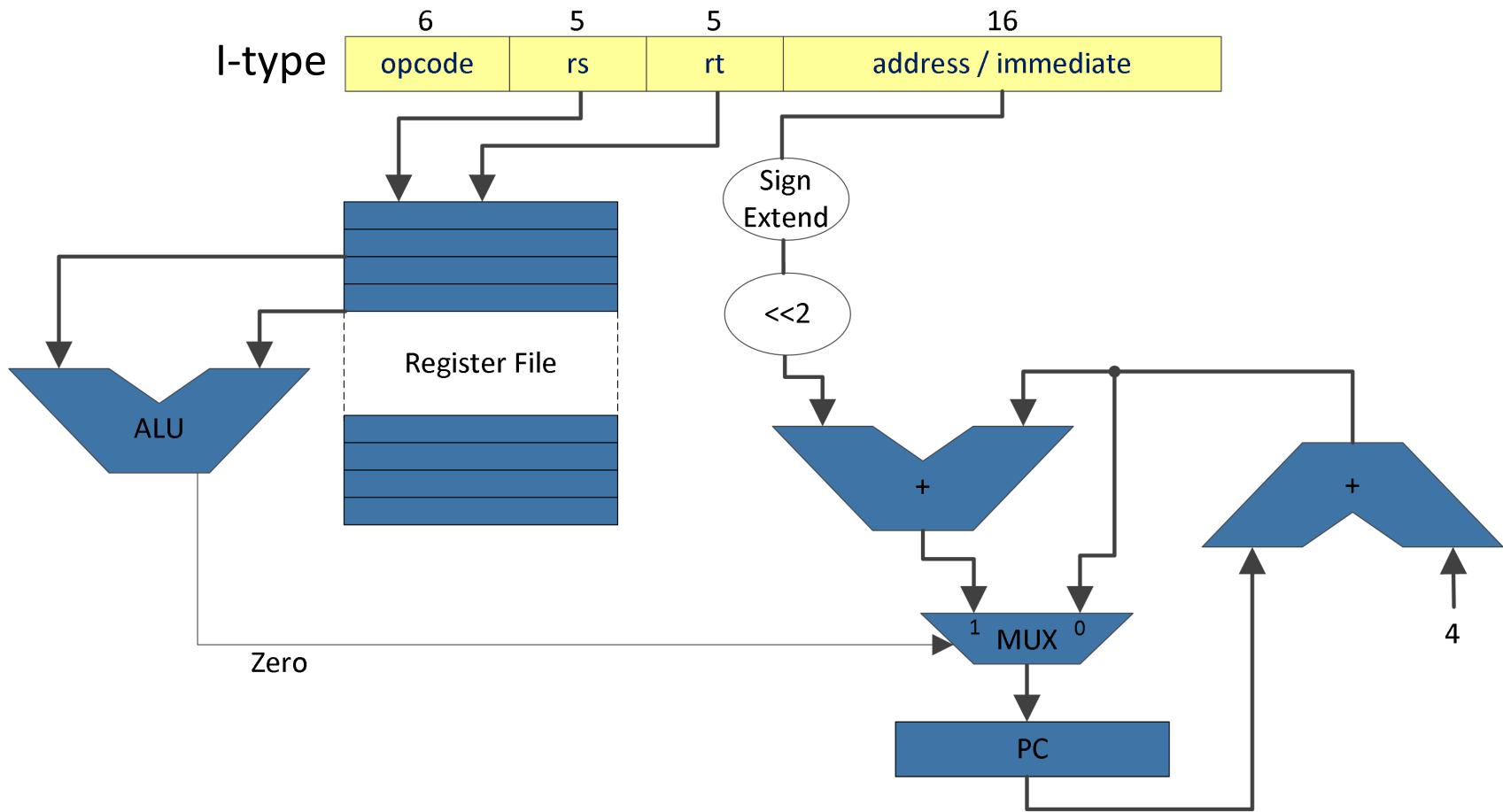


MIPS Instructions



- Branch on Equal Instruction

$\text{beq } \$rt, \$rs, \text{imm} \rightarrow \text{If}(RF[\text{rs}] == RF[\text{rt}]) \text{ then } PC \leftarrow PC + 4 + S_Ext(\text{imm}) \ll 2$



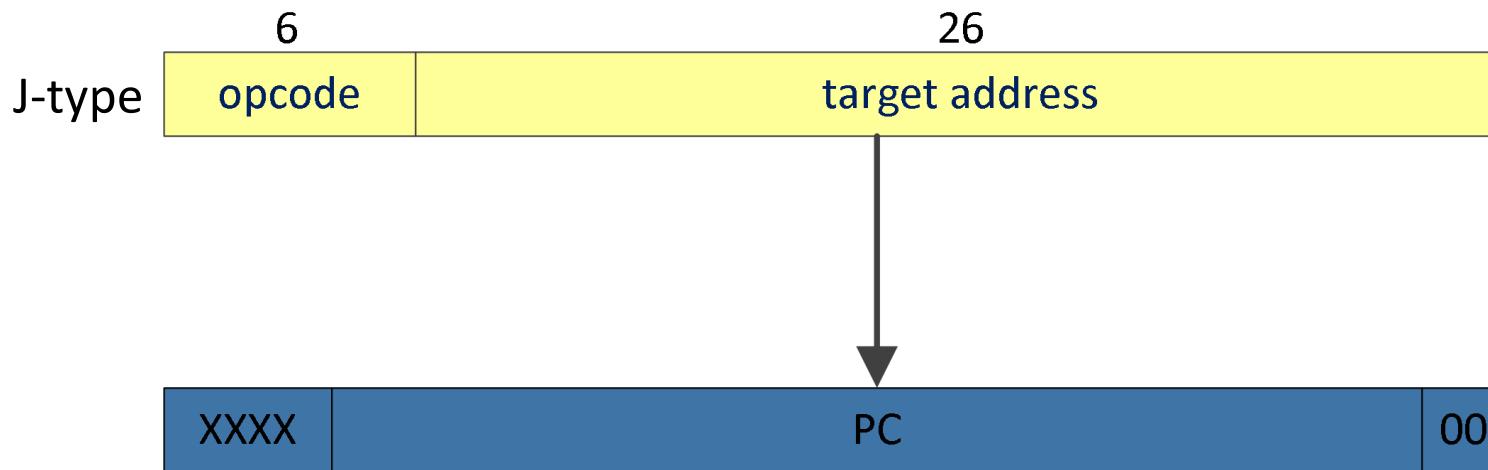


MIPS Instructions



- Jump Instruction

j target_address \rightarrow $PC \leftarrow PC[31:28] || target_address || 00$



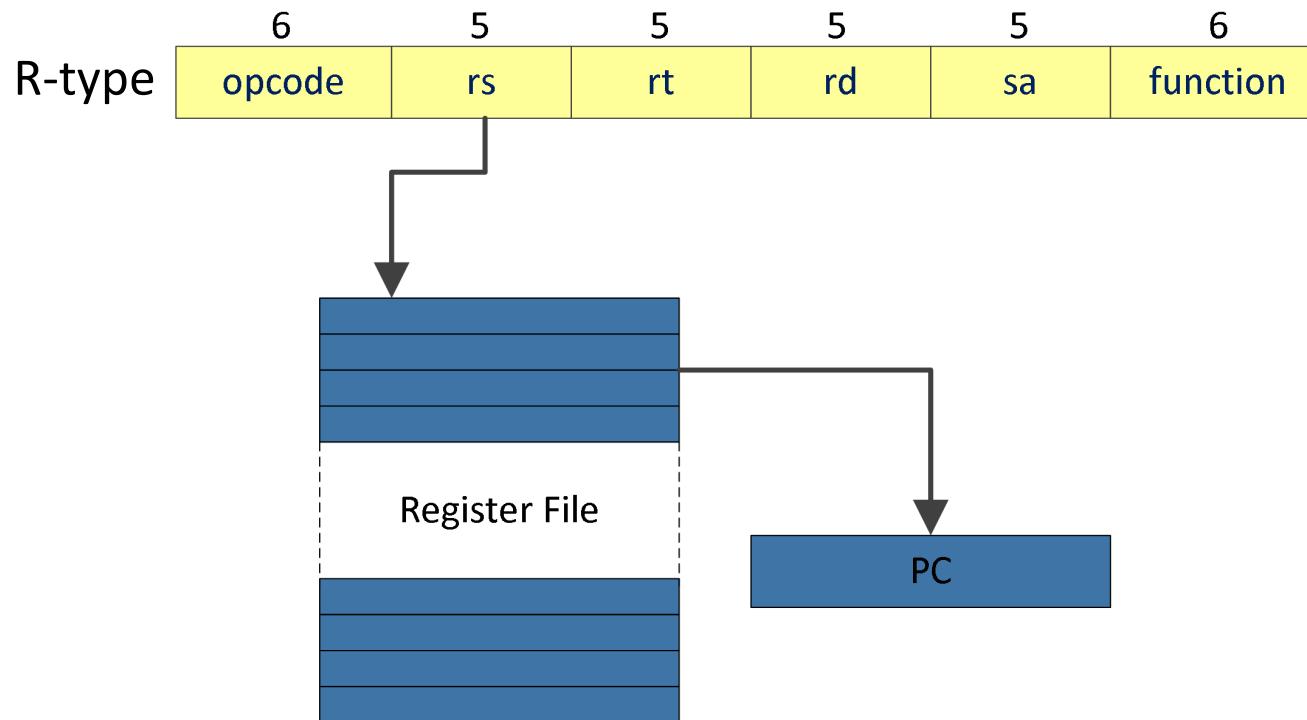


MIPS Instructions



- Jump Register Instruction

$\text{jr } \$rs \rightarrow \text{PC} \leftarrow \text{RF}[rs]$





Pseudo Instructions



- Not accepted by the processor
- Can be accepted by compiler and implemented by native processor instructions
- Examples:

| Absolute Value | | |
|-------------------------|---|---|
| abs \$rd, \$rs | addu \$rd, \$0, \$rs bgez \$rs, 1 sub \$rd, \$0, \$rs | slt \$ra, \$rs, \$0 bne \$ra, \$0, NEG add \$rd, \$rs, \$0 j DONE NEG: sub \$rd, \$0, \$rs DONE: |
| Branch on Equal to Zero | | |
| beqz \$rs, label | beq \$rs, \$0, label | |



Pseudo Instructions



| | | | |
|--|--|----------------|----------------------|
| Branch on Greater than or Equal | | Move | |
| bge \$rs, \$rt, label | slt \$at, \$rs, \$rt beq \$at, \$0, label | move \$rd,\$rs | addu \$rd, \$0, \$rs |
| Branch on Greater than or Equal Unsigned | | Negate | |
| bgeu \$rs, \$rt, label | sltu \$at, \$rs, \$rt beq \$at, \$0, label | neg \$rd, \$rs | sub \$rd, \$0, \$rs |
| Branch if Greater Than | | No Operation | |
| bgt \$rs, \$rt, label | slt \$at, \$rt, \$rs bne \$at, \$0, label | nop | or \$0, \$0, \$0 |
| | | nop | sll \$0, \$0, \$0 |
| Load Address | | | |
| la \$rd, label | lui \$at, Upper 16-bits of label ori Rd, \$at, Lower 16-bits of label | | |
| Load Immediate | | | |
| li \$rt, value | ori \$rt, \$0, value | | |



MIPS Interrupt Processing



- MIPS hardware interrupt processing → 3 steps
- Step 1: Save PC
 - EPC (Exception Program counter) gets a value equal to:
 - The address of the instruction that generates an exception (address error, reserved instruction) or hardware malfunction detected (memory parity error)
 - The address of the next instruction: external interrupts
- Step 2: PC gets new value and interrupt cause code is saved
 - $PC \leftarrow 8000\ 0180_{16}$ (the address depends on MIPS)
 - Cause Register \leftarrow interrupt code
 - Each interrupt has its code, e.g.:
 - hardware interrupt = 0
 - illegal memory address (load/fetch or store) = 4 or 5
 - bus error (fetch or load/store)= 6 or 7
 - syscall instruction execution = 8
 - illegal op-code, i.e. reserved or undefined op-code= 10
 - integer overflow = 12
 - any floating point exception = 15
- Step3: CPU Mode operation set to kernel mode
 - CPU Mode bit $\leftarrow 0$



Problems – Homework



- What is the address space of a MIPS processor (what is the total memory size that it can address)
- Using real MIPS instructions, write a pseudo instruction that computes the following:
 - Absolute value.
 - Branch on Greater Than
 - Branch on Less or Equal
 - Swap two registers
- Define new MIPS instructions:
 - LWR (load word register) – sums two registers to obtain the memory address.
 - SWR (store word register) – sums two registers to obtain the memory address.
 - LWA – uses a single register to obtain the memory address.
 - SWA – uses a single register to obtain the memory address.
 - Identify the instruction formats, write the RTL abstract and draw the processing diagram for each new instruction.



References



1. D. A. Patterson, J. L. Hennessy, “Computer Organization and Design: The Hardware/Software Interface”, 5th edition, ed. Morgan–Kaufmann, 2013.
2. D. A. Patterson and J. L. Hennessy, “Computer Organization and Design: A Quantitative Approach”, 5th edition, ed. Morgan-Kaufmann, 2011.
3. MIPS32™ Architecture for Programmers, Volume I: “Introduction to the MIPS32™ Architecture”.
4. MIPS32™ Architecture for Programmers Volume II: “The MIPS32™ Instruction Set”.



Computer Architecture

Lecturer: Mihai Negru

2nd Year, Computer Science

Lecture 4: Single-Cycle CPU Design

<http://users.utcluj.ro/~negrum/>



Processor Design Phases

1. Analyze instruction set → data-path requirements
 - Write the micro-operation sequences for the target ISA
 - RTL statements specify the data-path components and their interconnection
2. Select a set of data-path components and establish clocking methodology
 - Define when storage or state elements can be read and when they can be written, e.g. clock edge-triggered
 - Find the worst-time propagation delay in the data-path to determine the data-path clock cycle (CPU clock cycle)
3. Assemble data-path meeting the requirements
 - Create an initial data-path (i.e. registers, ALU, memories)
 - Establish the connectivity requirements
 - Whenever multiple sources are connected to a single input (or destination), a multiplexer of appropriate size is added
 - Complete the micro-operation sequences for all remaining instructions adding data path components + connections/multiplexers as needed



Processor Design Phases

4. Identify and define the function of all control points or signals needed by the data-path
 - For each instruction from the target ISA identify the values of the control signals that affect the register transfers
5. Assemble the control logic
 - Design the control unit based on the identified control signals
 - 3 main types of control unit
 - Combinational logic → single cycle CPU (Any instruction completed in one cycle)
 - Hard-Wired: Finite-state machine implementation
 - Micro-programmed
 - MIPS Single-Cycle CPU Design adapted from [1]



Single-Cycle CPU Design – Step 1



- Design Step 1: MIPS-Lite Subset
 - Select a number of representative target instructions

| Instruction | RTL Abstract | Program Counter |
|----------------------|---|--|
| add \$rd, \$rs, \$rt | $RF[rd] \leftarrow RF[rs] + RF[rt]$ | $PC \leftarrow PC + 4$ |
| sub \$rd, \$rs, \$rt | $RF[rd] \leftarrow RF[rs] - RF[rt]$ | $PC \leftarrow PC + 4$ |
| ori \$rt, \$rs, imm | $RF[rt] \leftarrow RF[rs] Z_Ext(imm)$ | $PC \leftarrow PC + 4$ |
| lw \$rt, imm(\$rs) | $RF[rt] \leftarrow M[RF[rs] + S_Ext(imm)]$ | $PC \leftarrow PC + 4$ |
| sw \$rt, imm(\$rs) | $M[RF[rs] + S_Ext(imm)] \leftarrow RF[rt]$ | $PC \leftarrow PC + 4$ |
| beq \$rt, \$rs, imm | If($RF[rs] == RF[rt]$) then | $PC \leftarrow PC + 4 + S_Ext(imm) \ll 2$ |
| | else | $PC \leftarrow PC + 4$ |

- RTL Abstract defines the behavior of each instruction
- Remember the instruction execution cycle (previous lecture)
 - IF, ID/OF, EX, MEM, WB
 - IF, ID and OF are common for all instructions

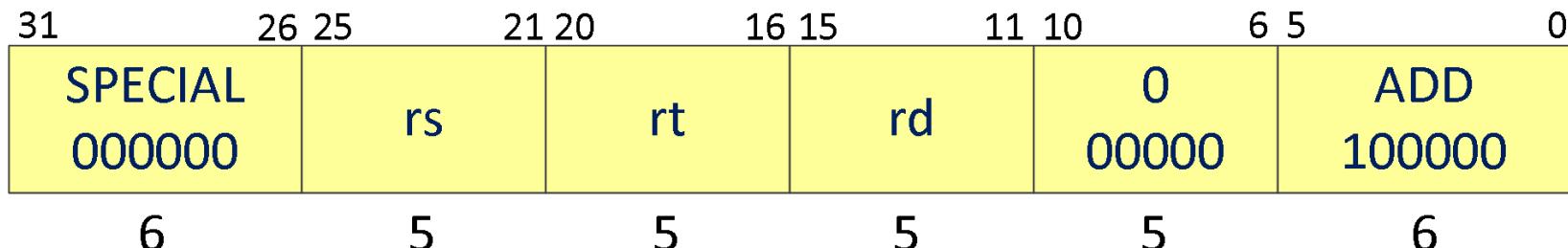


Single-Cycle CPU Design – Step 1



- R-type Instructions

- Basic operation: $RF[rd] \leftarrow RF[rs] \text{ op } RF[rt]$
- Next instruction PC: $PC \leftarrow PC + 4$
- OPCODE is always Zero for R-type Instructions



- ADD \$rd, \$rs, \$rt

- $RF[rd] \leftarrow RF[rs] + RF[rt]$
- $PC \leftarrow PC + 4$
- Add signed 32-bit numbers.
- Exception on OVERFLOW
- Addressing Modes
 - Register direct

| Necessary Resources | |
|---------------------|----------------------------------|
| IF | PC, Instr. Memory, Adder |
| ID/OF | Register File, Main Control Unit |
| EX | ALU, ALU Control Unit |
| MEM | No operation |
| WB | Register File |

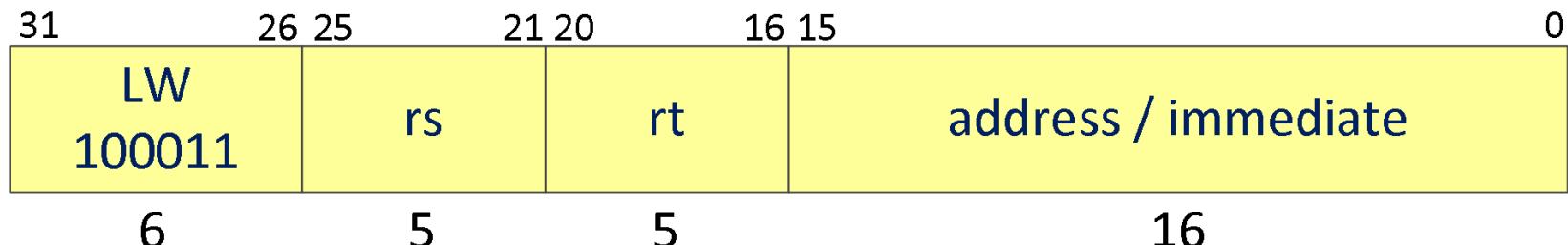


Single-Cycle CPU Design – Step 1



- I-type Instructions: Load Word – LW

- Load a word (32 bits) from the Data Memory into a Register



- Iw \$rt, imm(\$rs)**

- $RF[rt] \leftarrow M[RF[rs] + S_Ext(imm)]$

- $PC \leftarrow PC + 4$

- Addressing Modes
 - Register Direct
 - Base addressing

| Necessary Resources | |
|---------------------|--|
| IF | PC, Instr. Memory, Adder |
| ID/OF | Register File, Main Control Unit, Extender |
| EX | ALU, ALU Control Unit |
| MEM | Data Memory |
| WB | Register File |

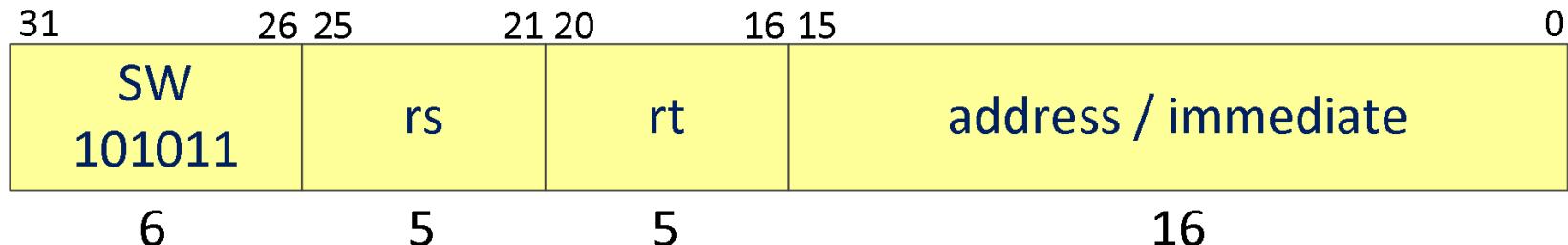
A blue curved arrow labeled "MUX" points from the MEM row to the WB row.



Single-Cycle CPU Design – Step 1



- I-type Instructions: Store Word – SW
 - Store a word (32-bits) from the Register File in the Data Memory



- $sw \$rt, imm(\$rs)$
 - $M[RF[rs] + S_Ext(imm)] \leftarrow RF[rt]$
 - $PC \leftarrow PC + 4$
 - Addressing Modes
 - Register Direct
 - Base addressing

| Necessary Resources | |
|---------------------|--|
| IF | PC, Instr. Memory, Adder |
| ID/OF | Register File, Main Control Unit, Extender |
| EX | ALU, ALU Control Unit |
| MEM | Data Memory |
| WB | No operation |



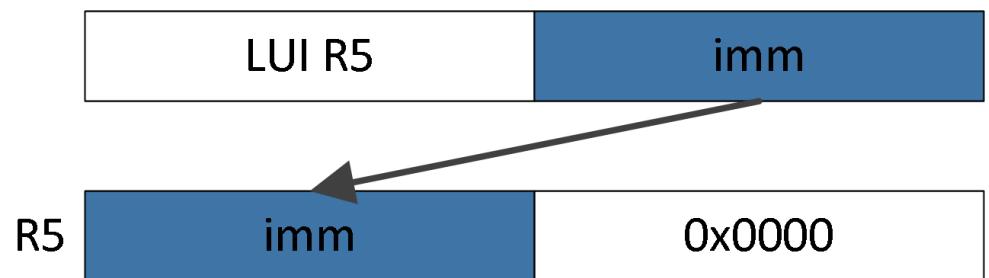
Single-Cycle CPU Design – Step 1



- I-type Instructions: Load Upper Immediate – LUI
 - Load a constant in high part of a word



- lui \$rt, imm
 - $RF[rt] \leftarrow imm \mid\mid 0x0000$
 - $PC \leftarrow PC + 4$
 - Addressing Modes
 - Register direct
 - Used to form 32 bits constants with ORI
 - Additional Resources: shifter implemented in the ALU

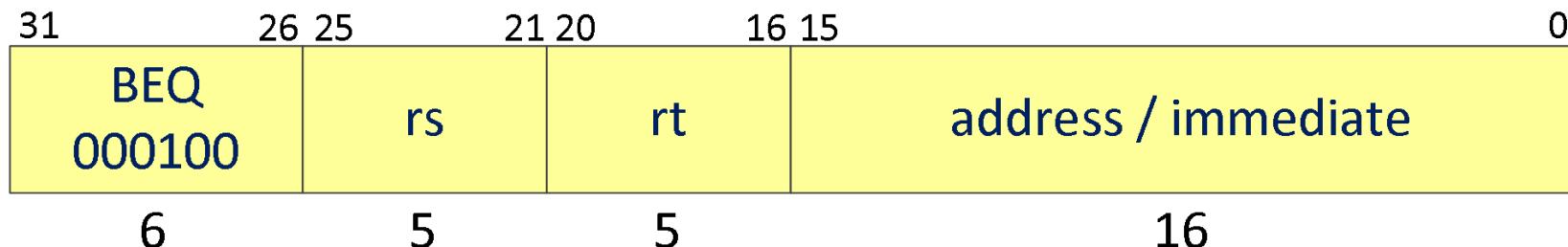




Single-Cycle CPU Design – Step 1



- I-type Instructions: Branch on Equal – BEQ
 - Compare two registers, then perform a conditional jump relative to the PC



- beq \$rt, \$rs, imm
 - If($RF[rs] == RF[rt]$) $\rightarrow PC \leftarrow PC + 4 + S_Ext(imm) << 2$ else $PC \leftarrow PC + 4$
 - Addressing Modes
 - PC-relative addressing
 - If condition is not true
 - Sequential execution, + 4
 - If condition is true
 - Jump, $PC + 4 + S_Ext(imm) << 2$

| Necessary Resources | |
|---------------------|---|
| IF | PC, Instr. Memory, Adder, Adder, MUX |
| ID/OF | Register File, Main Control Unit, Extender |
| EX | ALU, ALU Control Unit |
| MEM | No operation |
| WB | No operation |



Single-Cycle CPU Design – Step 1



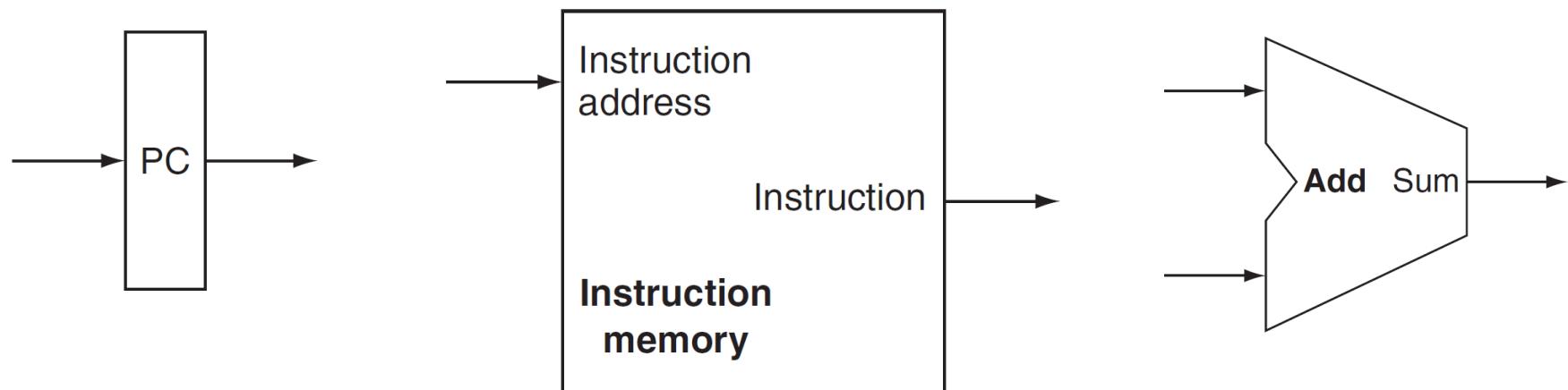
- Needed Resources (so far)
 - PC – Program Counter
 - Memories
 - Instruction and Data Memory
 - Register File (32 x 32 bits)
 - Read R[rs], Read R[rt]
 - Write R[rd] or R[rt]  **MUX**
 - Sign / Zero Extender for address / immediate field
 - Shift Left 2
 - ALU – Arithmetic Logic Unit  **MUX**
 - Arithmetic or Logical operations with two registers
 - Arithmetic or Logical operations with one register and an extended immediate value
 - Add PC with 4 or with 4 + Sign Extended Immediate << 2 for next instructions address (PC) computation  **MUX**



Single-Cycle CPU Design – Step 2

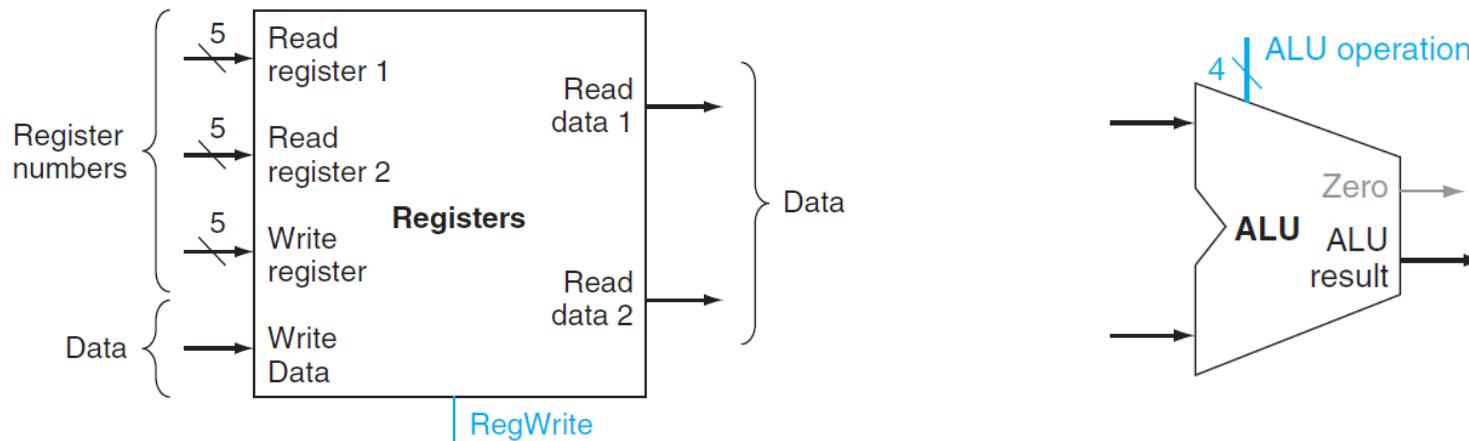


- Design Step 2: Data-Path Components



[1]

- Program Counter – PC
 - 32-bit positive edge triggered D flip-flop
- Instruction Memory (ideal ROM model)
 - One input bus: Instruction address
 - One output bus: Instruction
 - Memory word is selected by Instruction address, no control signals
- Adder
 - 32-bit Ripple Carry Adder to form the next instruction address



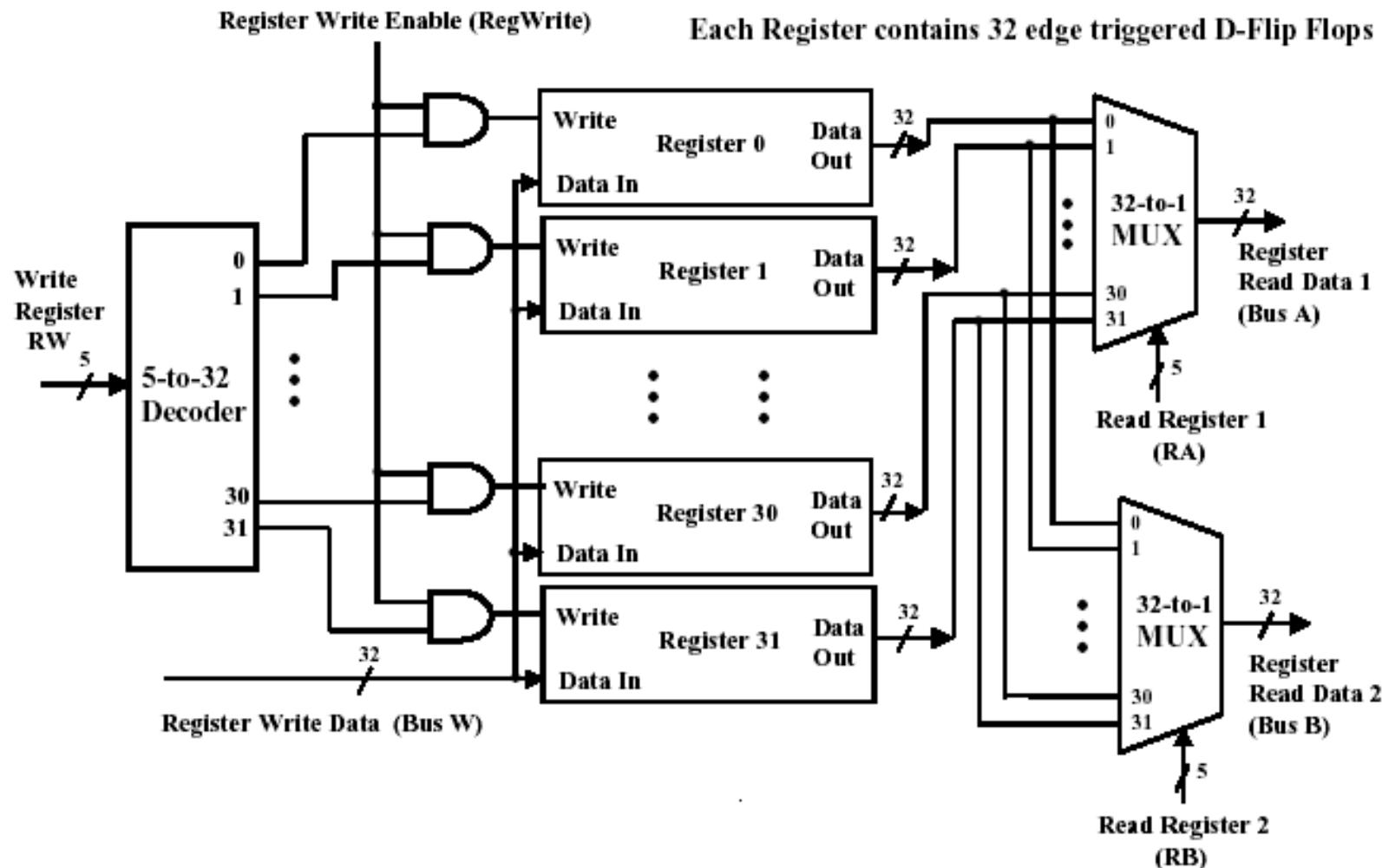
[1]

- **Register File 32x32-bits**

- Built using D flip-flops (didactic model), SRAM in real machines
- Two 32-bit data outputs: Read data 1 and Read data 2
- One 32-bit data input: Write data
- **Multi-access: 2 asynchronous Reads + 1 edge triggered Write in the same clock period**
- Read register 1 selects the register to put on Read data 1 output
- Read register 2 selects the register to put on Read data 2 output
- Write register selects the register to be written by Write Data when RegWrite is asserted
- During read operation, Registers behaves as a combinational logic block

- **Arithmetic Logic Unit – ALU**

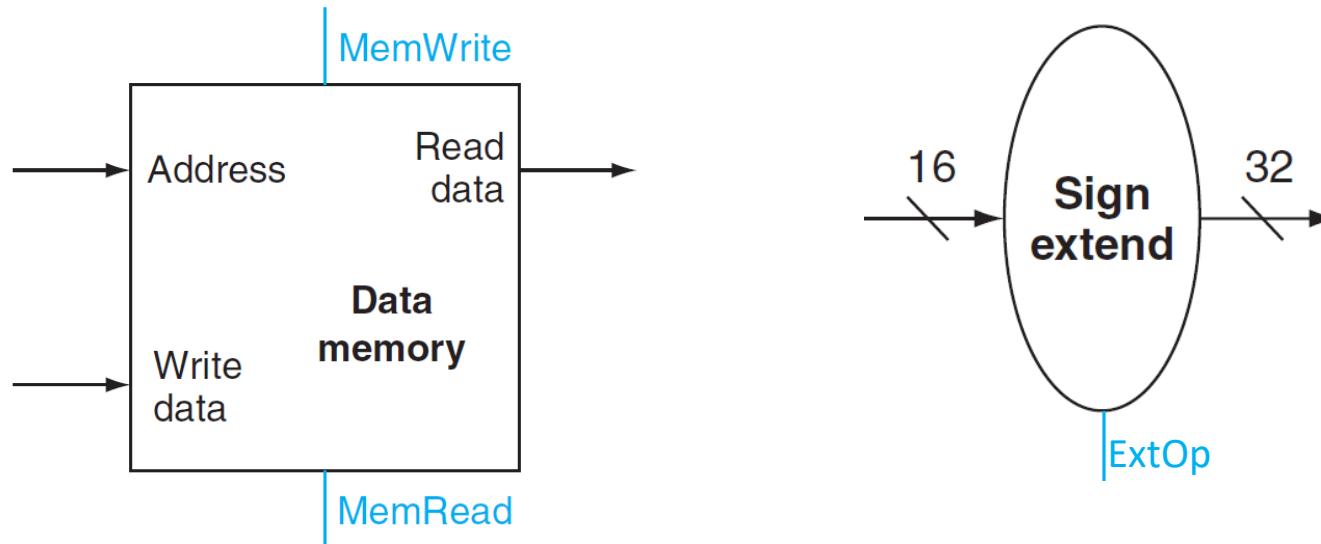
- Designed according to ISA requirements



Register File Implementation



Single-Cycle CPU Design – Step 2



[1]

- **Data Memory (ideal SRAM model)**
 - Two input buses: Address and Write data
 - One output bus: Read data
 - Two control signals: MemRead and MemWrite
- **Sign Extension Unit**
 - The control signal will only be added later
 - $\text{ExtOp} = 1 \rightarrow \text{Sign Extender}$
 - $\text{ExtOp} = 0 \rightarrow \text{Zero Extender}$



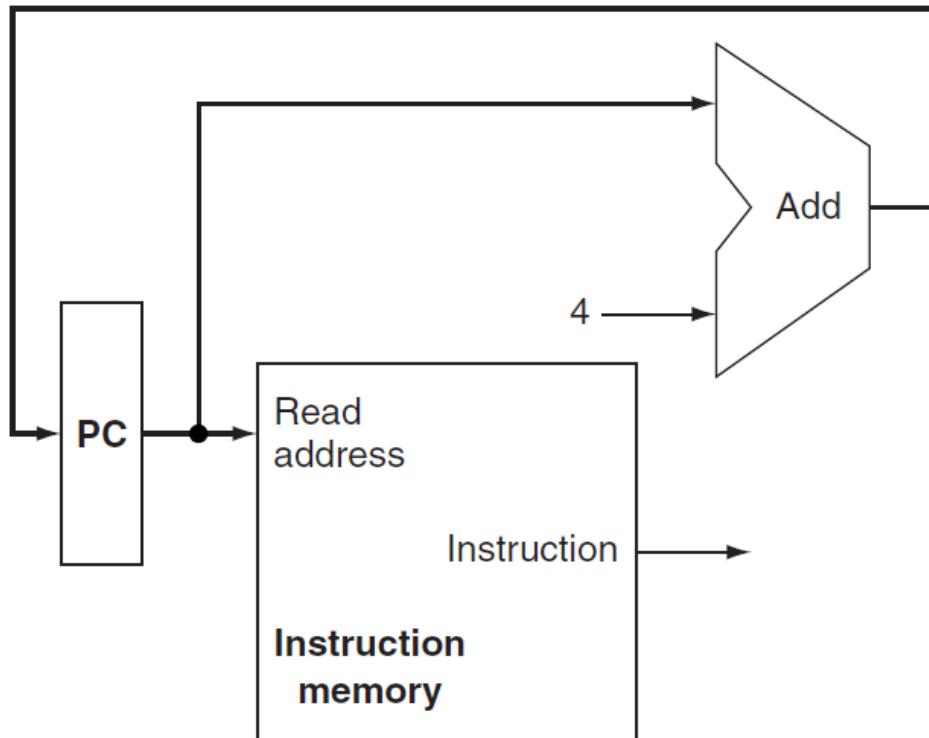
Single-Cycle CPU Design – Step 2



- Design Step 2: Clocking Methodology
 - Clocking methodology defines when signals can be read and written
 - Determines when data is valid and stable relative to the clock
- Clocking alternatives
 - Falling edge triggered system
 - Rising edge triggered system
 - Two phase clocking
- All storage elements (e.g. Flip-Flops, Registers, Data Memory) writes are triggered by the same clock edge.
 - Usually, State elements are written on every clock cycle
 - If not, we need an explicit **write control signal**.
 - Write only when both the write control is asserted and the clock edge occurs



- Design Step 3: Assemble Data-Path – Single-Cycle



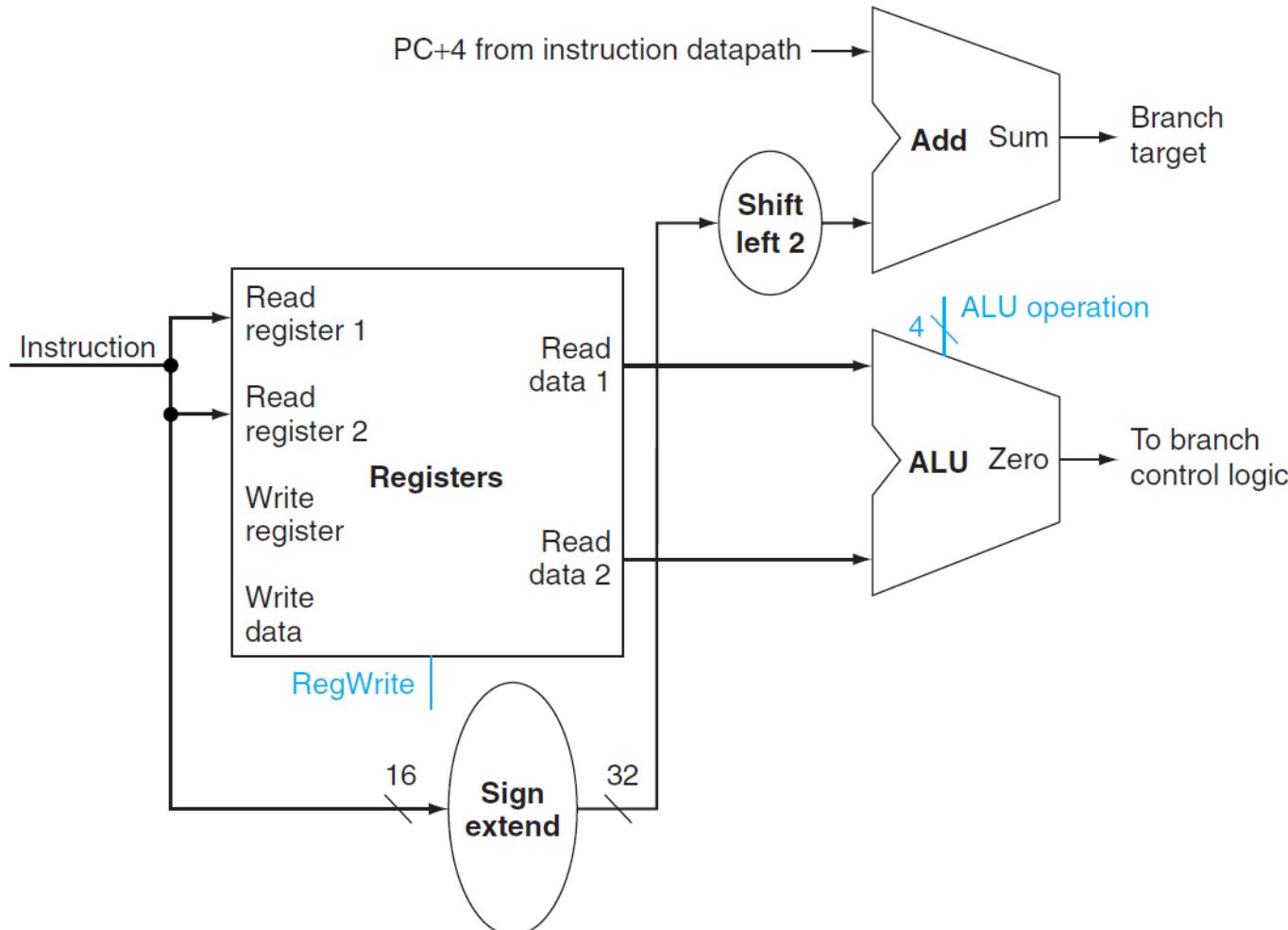
[1]

Instruction Fetch

- 32-bit Program Counter, 32-bit Adder and instruction Memory
- $\text{Instruction} \leftarrow \text{IM}[\text{PC}]$; $\text{PC} \leftarrow \text{PC} + 4$



Single-Cycle CPU Design – Step 3



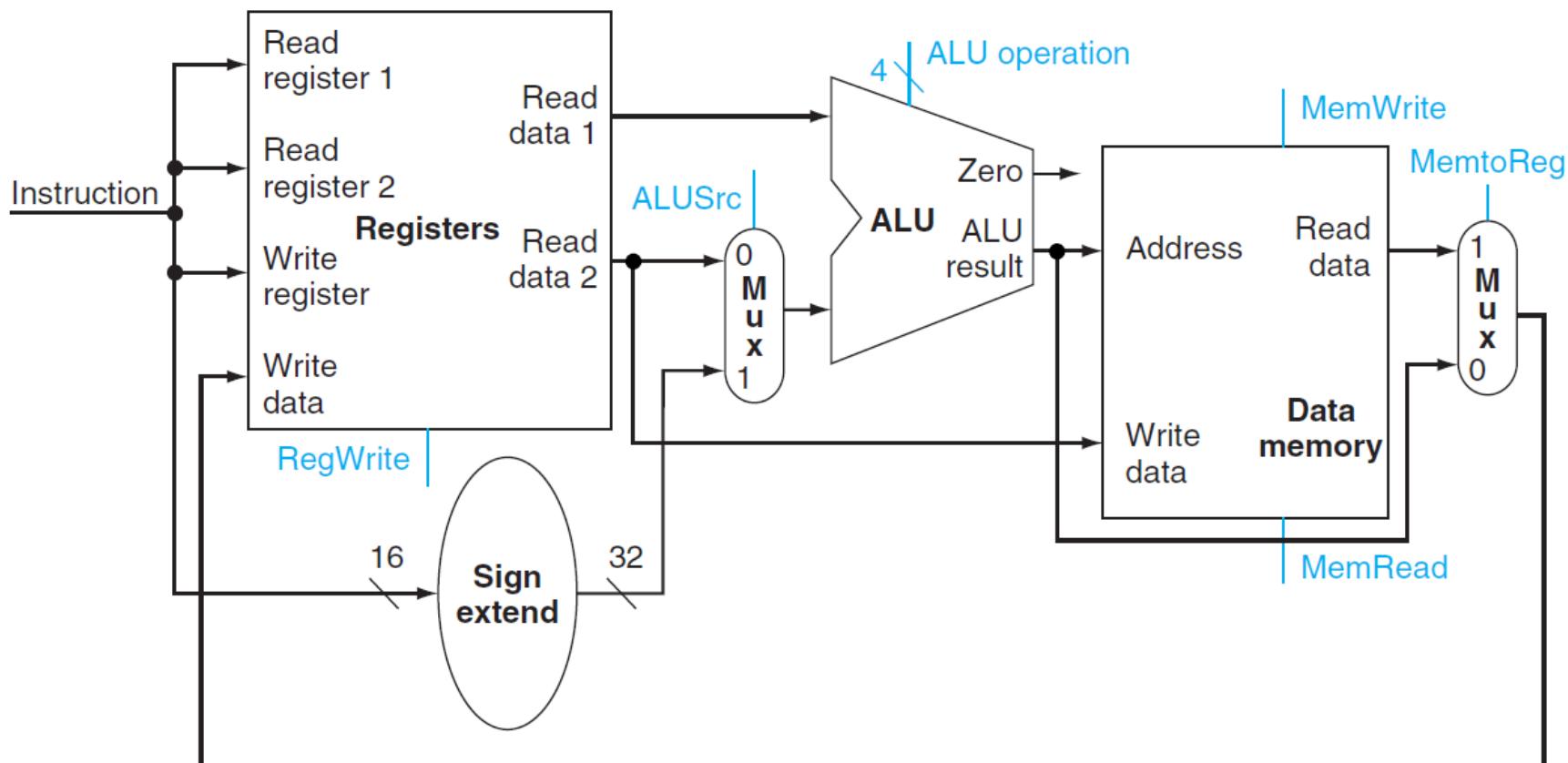
[1]

Branch Instruction:

If($RF[rs] == RF[rt]$) then
else

$PC \leftarrow PC + 4 + S_Ext(imm) \ll 2$
 $PC \leftarrow PC + 4$

Single-Cycle CPU Design – Step 3



[1]

R-type Instructions:

$$RF[rd] \leftarrow RF[rs] \text{ op } RF[rt]$$

I-type Instruction – Load:

$$RF[rt] \leftarrow M[RF[rs] + S_Ext(imm)]$$

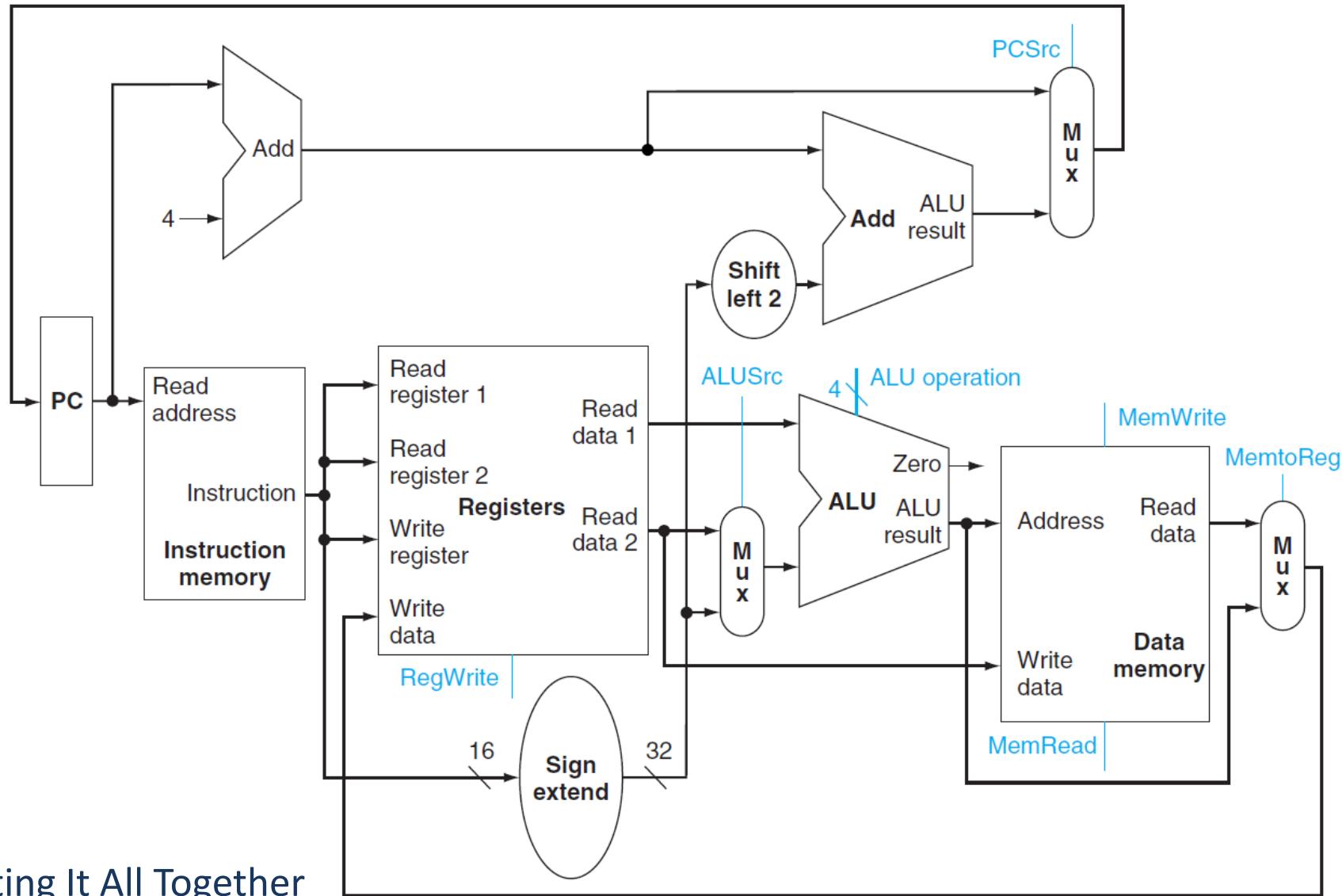
I-type Instruction – Store:

$$M[RF[rs] + S_Ext(imm)] \leftarrow RF[rt]$$

.....



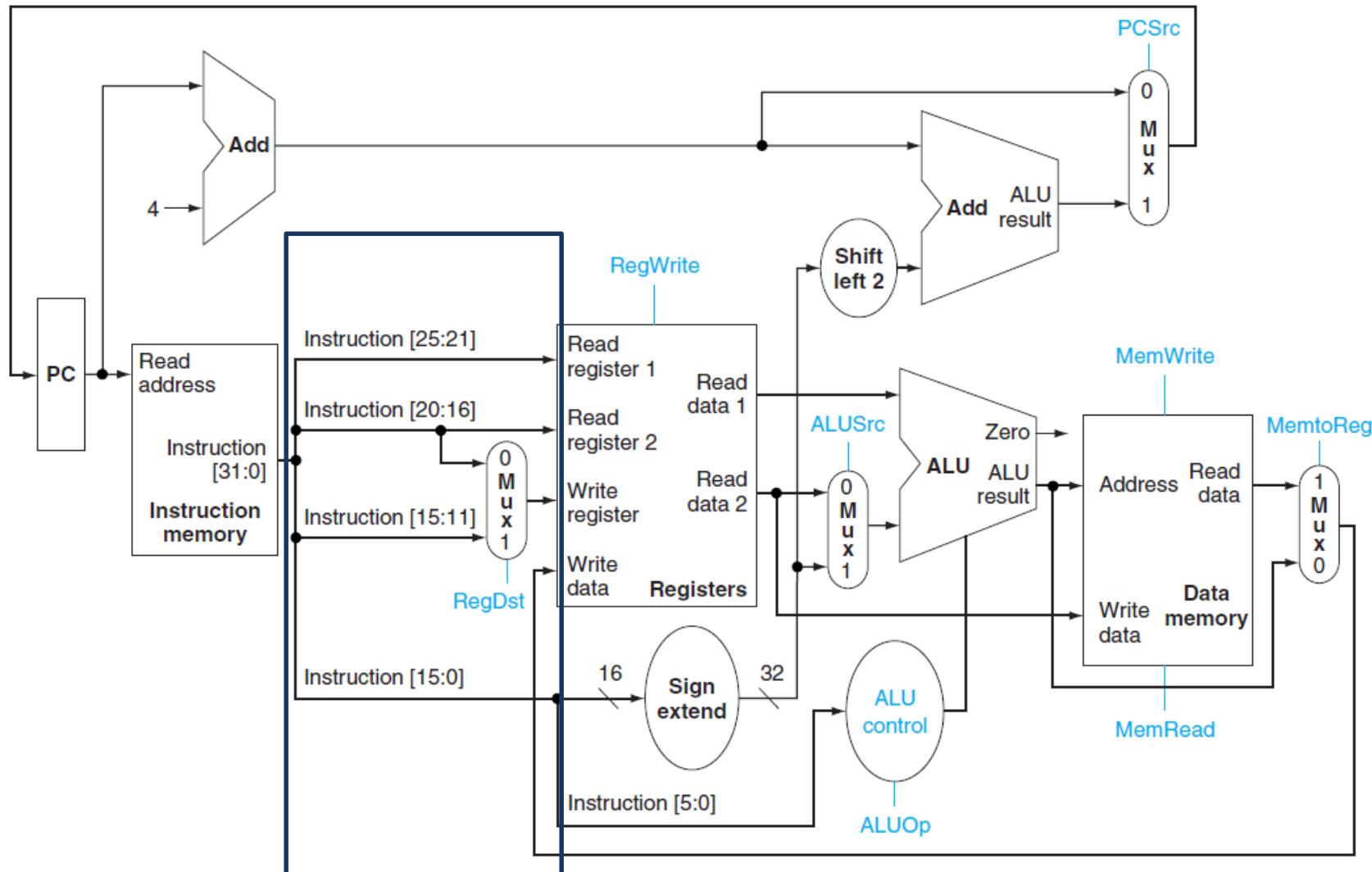
Single-Cycle CPU Design – Step 3



Putting It All Together

[1]

Single-Cycle CPU Design – Step 3

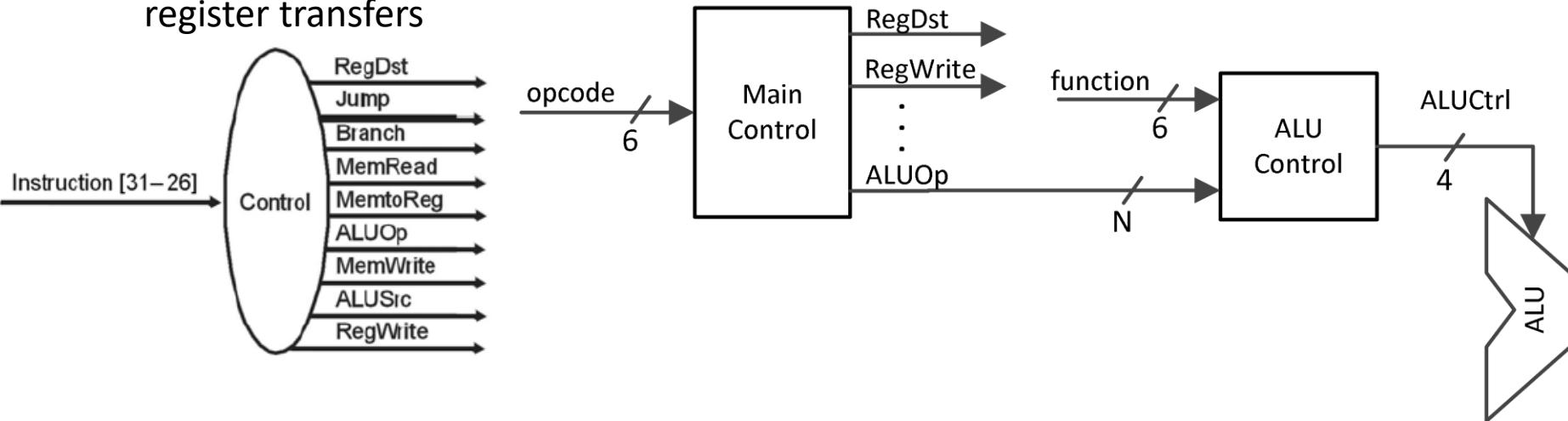


Single-Cycle Data-Path with Control Signals for MIPS-lite

[1]



- Design Step 4: Identifying the Control Signals
 - Identify and define the function of all control signals needed by the data-path
 - Analyze each instruction to determine the setting of control points that affect the register transfers



The main control signal values for MIPS-lite

| Instruction | Reg Dst | Reg Write | ALU Src | PC Src | Mem Read | Mem Write | Memto Reg | ALU Op |
|-------------|---------|-----------|---------|--------|----------|-----------|-----------|--------|
| R-format | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 10 |
| lw | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 00 |
| sw | X | 0 | 1 | 0 | 0 | 1 | X | 00 |
| beq | X | 0 | 0 | 1 | 0 | 0 | X | 01 |



Single-Cycle CPU Design – Step 4



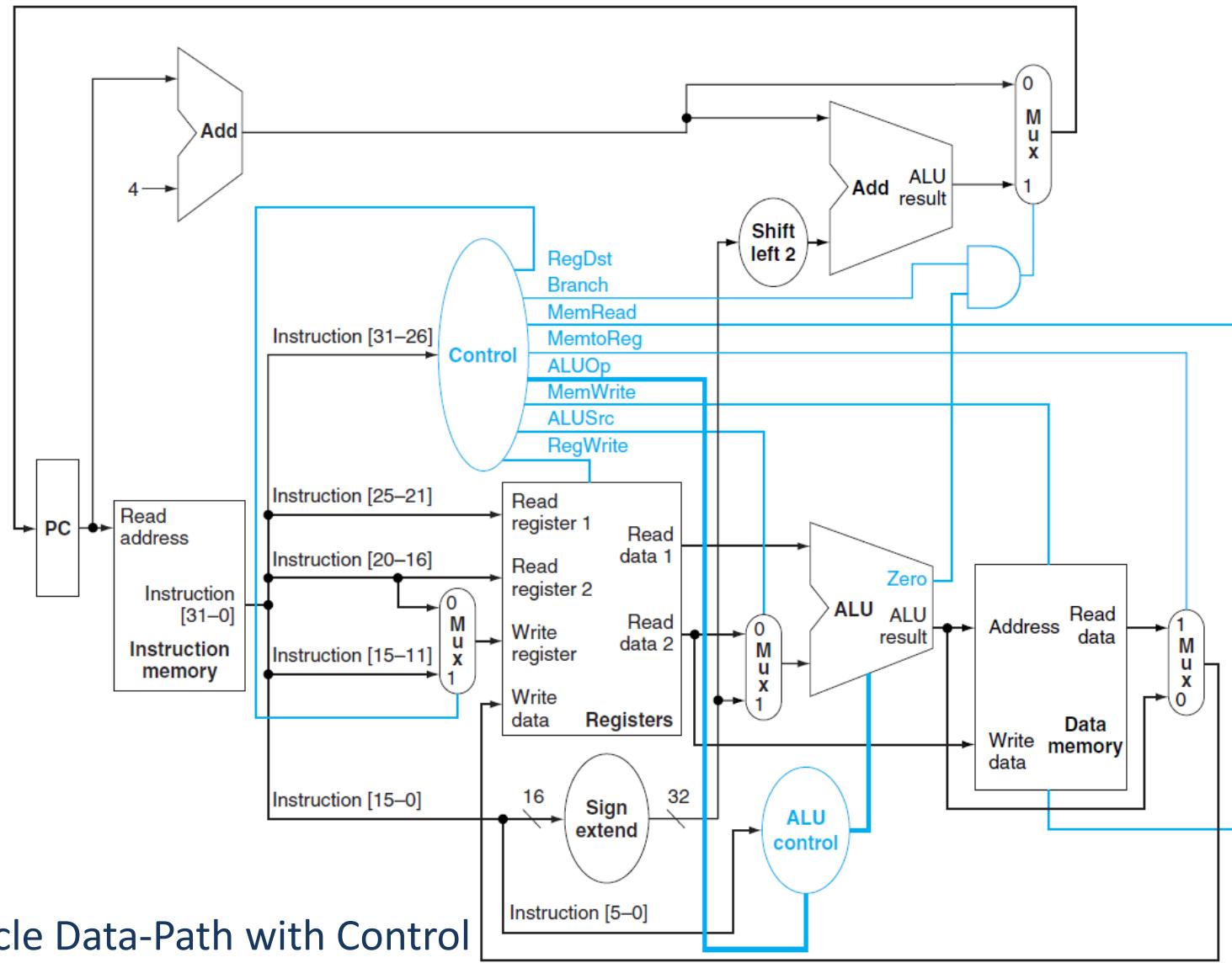
| Signal name | Effect when deasserted (0) | Effect when asserted (1) |
|-------------|---|---|
| RegDst | The register destination number for the Write register comes from the rt field | The register destination number for the Write register comes from the rd field |
| RegWrite | None | The register on the Write register input is written into with the value on the Write data input |
| ALUSrc | The second ALU operand comes from the second Register file output | The second ALU operand is the sign-extended lower 16-bits of the instruction |
| PCSrc | The PC is replaced by the output of the adder that computes the value of PC + 4 | The PC is replaced by the output of the adder that computes the branch address |
| MemRead | None | Data memory contents at the read address are put on read data output |
| MemWrite | None | Data memory contents at address given by write address is replaced by value on write data input |
| MemtoReg | The value fed to the register write data input comes from the ALU | The value fed to the register write data input comes from the data memory |

The Meaning of the Control Signals

ALUOp – defines the behavior of the ALU control

PCSrc = Branch AND Zero

Single-Cycle CPU Design – Step 4



Single-Cycle Data-Path with Control

[1]



Single-Cycle CPU Design – Step 4

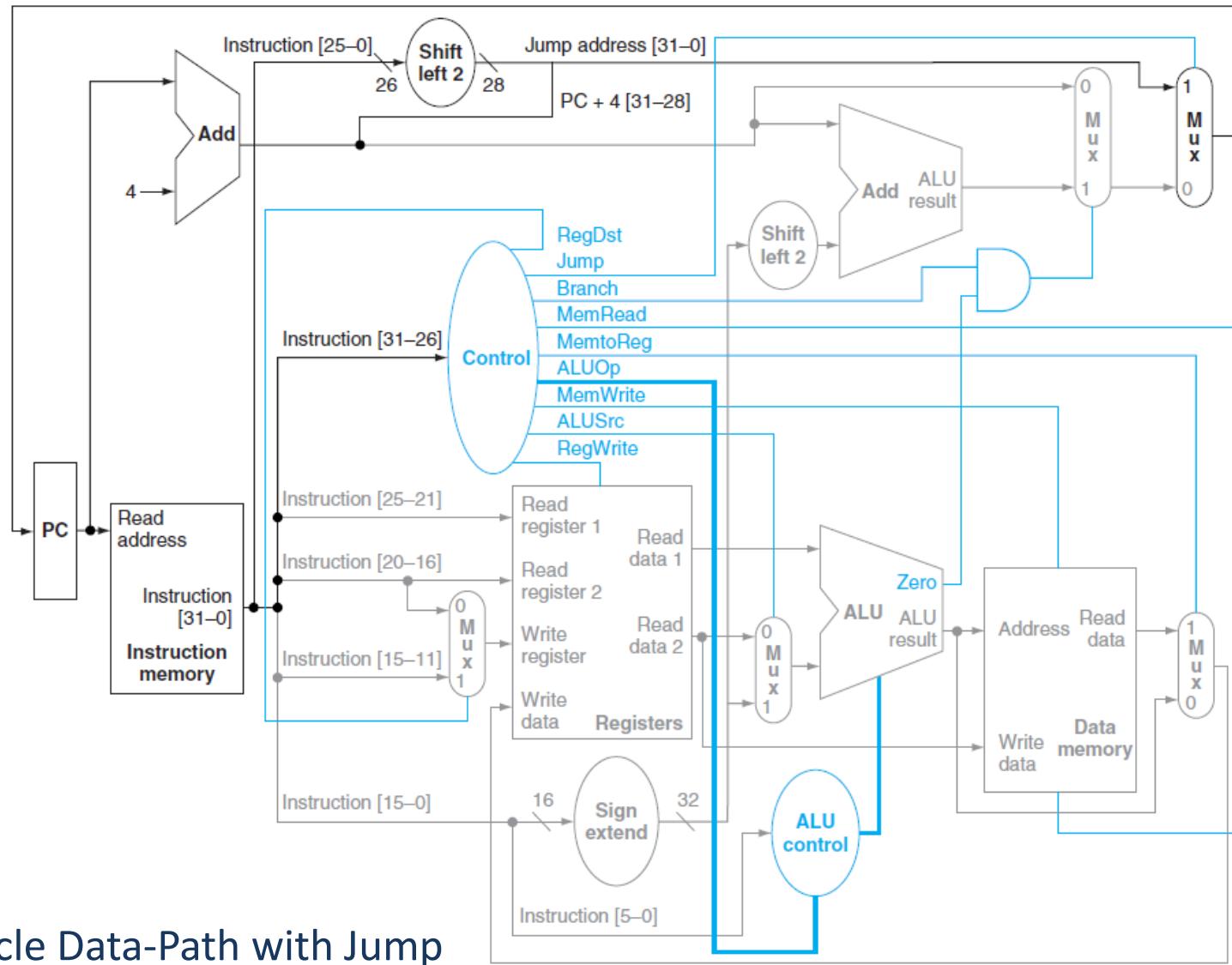


| Instruction Opcode | ALUOp | Instruction Operation | Function Field | Desired ALU Operation | ALU Control |
|--------------------|-------|-----------------------|----------------|-----------------------|-------------|
| LW | 00 | load word | XXXXXX | add | 0010 |
| SW | 00 | store word | XXXXXX | add | 0010 |
| Branch equal | 01 | branch equal | XXXXXX | subtract | 0110 |
| R-type | 10 | add | 100000 | add | 0010 |
| R-type | 10 | subtract | 100010 | subtract | 0110 |
| R-type | 10 | and | 100100 | and | 0000 |
| R-type | 10 | or | 100101 | or | 0001 |
| R-type | 10 | set on less than | 101010 | set on less than | 0111 |

Local Control for the ALU
opcode → ALUOp → ALUCtrl

Our example uses 2-bits for ALUOp. It can be further extended if necessary

Single-Cycle CPU Design – Step 4

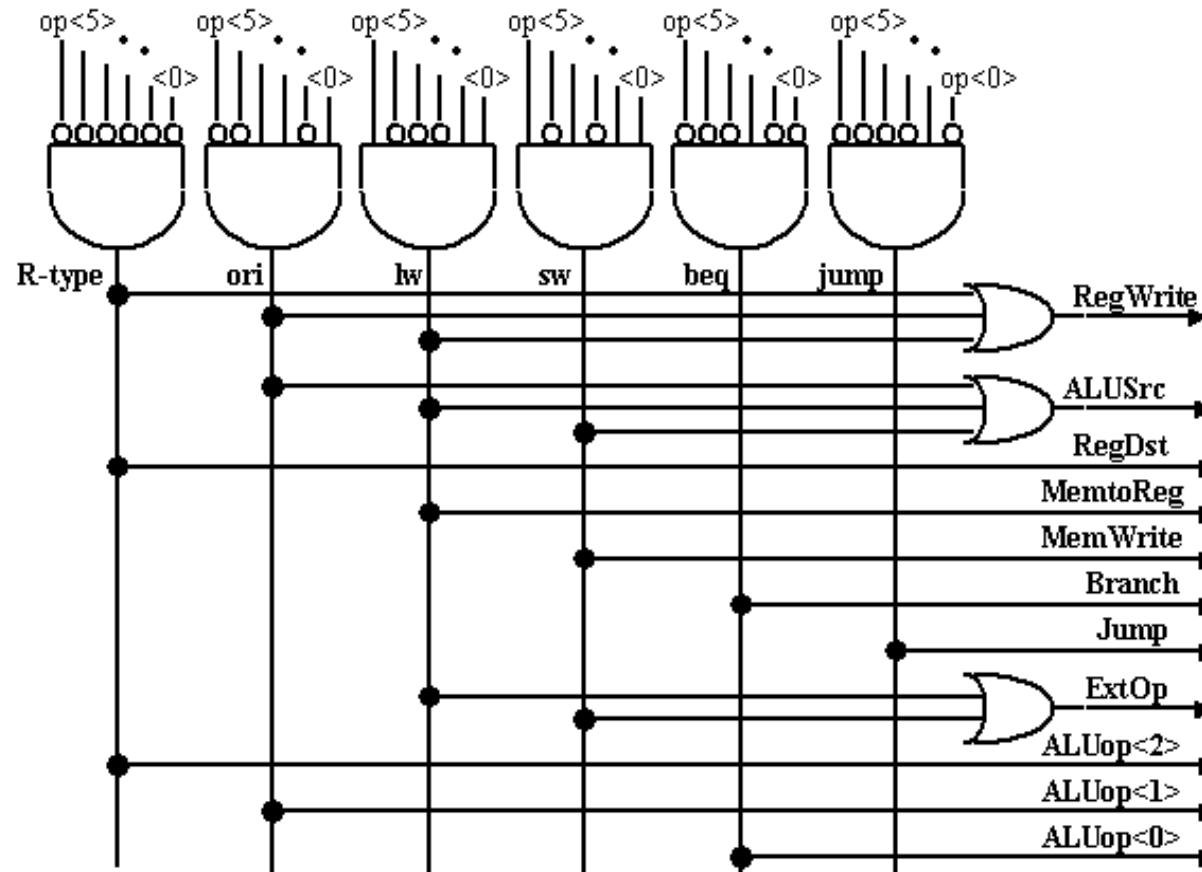


Single-Cycle Data-Path with Jump

[1]



- Design Step 5: Implement the Control
 - Only combinational logic is needed for the Single-Cycle Control



A possible PLA implementation of the Main Control Unit

[1]



Single-Cycle CPU Design



- Critical Path

- Load Word operation: PC's Clk-to-Q + Instruction Memory's Access Time + Register File's Access Time + ALU to Perform a 32-bit Add + Data Memory Access Time + Setup Time for Register File Write + Clock Skew

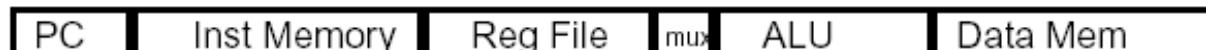
Arithmetic & Logical



Load



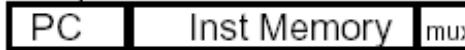
Store



Branch



Jump



Single cycle: Instruction Timing Comparison

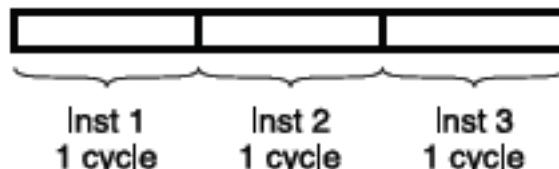


Single-Cycle CPU Design

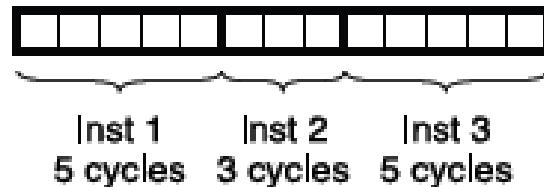


- Single-Cycle disadvantages
 - The clock cycle is chosen to fulfill the critical path (lw) → slow clock
 - The time needed for a load is much larger than for other instructions

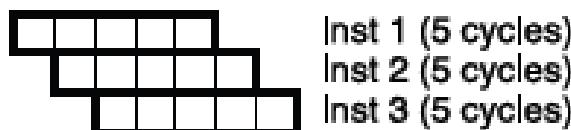
Single-Cycle
Unpipelined
CPI = 1



Multi-Cycle
Unpipelined
CPI = 4.33



Pipelined
CPI = 1



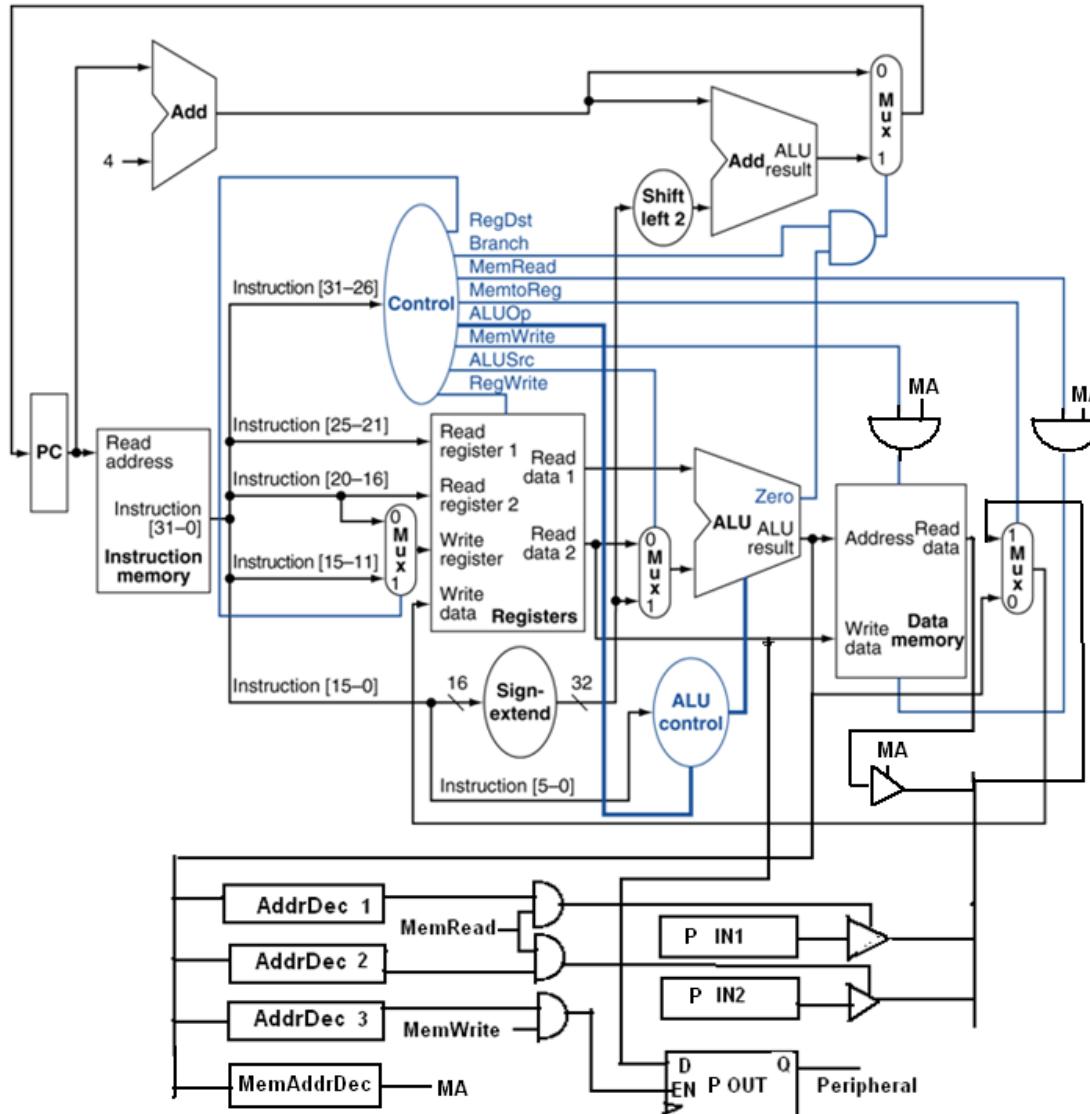
| Processor type | CPI | CLK cycle time |
|----------------|-----|------------------|
| Single-Cycle | 1 | Long cycle time |
| Multi-Cycle | >1 | Short cycle time |
| Pipelined | ~1 | Short cycle time |

Remember: Computer Performance

$$\frac{\text{Time}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \cdot \frac{\text{Cycles}}{\text{Instruction}} \cdot \frac{\text{Time}}{\text{Cycle}}$$

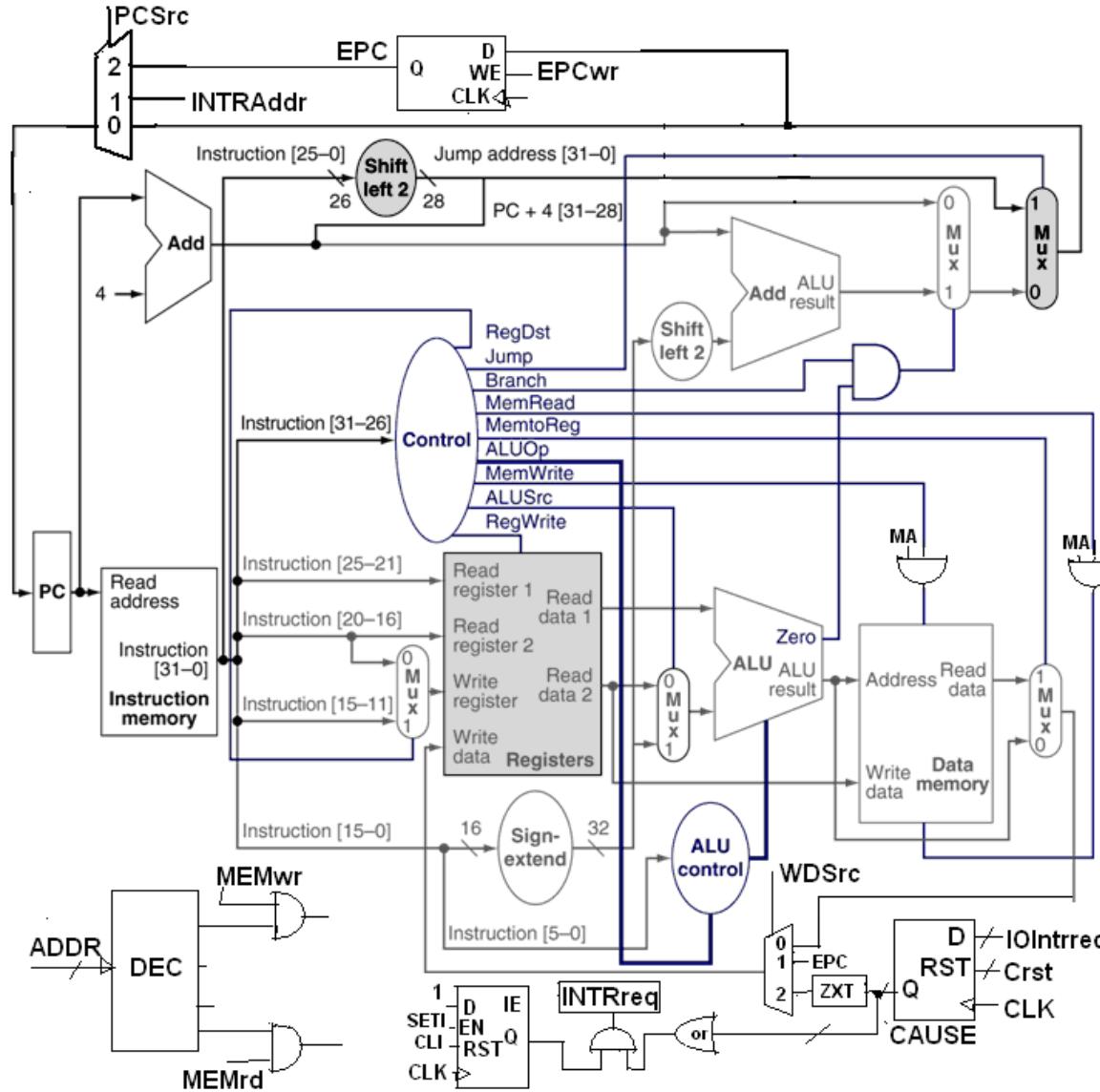


- Single-Cycle CPU Extensions
 - Problem: define some ports for I/O communication
 - A convenient solution (without introducing dedicated instructions)
 - I/O mapped through the memory address space
 - Some addresses from data memory will be reserved for I/O ports
 - Writing and reading to this I/O ports is carried out by using the standard instructions for memory accesses (lw and sw)
 - LW and SW used for word (32-bits) transfers (use LH and SH for half words, LB and SB for bytes)
 - You need to specify in the RTL description how to handle the reserved addresses for peripherals. From RTL will result the necessary supplementary components (address decoders, etc.)



Connecting I/O Devices

- Devices are mapped through the memory address space
 - 2 INPUTS
 - 1 OUTPUT
- Control Signals:
 - MemRead
 - MemWrite
 - MA



MIPS Interrupt Mechanism

- IE – Interrupt Enable
- ZXT – Zero Extender
- See the previous course



Problems – Homework



- Implement other instructions for the Single-Cycle MIPS CPU
 - add, sub, and, or, lw, sw, beq, j, addi, andi, ori
 - sll, srl, sra, sllv, srav, srlv
 - slt, slti
 - bne , bgez, bltz,...
 - jr, jal
 -
- Implement new instructions for the Single-Cycle MIPS CPU
 - LWR, SWR (sums two registers to obtain the memory address)
 - LWA, SWA (uses a single register to obtain the memory address)
 - SWAP two registers
 - Arithmetic/logical instructions with memory operands
 - addm \$t2, 100(\$t3) $\$t2 \leftarrow \$t2 + M[\$t3+100]$



References



1. D. A. Patterson, J. L. Hennessy, “Computer Organization and Design: The Hardware/Software Interface”, 5th edition, ed. Morgan–Kaufmann, 2013.
2. D. A. Patterson and J. L. Hennessy, “Computer Organization and Design: A Quantitative Approach”, 5th edition, ed. Morgan-Kaufmann, 2011.
3. MIPS32™ Architecture for Programmers, Volume I: “Introduction to the MIPS32™ Architecture”.
4. MIPS32™ Architecture for Programmers Volume II: “The MIPS32™ Instruction Set”.



Computer Architecture

Lecturer: Mihai Negru

2nd Year, Computer Science

Lecture 5: ALU Design

<http://users.utcluj.ro/~negrum/>



Binary Number Representation



| Sign Magnitude | One's Complement | Two's Complement |
|----------------|------------------|------------------|
| 000 = +0 | 000 = +0 | 000 = +0 |
| 001 = +1 | 001 = +1 | 001 = +1 |
| 010 = +2 | 010 = +2 | 010 = +2 |
| 011 = +3 | 011 = +3 | 011 = +3 |
| 100 = -0 | 100 = -3 | 100 = -4 |
| 101 = -1 | 101 = -2 | 101 = -3 |
| 110 = -2 | 110 = -1 | 110 = -2 |
| 111 = -3 | 111 = -0 | 111 = -1 |

- 2's complement – advantages
 - Subtract can share the same logic as add
 - Sign bit can be treated as a normal number bit in addition
- 1's complement disadvantage
 - Two zero representations



Binary Number Representation – MIPS



- Unsigned binary integers
 - Typically represent addresses or other values that are guaranteed not to be negative
 - Unsigned value
 - An n-bit unsigned binary integer has a range from 0 to $2^n - 1$
- Signed binary integers
 - Typically used to represent data that is either positive or negative
 - The most common representation → the 2's complement format
 - Signed value (2's complement)

$$value = \sum_{i=0}^{n-1} b_i \cdot 2^i$$

$$value = -b_{n-1} \cdot 2^{n-1} + \sum_{i=0}^{n-2} b_i \cdot 2^i$$

- An n-bit 2's complement binary integer has a range from -2^{n-1} to $2^{n-1} - 1$



Binary Number Representation – MIPS



- 32-bit signed numbers

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_2 = 0_{10}$

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_2 = +1_{10}$

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_2 = +2_{10}$

...

$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_2 = +2,147,483,646_{10}$

$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_2 = +2,147,483,647_{10} \rightarrow \text{max_int}$

$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_2 = -2,147,483,648_{10} \rightarrow \text{min_int}$

$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_2 = -2,147,483,647_{10}$

$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_2 = -2,147,483,646_{10}$

...

$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_2 = -3_{10}$

$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_2 = -2_{10}$

$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_2 = -1_{10}$



Binary Number Operations – MIPS



- 2's complement negation
 - Invert all bits and add one to the least significant bit
 - 2's complement representation: $6 = 0110$ $-4 = (\text{not } 0100 + 0001) = 1100$

- 2's complement addition
 - Add the corresponding bits of both numbers with carry between bits

| | | | |
|--------------|---------------|--------------|---------------|
| $3 = 0011$ | $-3 = 1101$ | $-3 = 1101$ | $3 = 0011$ |
| $+ 2 = 0010$ | $+ -2 = 1110$ | $+ 2 = 0010$ | $+ -2 = 1110$ |
| ----- | ----- | ----- | ----- |

- Unsigned and 2's complement addition are performed in exactly the same way, only the **overflow** detection differs
- 2's complement subtraction
 - Negate the second number and then perform addition

| | | | |
|--------------|---------------|-------------|---------------|
| $3 = 0011$ | $-3 = 1101$ | $-3 = 1101$ | $3 = 0011$ |
| $- 2 = 0010$ | $- -2 = 1110$ | $-2 = 0010$ | $- -2 = 1110$ |
| ----- | ----- | ----- | ----- |



Binary Number Operations – MIPS



- Overflow

- The sum or difference can go beyond the range of representable numbers
- Overflow: the result is too large or too small for proper representation

| | | | |
|------------|-------------|-------------|-------------|
| 5 = 0101 | -5 = 1011 | +5 = 0101 | -5 = 1011 |
| + 6 = 0110 | + -6 = 1010 | -- 6 = 1010 | - +6 = 0110 |
| ----- | ----- | ----- | ----- |
| -5=1011 | 5=0101 | -5=1011 | 5=0101 |

- Overflow generates an incorrect result that should be detected
- Overflow occurs when the **resulting value affects the sign**
 - 2 positive numbers and the sum is negative
 - 2 negative numbers and the sum is positive

| Operation | A | B | Result indicating overflow |
|-----------|----------|----------|----------------------------|
| A + B | ≥ 0 | ≥ 0 | < 0 |
| A + B | < 0 | < 0 | ≥ 0 |
| A - B | ≥ 0 | < 0 | < 0 |
| A - B | < 0 | ≥ 0 | ≥ 0 |

No overflow when

- signs are different for addition
- signs are the same for subtraction



Binary Number Operations – MIPS



- Overflow detection – 2's complement numbers
 - When adding 2's complement numbers, overflow will occur only if
 - the numbers being added have the same sign
 - the sign of the result is different

$$\begin{array}{r} \overline{a_{n-1} a_{n-2} \dots a_1 a_0} \\ + \overline{b_{n-1} b_{n-2} \dots b_1 b_0} \\ \hline = s_{n-1} s_{n-2} \dots s_1 s_0 \end{array}$$

$$\text{overflow} = a_{n-1} \cdot b_{n-1} \cdot s_{n-1} + \overline{a_{n-1}} \cdot \overline{b_{n-1}} \cdot s_{n-1}$$

- If c_{n-1} and c_n represent the input and output carry signals for the MSB

| Operands | Result | c_n | s_{n-1} | a_{n-1} | b_{n-1} | c_{n-1} | event ? |
|----------|----------|-------|-----------|-----------|-----------|-----------|---|
| Positive | Positive | 0 | 0 | 0 | 0 | 0 | $c_n = c_{n-1} \Leftrightarrow$ no overflow |
| | Negative | 0 | 1 | 0 | 0 | 1 | $c_n \neq c_{n-1} \Leftrightarrow$ overflow |
| Negative | Positive | 1 | 0 | 1 | 1 | 0 | $c_n \neq c_{n-1} \Leftrightarrow$ overflow |
| | Negative | 1 | 1 | 1 | 1 | 1 | $c_n = c_{n-1} \Leftrightarrow$ no overflow |

Overflow means

$$\rightarrow c_n \neq c_{n-1}$$

Overflow detection

$$\rightarrow \text{overflow} = \text{CarryOut MSB} \text{ xor } \text{CarryInMSB}$$

$$\text{overflow} = c_n \otimes c_{n-1}$$



Binary Number Operations – MIPS



- Overflow detection – unsigned numbers
 - Unsigned numbers – overflow → carry out of the most significant bit

$$\text{overflow} = c_n$$

$$\begin{array}{rcl} 1001 & = 9 \\ + 1000 & = 8 \\ \hline & & \\ & & \\ = 0001 & = 1 \\ \hline c_n & = 1 \end{array}$$

- MIPS architecture
 - Overflow exceptions are **signaled** for 2's complement arithmetic
 - add, sub, addi
 - Overflow exceptions are **not signaled** for unsigned arithmetic
 - addu, subu, addiu



Binary Number Operations – MIPS



- Set on Less Than – SLT

- Signed integers less than condition $\rightarrow SF \neq OF \rightarrow$ sign xor overflow
 - Set-on-Less Than Signed (SLT) instruction

slt \$rd, \$rs, \$rt // R-type

If($RF[rs] < RF[rt]$) $\rightarrow RF[rd] \leftarrow 1$ else $RF[rd] \leftarrow 0$
If($RF[rs] - RF[rt]$) $\rightarrow (SF \text{ xor } OF) = 1 \rightarrow RF[rd] \leftarrow 1$ else $RF[rd] \leftarrow 0$

- Set on Less Than Unsigned – SLTU

- Unsigned integer numbers

sltu \$rd, \$rs, \$rt // R-type

If($RF[rs] < RF[rt]$) $\rightarrow RF[rd] \leftarrow 1$ else $RF[rd] \leftarrow 0$
If($RF[rs] - RF[rt]$) $\rightarrow CF = 0 \rightarrow RF[rd] \leftarrow 1$ else $RF[rd] \leftarrow 0$

- SLT or SLTU can be accomplished by

- subtracting \$rt from \$rs
 - setting the least significant bit of the result to $((SF \text{ xor } OF) \text{ or } \sim CF)$ and setting all other bits to zero



Arithmetic Logic Unit Design



- The Design Process
 - “To Design Is To Represent”
 - Design Begins With Requirements
 - Functional capabilities
 - Performance characteristics
 - Design Finishes as Assembly
 - Design understood in terms of components and how they have been assembled
 - Top-Down *decomposition* of complex functions (behaviors) into more primitive ones
 - Bottom-up *composition* of primitive building blocks into more complex assemblies



Arithmetic Logic Unit Design



- The ALU should support a subset of arithmetic-logic instructions of MIPS.

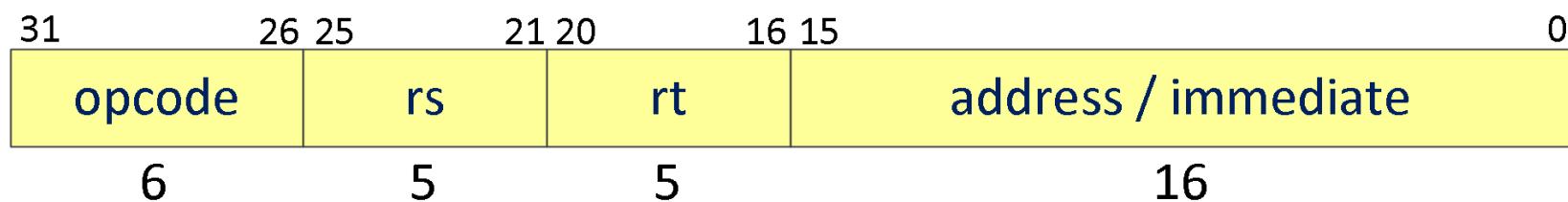
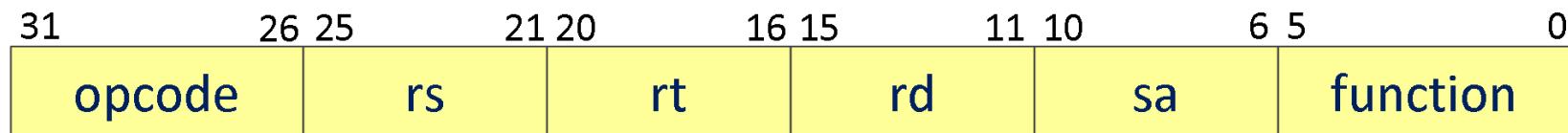
| Type | opcode | function |
|-------|--------|----------|
| addi | 001000 | xxxxxx |
| addiu | 001001 | xxxxxx |
| slti | 001010 | xxxxxx |
| sltiu | 001011 | xxxxxx |
| andi | 001100 | xxxxxx |
| ori | 001101 | xxxxxx |
| xori | 001110 | xxxxxx |
| lui | 001111 | xxxxxx |
| beq | 000100 | xxxxxx |

beq – specific for ALU operation

| Type | opcode | function |
|------|--------|----------|
| add | 000000 | 100000 |
| addu | 000000 | 100001 |
| sub | 000000 | 100010 |
| subu | 000000 | 100011 |
| and | 000000 | 100100 |
| or | 000000 | 100101 |
| xor | 000000 | 100110 |
| nor | 000000 | 100111 |
| slt | 000000 | 101010 |
| sltu | 000000 | 101011 |

Designing an ALU for the MIPS ISA

- MIPS ALU requirements for a limited subset of instructions
 - add, addu, sub, subu, addi, addiu → 2's complement adder / subtractor with overflow detection (signed arithmetic generates overflow → detection)
 - and, andi, or, ori, xor, xori, nor → Logical AND, OR, XOR, NOR
 - slt, sltu, slti, sltiu → 2's complement adder, check sign/overflow of result
 - beq → zero detector
- MIPS arithmetic-logical instruction formats



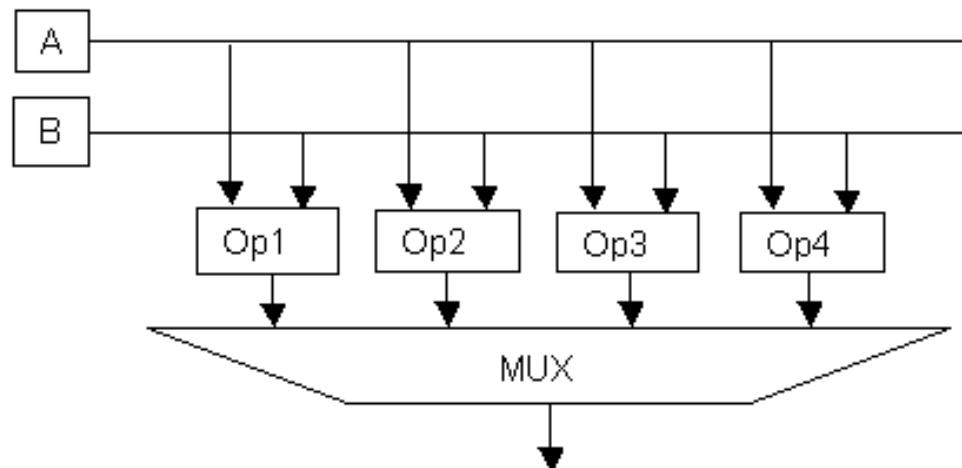


Arithmetic Logic Unit Design



- Design Trick 1: Divide et Impera
 - Break the problem into simpler ones
 - Solve them and glue together the solution
 - Example:
 - Sign / Zero Extended immediates before the ALU
 - No specific ALU ops for Immediates: addi, addiu, executed as add, addu
- Refined requirements (Functional Specification)
 - ALU inputs:
 - 2 x 32-bit operands A, B
 - 4-bit operation code
 - ALU outputs:
 - 32-bit result
 - Sign, Carry, Overflow flags
 - ALU operations:
 - add, addu, sub, subu, and, or, xor, nor, slt, sltu

- Design Trick 2:
 - Take standard digital logic components (AND, OR, +, ...),
 - Connect them conform specification, and select the required operation by MUX (Laboratory version)



- Design trick 3:
 - Solve part of the problem and extend it
 - Start with AND, OR



Arithmetic Logic Unit Design



- Building a basic Arithmetic Logic Unit

- Construct an ALU from:
 - logic gates: AND, OR, XOR, etc.
 - inverters
 - multiplexers

- MIPS word is 32 bits wide

→ we need a 32-bit-wide ALU

- Connect 32 x 1-bit ALUs to create the MIPS ALU

- 1-bit ALU Design

- Start with logical operations

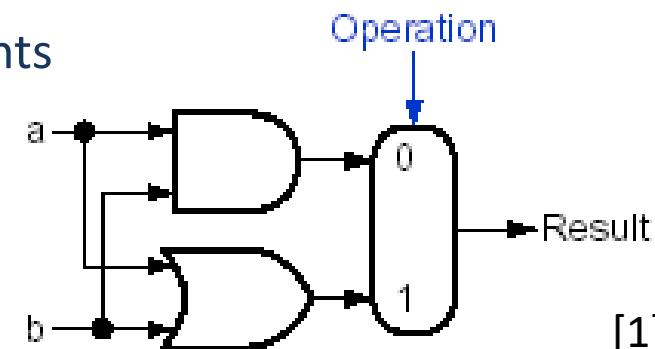
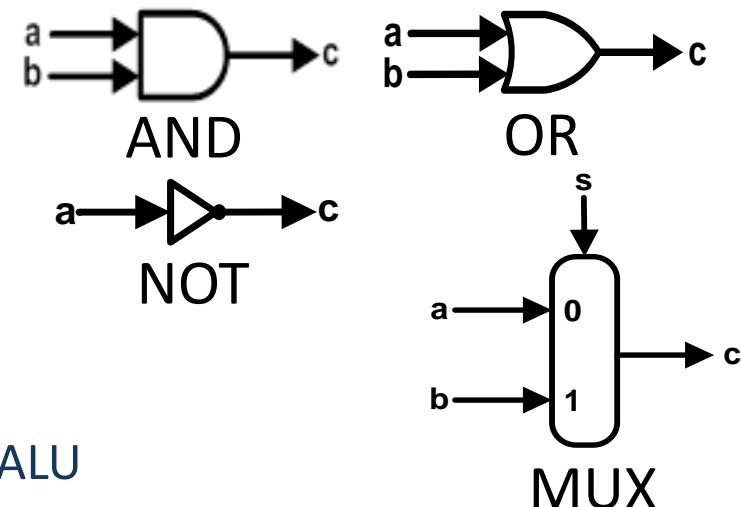
→ they map directly onto the Hardware components

1-bit ALU for Logical AND and Logical OR

The multiplexer selects the Results as:

- a AND b
 - a OR b

Basic ALU Building Blocks



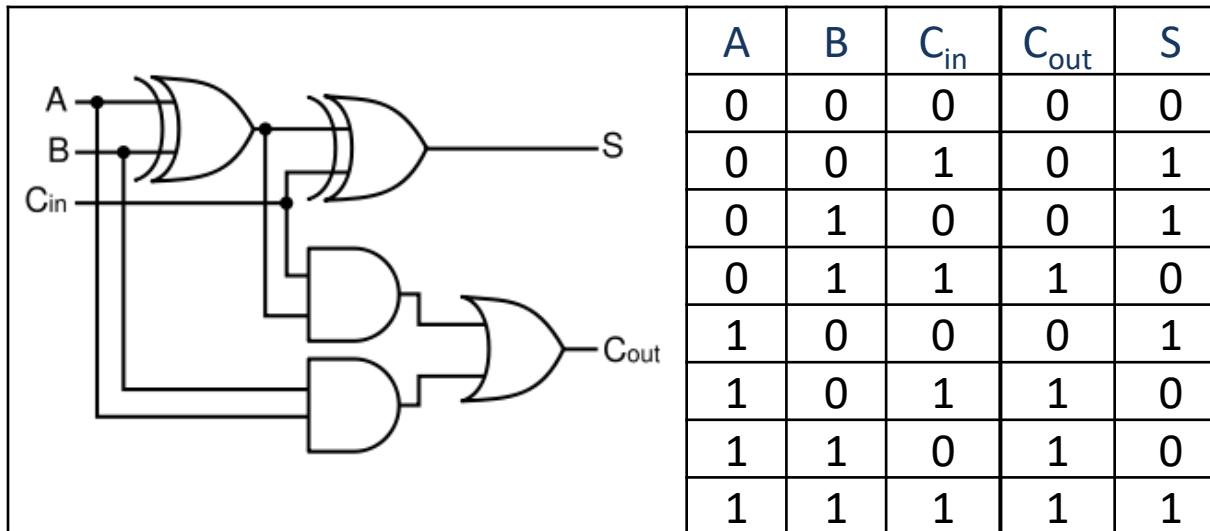
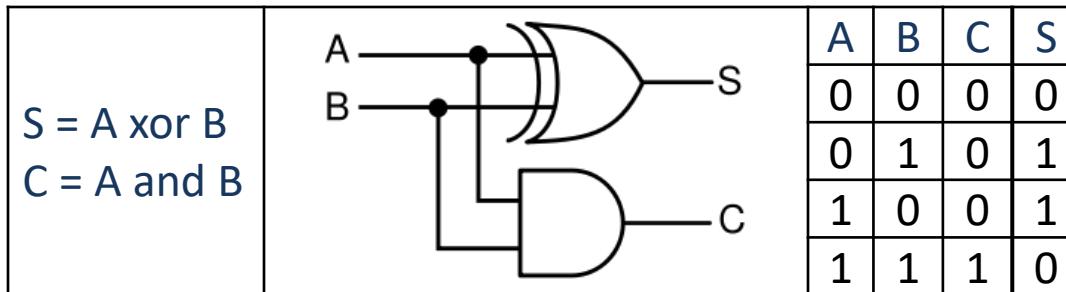
[1]



Arithmetic Logic Unit Design



- The next function to include is addition
 - HALF ADDER / FULL ADDER



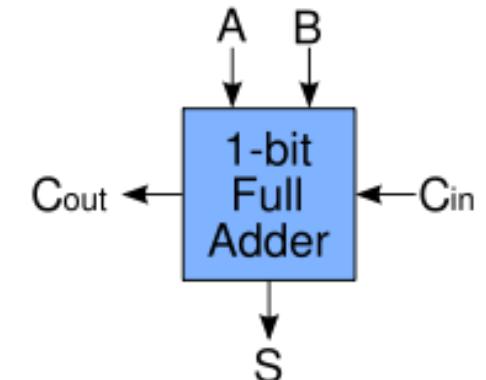
$$S = A \text{ xor } B \text{ xor } C_{in}$$

$$C_{out} = A \text{ and } B \text{ or } A \text{ and } C_{in} \text{ or } B \text{ and } C_{in} = A \text{ and } B \text{ or } C_{in} \text{ and } (A \text{ xor } B)$$

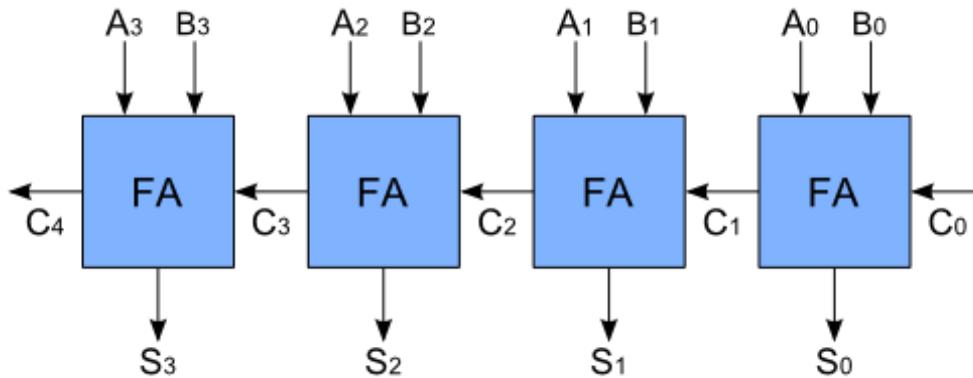
Observation for Full Adder:

$$C_{in} \text{ and } (A \text{ or } B) \rightarrow C_{in} \text{ and } (A \text{ xor } B)$$

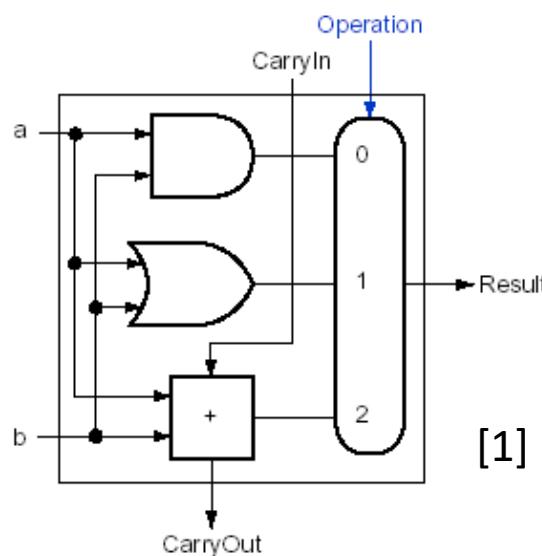
The difference between or/xor is in the $A=B=1$ case (covered by A and B)



1-bit Full Adder Symbol

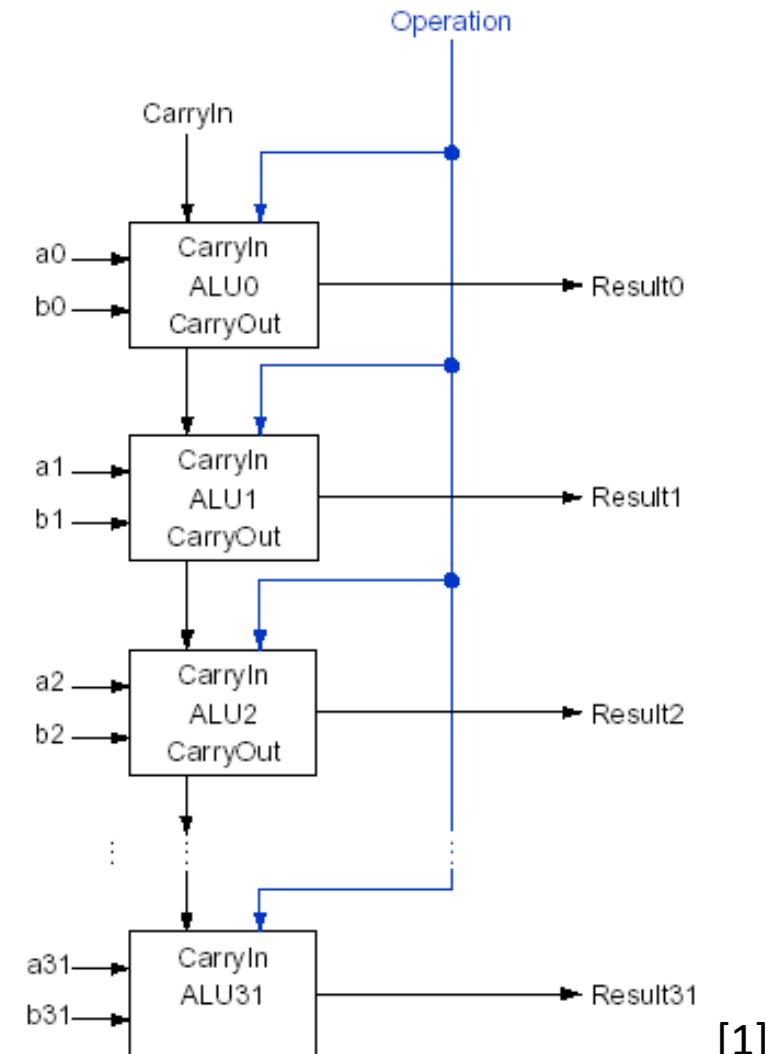


4-bit Ripple Carry Adder:
 $C_{in} \leftarrow C_{out}$ between stages



1-bit ALU for AND, OR, ADD; Operation – 2bits

Carry propagation delay

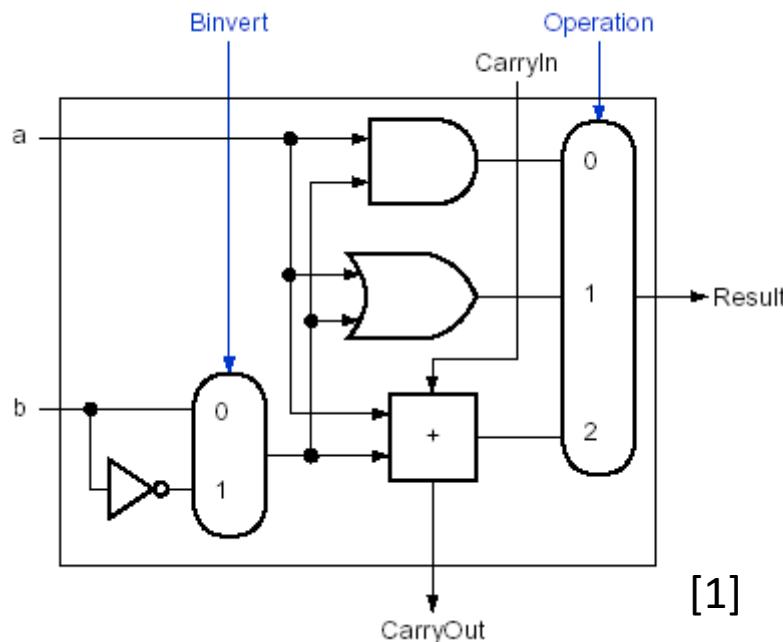


32-bit ALU built with 32 x 1-bit ALUs



- Additional Operations: subtraction

- $a - b = a + (-b)$
- Subtraction is the same as adding the negative version of an operand
- 2's complement negate \rightarrow invert each bit and then add 1
- To invert each bit, we simply add a **2:1 mux** that chooses between b and \bar{b}



1-bit ALU with subtraction

$$a - b = a + \bar{b} + 1$$

Operation = 2
 Binvert = 1
 CarryIn = 1

Subtraction \Leftrightarrow Adding 2's complement of b to a

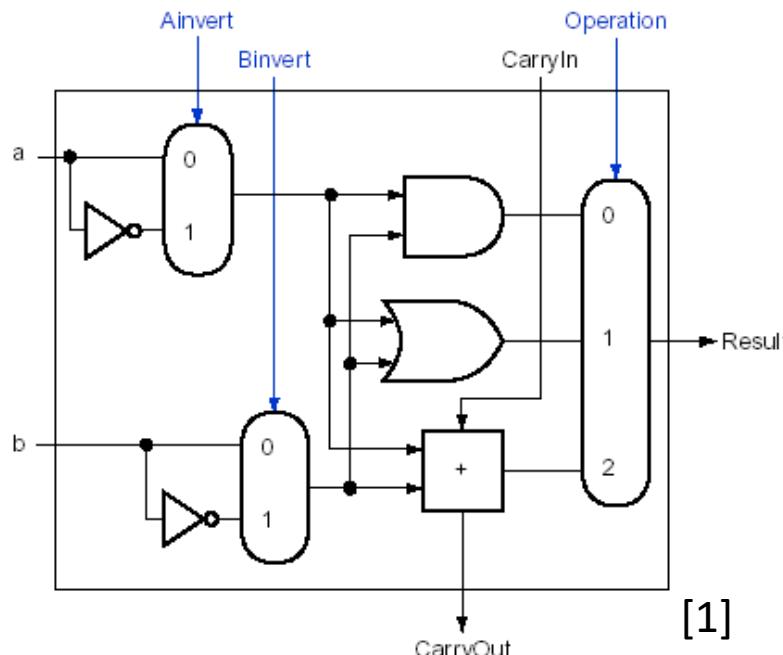


- Additional operations: NOR

- Instead of adding a separate NOR gate, reuse the hardware already present in the ALU

$$NOR : \overline{a \text{ OR } b} = \overline{a} \text{ AND } \overline{b} \quad \text{De Morgan's Theorems}$$

- Because we already have **AND** and \bar{b} , we need to add \bar{a} to the ALU



Ainvert = 1

Binvert = 1

Operation = 0

→ we get (a NOR b) instead of (a AND b)

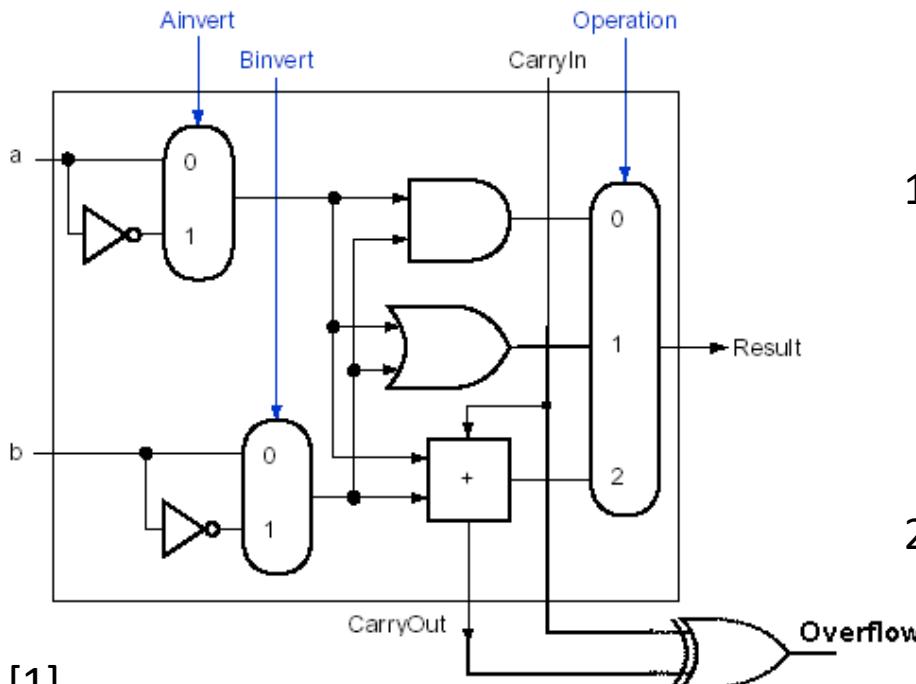
What about NAND? → homework

1-bit ALU: AND, OR, NOR, ADD, SUB



- Overflow Detection

$$\text{overflow} = \text{CarryOut MSB} \text{ xor } \text{CarryInMSB}$$



1-bit ALU for the MSB bit with overflow detection

CPU action at an overflow, two methods:

1. Ignore it → MIPS for unsigned instructions
 - Do not detect overflow for
 - addu, addiu, subu
 - **addiu still sign-extends!**
 - sltu, sltiu for unsigned comparisons
2. Recognize it → MIPS for signed Instructions
 - Generate a trap so that the programmer can try to deal with it
 - An **exception (interrupt)** occurs
 - Control jumps to predefined address for exception
 - Interrupted address is saved for possible resumption
 - MIPS instructions: add, sub

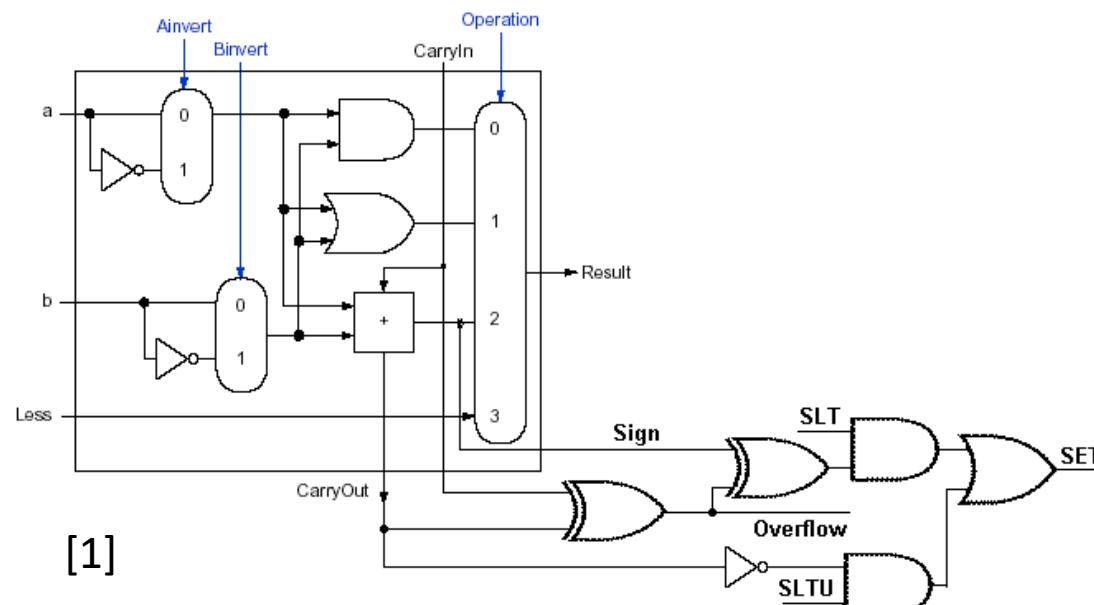


Arithmetic Logic Unit Design



- Additional operations: set on less than instruction (slt)
 - the slt operation produces 1, if $RF[rs] < RF[rt]$, and 0 otherwise
 - slt will set all bits except the LSB to 0
 - the LSB set according to the comparison $RF[rs] < RF[rt]$
 - expand the 3-input MUX of the ALU to add a new input for: Less \rightarrow slt result
 - Connect 0 to the Less inputs: Less[31:1] = 0
 - How to set the Less[0]?
 - Subtract b from a. If the difference is negative, then $a < b$
 - Analyze Bit 31 of results \rightarrow Sign Bit
 - Sign = 1 \rightarrow negative result ($a < b$) / Sign = 0 \rightarrow positive result ($a > b$)
 - Analyze Carry Flag CF
 - SLTU (Unsigned)
 - Less[0] \leftarrow Set $\leftarrow \sim CF$
 - SLT (Signed)
 - Less[0] \leftarrow Set $\leftarrow \text{Sign xor Overflow}$

Arithmetic Logic Unit Design



1-bit ALU for the MSB bit with overflow detection and set generation

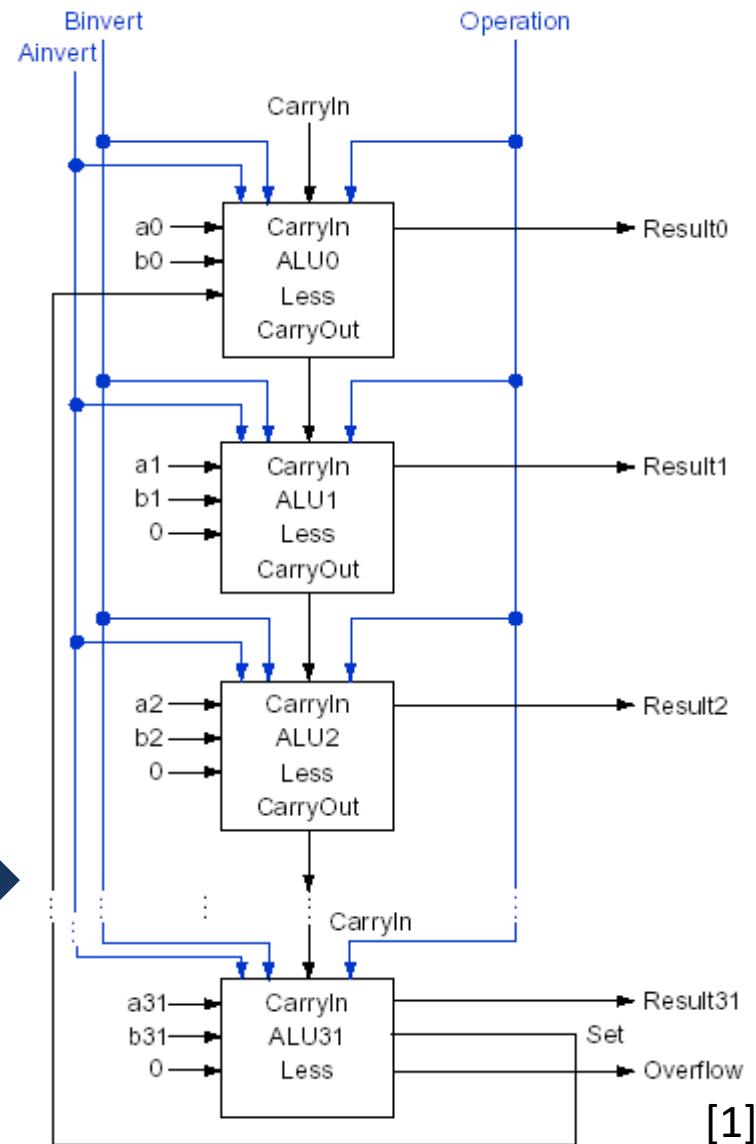
32-bit ALU built with 32×1 -bit ALUs

$\text{Less}[31:1] = 0$, $\text{Less}[0] = \text{Set}$ from ALU31

Subtraction operations: $\text{CarryIn} = \text{Binvert} = 1$

Addition or logical operations: $\text{CarryIn} = \text{Binvert} = 0$

We can simplify the control: $\text{CarryIn} = \text{Binvert} = \text{Bnegate}$





- Additional operations: test for Conditional Branch Instructions

beq – Branch if 2 registers are equal

bne – Branch if 2 registers are not equal

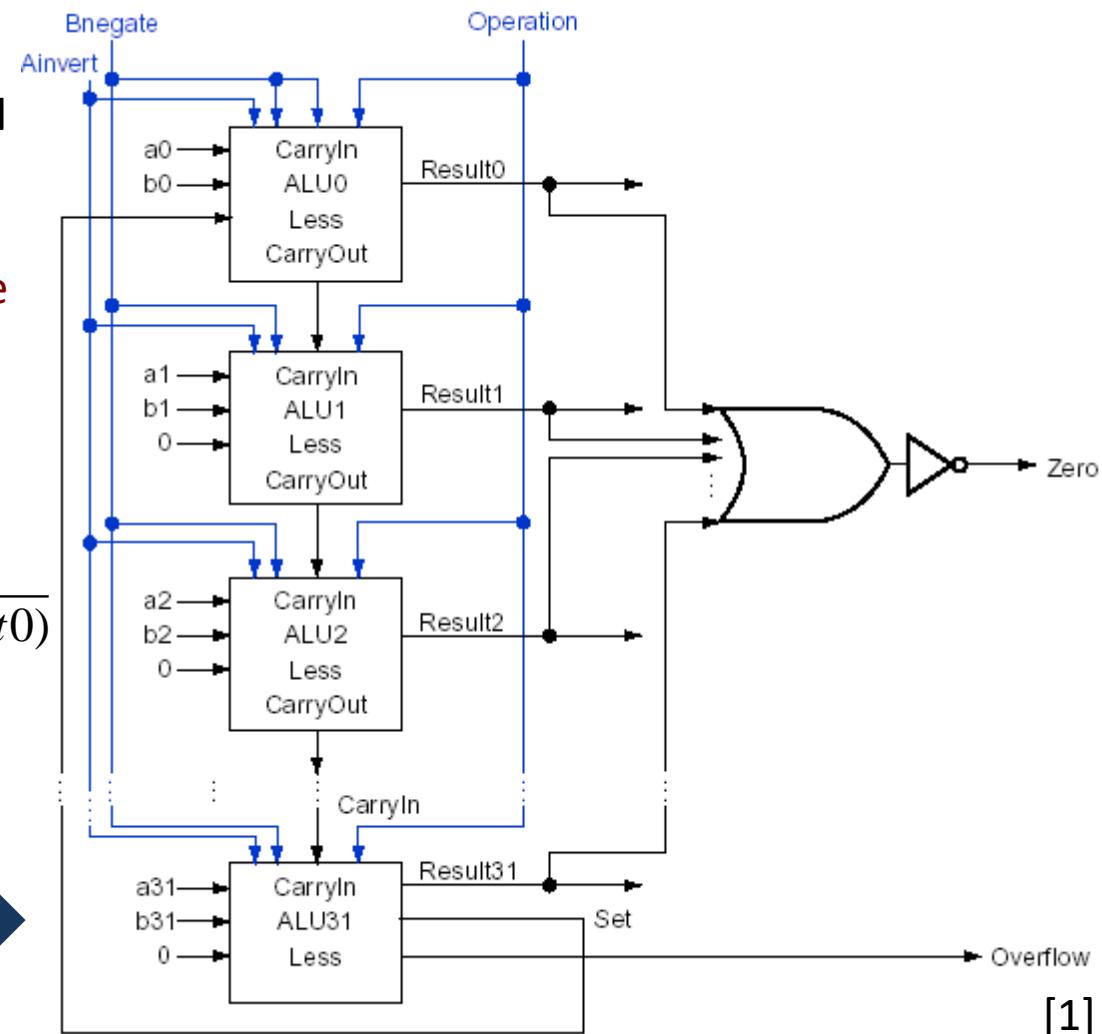
Equality test:

→ subtract b from a and then test if the result is 0

Add specific hardware to test if the result is 0: NOR (negated OR)

$$Zero = \overline{(Result31 \text{ or } \dots \text{ or } Result1 \text{ or } Result0)}$$

Final 32-bit ALU with Zero detector, overflow and Set-Less

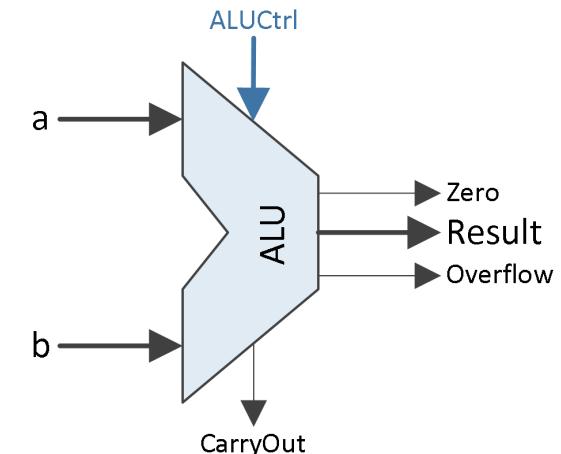


[1]



| ALU control lines – ALUCtrl | | | ALU Operation |
|-----------------------------|---------|-----------|--------------------|
| Ainvert | Bnegate | Operation | |
| 0 | 0 | 0 | 0 AND |
| 0 | 0 | 0 | 1 OR |
| 0 | 0 | 1 | 0 ADD |
| 0 | 1 | 1 | 0 SUBTRACT |
| 0 | 1 | 1 | 1 SET ON LESS THAN |
| 1 | 1 | 0 | 0 NOR |

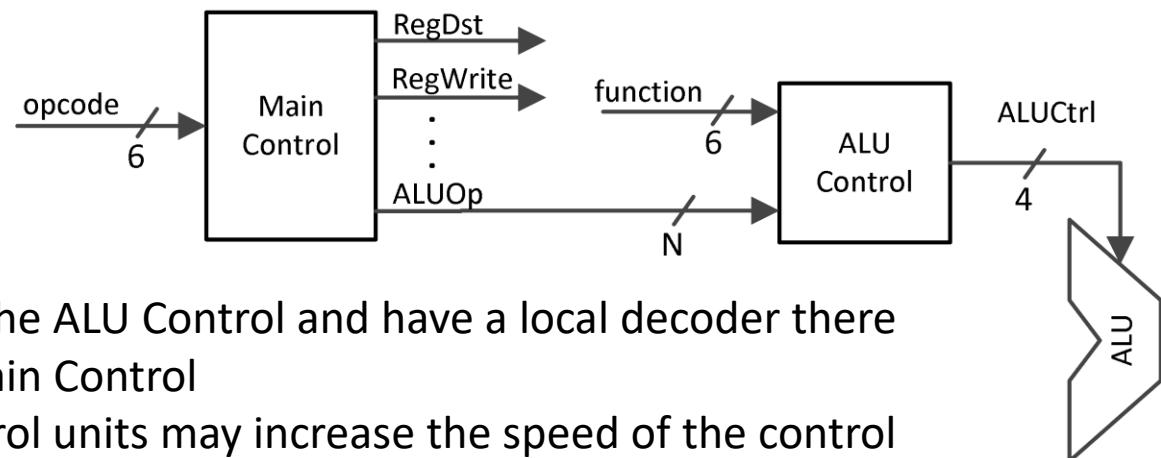
ALU Control Lines and the corresponding ALU operation



ALU Symbol

ALU Control Unit Design

- Multilevel decoding
 - **Hierarchical control**
 - Pass the function field to the ALU Control and have a local decoder there
 - Reduces the size of the Main Control
 - Using several smaller control units may increase the speed of the control unit





Arithmetic Logic Unit Control



| Instruction Opcode | ALUOp | Instruction Operation | Function Field | Desired ALU Operation | ALU Control |
|--------------------|-------|-----------------------|----------------|-----------------------|-------------|
| LW | 00 | load word | XXXXXX | add | 0010 |
| SW | 00 | store word | XXXXXX | add | 0010 |
| Branch equal | 01 | branch equal | XXXXXX | subtract | 0110 |
| R-type | 10 | add | 100000 | add | 0010 |
| R-type | 10 | subtract | 100010 | subtract | 0110 |
| R-type | 10 | and | 100100 | and | 0000 |
| R-type | 10 | or | 100101 | or | 0001 |
| R-type | 10 | set on less than | 101010 | set on less than | 0111 |

MIPS-lite ALU Control

ALU Control Inputs

- 6-bit function field from the R-type instruction format
- 2-bit ALUOp given by the Main Control, according to **opcode** field of the instructions.
The size of ALUOp can be increased if more MIPS instructions are implemented.



Additional MIPS ALU Requirements



- Multiplication
 - mult, multu – signed and unsigned 32-bit multiplication
 - Paper and pencil unsigned example: **1000 (8) * 1001 (9) = 0100 1000 (72)**

| | | | | |
|--------------|---|---|---|---|
| Multiplicand | 1 | 0 | 0 | 0 |
| Multiplier | 1 | 0 | 0 | 1 |
| | 1 | 0 | 0 | 0 |
| | 0 | 0 | 0 | 0 |
| | 0 | 0 | 0 | 0 |
| | 1 | 0 | 0 | 0 |
| Product | 0 | 1 | 0 | 0 |

! m bits * n bits → m + n bits result

Binary makes it easy:

- Multiplier i bit = 0 → place 0 (0 x multiplicand)
- Multiplier i bit = 1 → place a copy (1 x multiplicand)

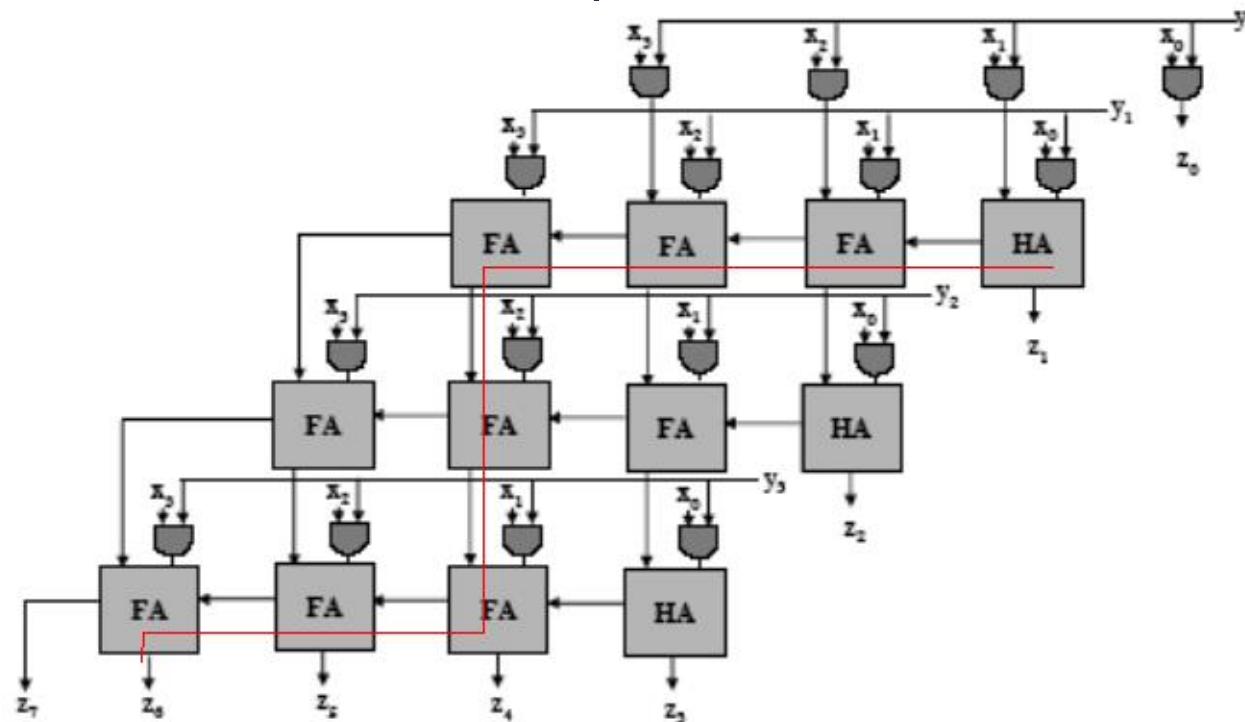
Multiplication is just a lot of additions and shifts!

Multiplier implementations:

- Combinational
- Pipelined
- Multi-Cycle (shift-add cycles)



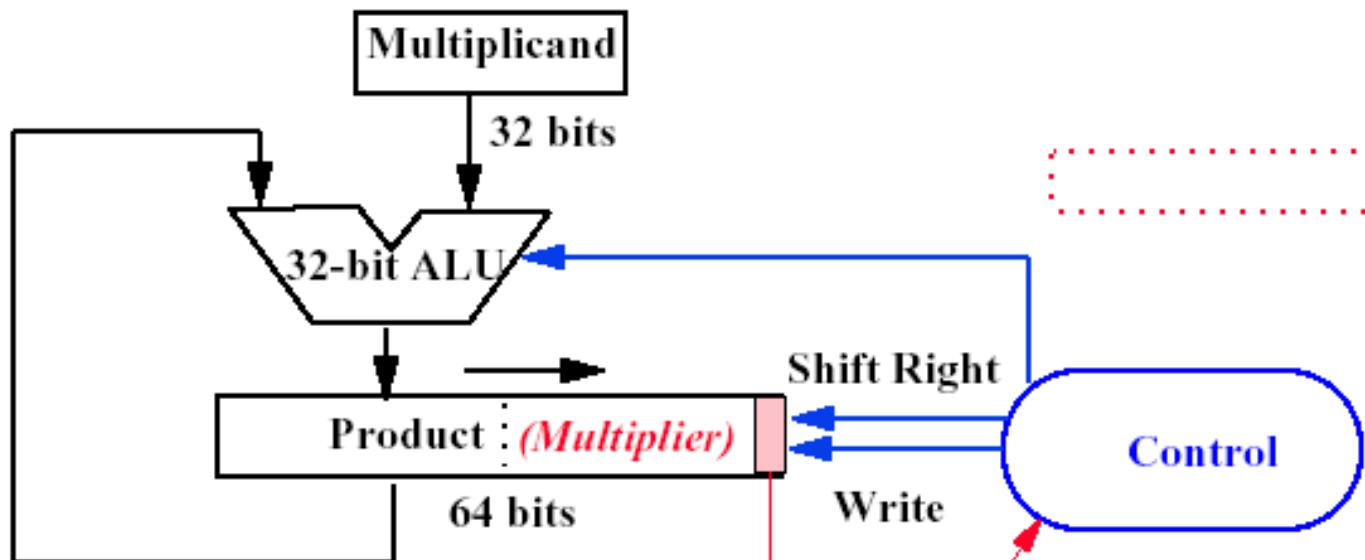
- Unsigned combinational multiplier



Ripple Carry Array Multiplier

- At each stage shift left X (multiplicand) ($<< 1 = x \cdot 2$)
- Use next bit of Y (multiplier) to determine whether to add in the shifted multiplicand or 0
- Accumulate the partial products
- Critical path is marked in red
- Pipelined versions improve the throughput of the multiplier

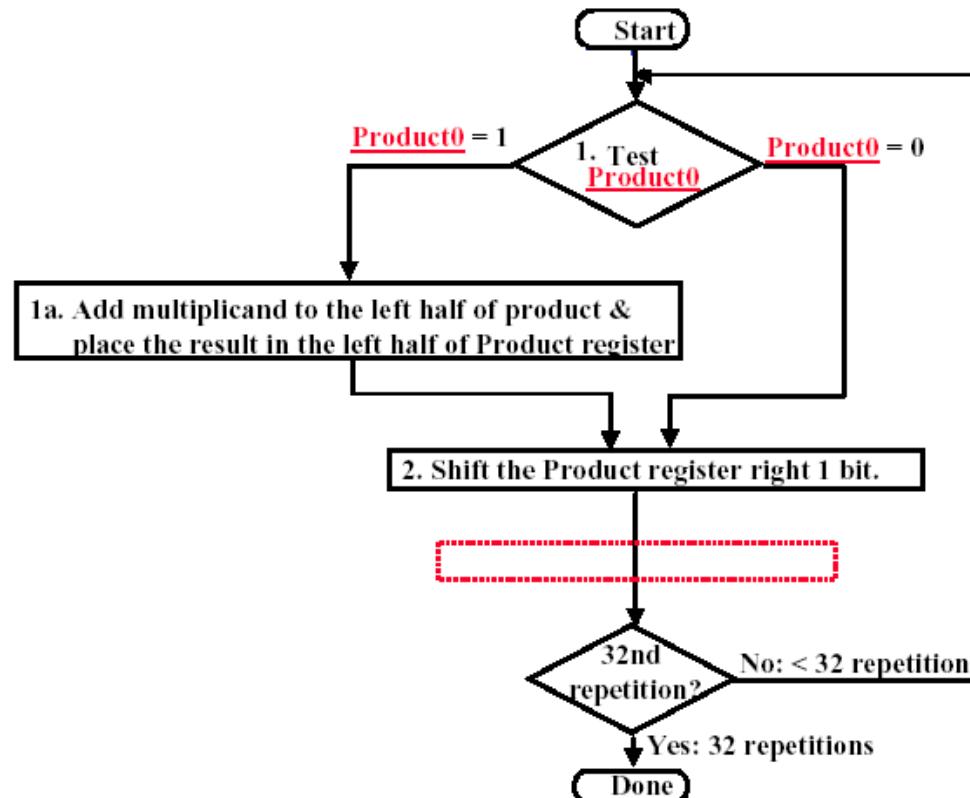
- Multi-cycle (shift-add) Multiplier
 - Implements the multiplication algorithm for binary numbers
 - No separate multiplier register
 - Multiplier placed on right side of 64-bit Product register
 - It has N (number of bits) iterations for summing up the partial products



Multi-cycle (shift-add) Multiplier block diagram



Additional MIPS ALU Requirements



Multiplication Algorithm ASM chart

Observations:

- 2 steps per bit because Shift by Shift register (Product & Multiplier)
- MIPS registers **Hi** and **Lo** are left and right half of Product
- MIPS instruction `multu` places the product in the **Hi** and **Lo** registers



Additional MIPS ALU Requirements



Example: 0010 * 0011

| Iteration | Step | Multiplicand | Product | Next |
|-----------|------|--------------|-----------|-------------------------|
| 0 | Init | 0010 | 0000 0011 | Product(0)='1' → add |
| 1 | 1a | | 0010 0011 | Shift Right |
| | 2 | 0010 | 0001 0001 | Product(0)='1' → add |
| 2 | 1a | | 0011 0001 | Shift Right |
| | 2 | | 0001 1000 | Product(0)='0' → no add |
| 3 | 1a | | 0001 1000 | Shift Right |
| | 2 | | 0000 1100 | Product(0)='0' → no add |
| 4 | 1a | | 0000 1100 | Shift Right |
| | 2 | | 0000 0110 | Done |

What about signed multiplication?

1. Easiest solution is to make both positive & remember whether to complement product when done (leave out the sign bit, run for 31 steps).
2. Apply definition of 2's complement: need to sign-extend partial products and subtract at the end.
3. Booth's algorithm.



Additional MIPS ALU Requirements



- Booth's algorithm
 - an elegant way to multiply signed numbers using the same hardware as before and save cycles
 - can handle multiple bits at a time
- Motivation for Booth's Algorithm
 - Booth algorithm gives a procedure for multiplying binary integers in 2's complement representation
 - Originally invented for Speed (when shift was faster than add)
 - Idea: string of 1's ...011...10... has as value the sum
$$2^n + 2^{n-1} + \dots + 2^m = 2^{n+1} - 2^m$$
 - Replace a string of 1s in the multiplier with an initial subtract when we first see a one and then later add after the last one

$$\begin{array}{r} -1 \\ +10000 \\ \hline 01111 \end{array}$$

| Current bit | Bit to the right | Explanation | Example | Op |
|-------------|------------------|---------------------|---------------------|------|
| 1 | 0 | Begins run of 1s | 000111 1 000 | sub |
| 1 | 1 | Middle of run of 1s | 00011 1 1000 | none |
| 0 | 1 | End of run of 1s | 00 0 1111000 | add |
| 0 | 0 | Middle of run of 0s | 00 0 1111000 | none |



Additional MIPS ALU Requirements



- Booth's algorithm
 1. Depending on the current and previous bits, do one of the following:
 - 00: Middle of a string of 0s – no arithmetic operations.
 - 01: End of a string of 1s – add the multiplicand to the left half of the product.
 - 10: Beginning of a string of 1s – subtract the multiplicand from the left half of the product.
 - 11: Middle of a string of 1s – no arithmetic operation
 2. As in the previous algorithm, shift the Product register right (arithmetic) 1 bit
- Booth Multiply
 - Modify Step 1 of the Shift/Add Multiply algorithm to consider 2 bits of the multiplier:
 - Instead of two alternatives, now there are four
 - The current bit and the bit to the right (i.e., **the current bit of the previous step**)
 - Modify Step 2 of Shift/Add Multiply algorithm to sign extend when the product is shifted right (arithmetic right shift, rather than logical right shift) because the product is a signed number.
 - Shift/Add Multiply algorithm and Booth share the same hardware, except Booth requires one extra flip-flop to remember the bit to the right of the current bit in the product register – which is the bit pushed out by the preceding right shift



Additional MIPS ALU Requirements



- Booth Example 1: $(2 * 7)$

Multiplicand $m = 0010$

Product $p = 0000\ 0111$

| Iteration | Multiplicand | Product | LSB-1 | Next |
|-----------------|--------------|------------------|-------|-----------------------------|
| 0. Init | 0010 | 0000 0111 | 0 | $10 \rightarrow$ sub |
| 1a. $P = P - m$ | - $m = 1110$ | 1110 0111 | 0 | Shift Right Arithmetic |
| 1b. | 0010 | 1111 0011 | 1 | 11 \rightarrow nop, shift |
| 2. | 0010 | 1111 1001 | 1 | 11 \rightarrow nop, shift |
| 3. | 0010 | 1111 1100 | 1 | 01 \rightarrow add |
| 4a. | 0010 | 0001 1100 | 1 | Shift |
| 4b. | 0010 | 0000 1110 | 0 | Done |

$$0000\ 1110 = 14$$



Additional MIPS ALU Requirements



- Booth Example 2: $(2 * -3)$

$$m = 0010$$

$$3=0011, -3 = 1101$$

$$p = 0000\ 1101$$

| Iteration | Multiplicand | Product | LSB-1 | Next |
|-----------------|--------------|-------------------|----------|------------------------|
| 0. Init | 0010 | 0000 1101 | 0 | $10 \rightarrow$ sub |
| 1a. $P = P - m$ | $-m = 1110$ | 1110 1101 | 0 | Shift Right Arithmetic |
| 1b. | 0010 | 1111 0 110 | 1 | $01 \rightarrow$ add |
| 2a. | | 0001 0 110 | 1 | Shift Right Arithmetic |
| 2b. | | 0000 10 11 | 0 | $10 \rightarrow$ sub |
| 3a. $P = P - m$ | $-m = 1110$ | 1110 10 11 | 0 | Shift Right Arithmetic |
| 3b. | 0010 | 1111 010 1 | 1 | $11 \rightarrow$ nop |
| 4a. | 0010 | 1111 010 1 | 1 | Shift Right Arithmetic |
| 4b. | 0010 | 1111 1010 | 1 | Done |

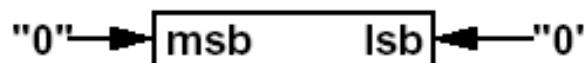
$$1111\ 1010 = -6$$

Additional MIPS ALU Requirements

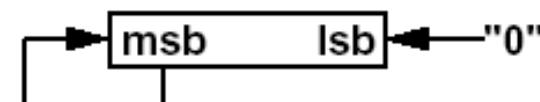


- Shifters

- sll, srl, sra MIPS instructions – shift left/right/right arithmetic by 0 to 31 bits



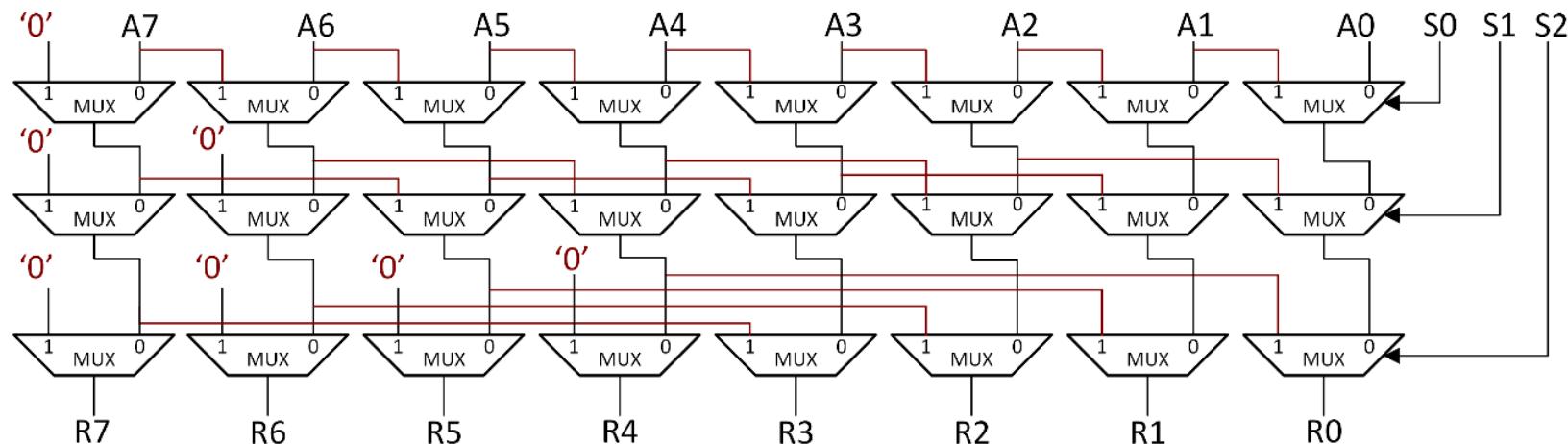
Logical: value shifted in is always "0"



Arithmetic: on right shift the sign bit is extended

Note: these are single bit shifts. Instructions can request 0 to 31 bits to be shifted

Combination shifter implemented with MUXes



How many levels for a 32-bit shifter?

If we added Right-to-left connections we could support Rotate operations (not implemented in MIPS)



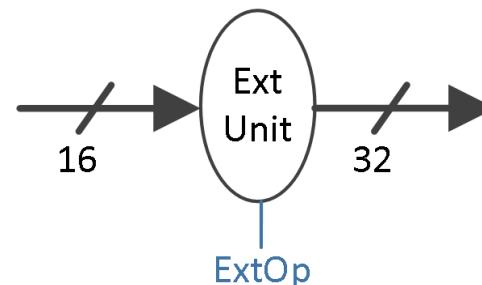
Additional MIPS ALU Requirements



- Sign / Zero Extender

- MIPS instructions use:

- Zero Extended (logic) operands ori: RF[rt] \leftarrow RF[rs] | Z_Ext(imm)
 - Sign Extended (arithmetic) operands lw: RF[rt] \leftarrow M[RF[rs] + S_Ext(imm)]



- Z_Ext (zero extension):

- The Extender takes a 16-bit number, imm[15:0] and extends it with 0's (extension in unsigned form) if the control line ExtOp = 0
 - $Z_Ext(imm16) = 0_{31} \dots 0_{16} | imm_{15} \dots imm_0$

- S_Ext (sign extension):

- The Extender takes a 16-bit number, imm[15:0] and extends it with the imm₁₅ bit if the control line ExtOp = 1
 - $S_Ext(imm16) = 0_{31} \dots 0_{16} | imm_{15} \dots imm_0 \quad \text{if } imm_{15} = 0$
 - $S_Ext(imm16) = 1_{31} \dots 1_{16} | imm_{15} \dots imm_0 \quad \text{if } imm_{15} = 1$



Problems – Homework



- Design an 8-bit ALU for the following operations: $A + B$, $A - B$, $\text{Incr}A$, $\text{Decr}A$, $\text{Pass}A$ and $\text{Negate}A$. Use a single adder circuit. Show the schematic with control signals and a table with control signal values for the required operations.
- Design an Add/Subtract unit that can work with 8, 16 and 32-bit data. You are given 32x1-bit full adders and the necessary auxiliary circuits. Show the block diagram and the control signals for 4x8-bit, 2x16-bit, 1x32-bit Add/Subtract operations.
- Describe the Booth multiplication method. Give a numerical example.



References



1. D. A. Patterson, J. L. Hennessy, “Computer Organization and Design: The Hardware/Software Interface”, 5th edition, ed. Morgan–Kaufmann, 2013.
2. D. A. Patterson and J. L. Hennessy, “Computer Organization and Design: A Quantitative Approach”, 5th edition, ed. Morgan-Kaufmann, 2011.
3. MIPS32™ Architecture for Programmers, Volume I: “Introduction to the MIPS32™ Architecture”.
4. MIPS32™ Architecture for Programmers Volume II: “The MIPS32™ Instruction Set”.



Computer Architecture

Lecturer: Mihai Negru

2nd Year, Computer Science

Lecture 6: Multi-Cycle CPU Design

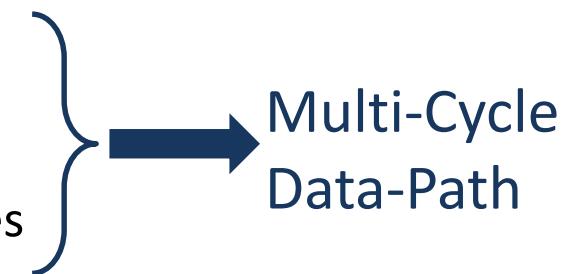
<http://users.utcluj.ro/~negrum/>



Multi-Cycle Processor Design

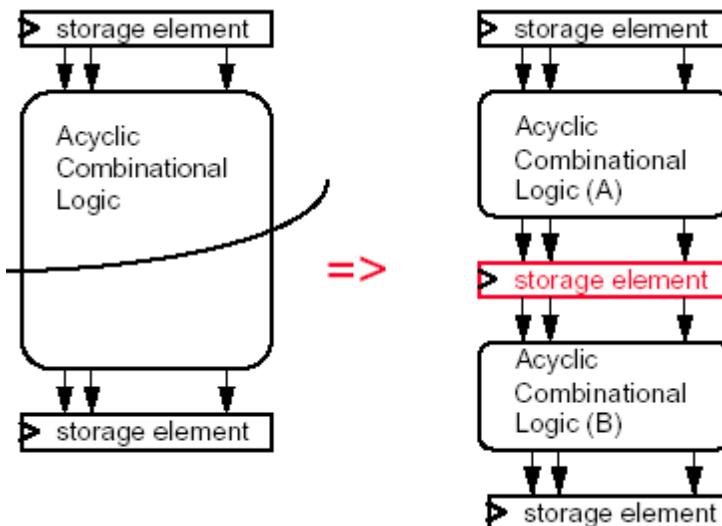


- Step-by-step Processor Design → Multi cycle MIPS
 - Step 1: ISA → Abstract RTL
 - Step 2: Components of the Data-Path
 - Step 3: RTL + Components → Data-Path
 - Step 4: Data-Path + Abstract RTL → Concrete RTL
 - Step 5: Concrete RTL → Control
- Single Cycle Problems
 - Long Cycle Time
 - All instructions take as much time as the slowest
 - What happens for floating point?
 - Waste of area: no component reuse
- One Possible Solution
 - use a “smaller” cycle time
 - different instructions take different numbers of cycles





- Cut the combinational dependency graph and insert registers
 - Do the same amount of work in two fast cycles, rather than one slow one



Break up the long combinational stages

| Limits on Cycle Time in different stages | | |
|--|---|--------------------------------|
| Next address logic | $PC \leftarrow \text{branch ? } PC + \text{offset : } PC + 4$ | Address logic computation time |
| Instruction Fetch | $IR \leftarrow M[PC]$ | Memory access time |
| Register Access | $A \leftarrow RF[rs]$ | Register file access time |
| ALU operation | $RF[rd] \leftarrow A + B$ | ALU operation delay |



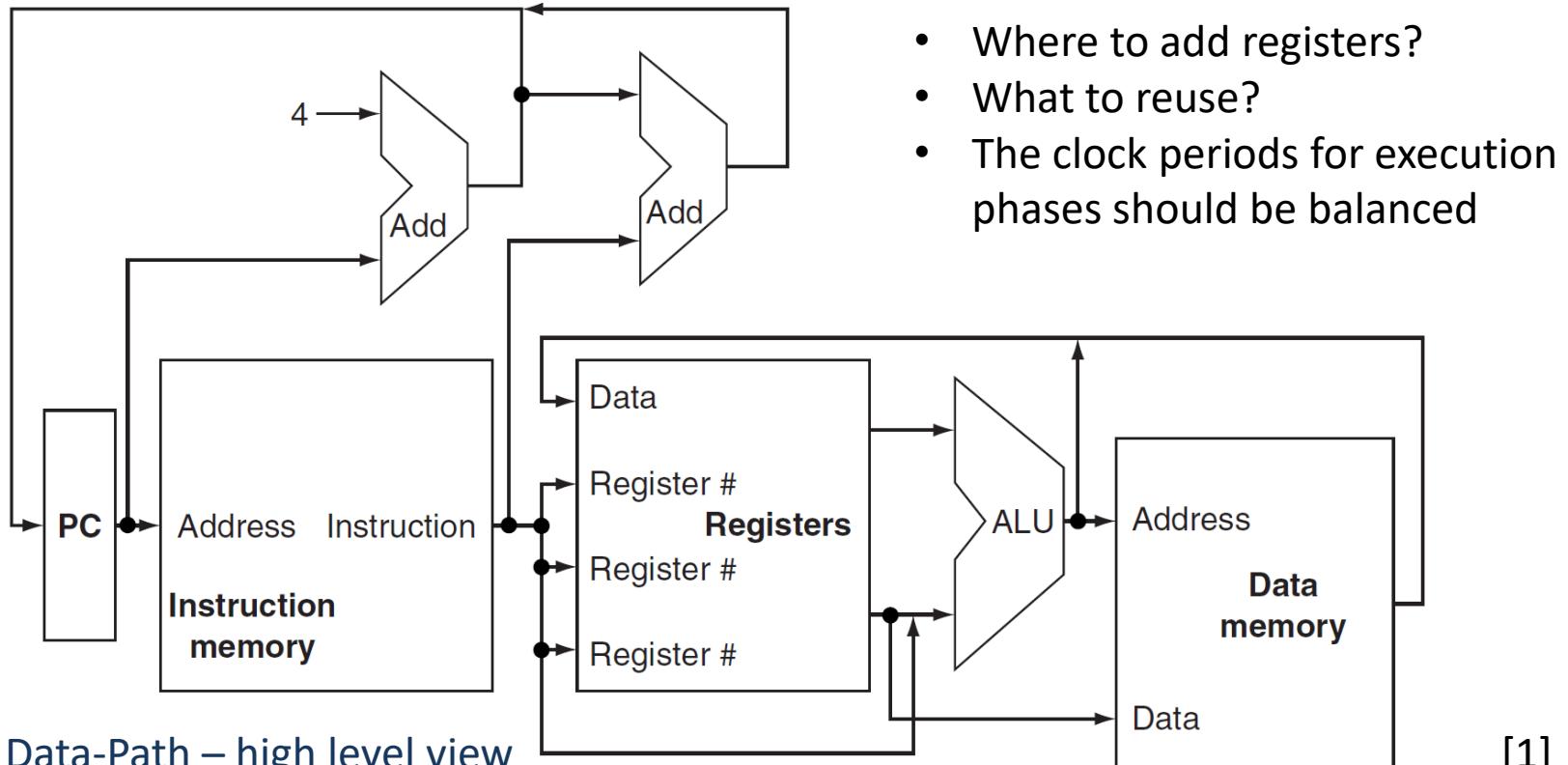
Multi-Cycle Approach



- Break up the instructions into steps, each step takes one cycle
 - Balance the amount of work to be done.
 - Restrict each cycle to use only one major functional unit.
- At the end of a cycle
 - Store values for use in later cycles.
 - Introduce additional “internal” registers (not programmer visible).
- Reuse functional units
 - ALU used to compute address and to increment PC (beside usual ALU operations)
 - Only one Memory used for Instruction and Data!
- Use a finite state machine (FSM) for control



- Step 1: ISA → Abstract RTL
 - The same instructions as for the Single-Cycle MIPS
- Step 2: Components of the Data-Path
 - Partitioning the Single-Cycle Data-Path

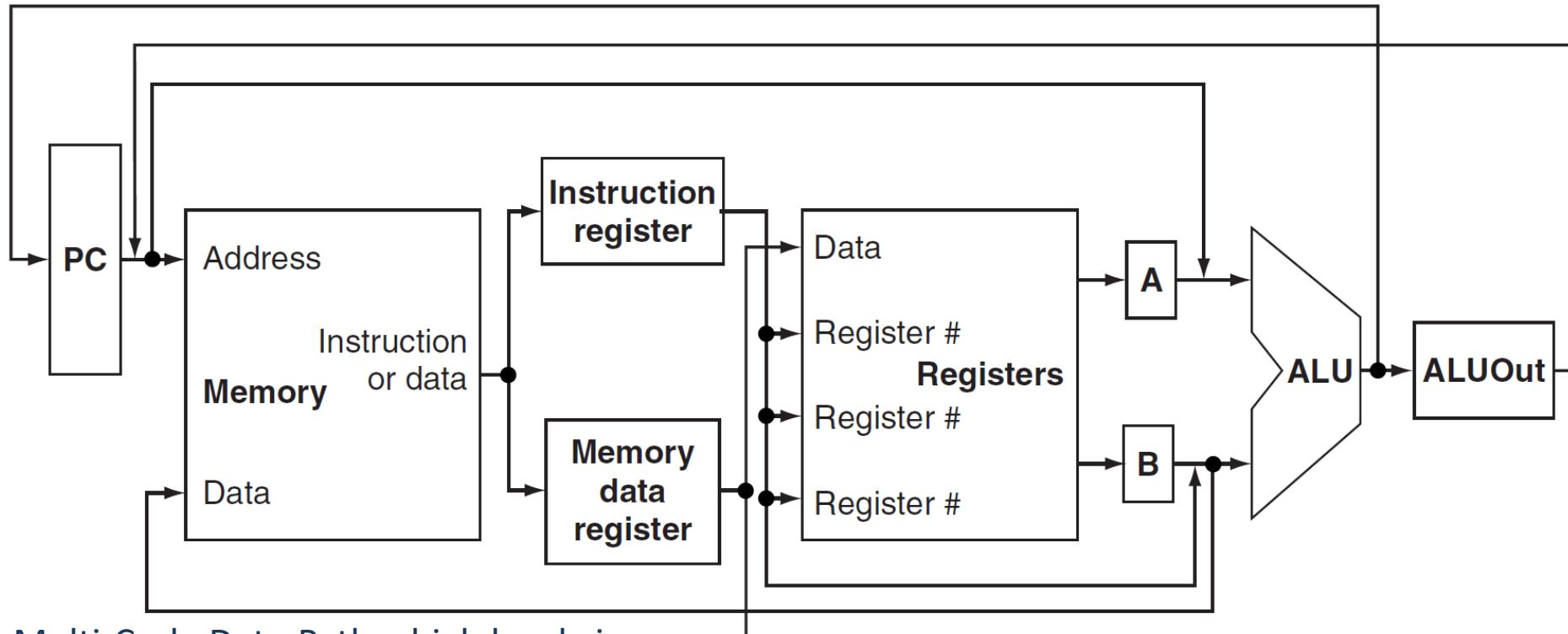


Single-Cycle Data-Path – high level view

[1]



Multi-Cycle CPU Design – Step 2

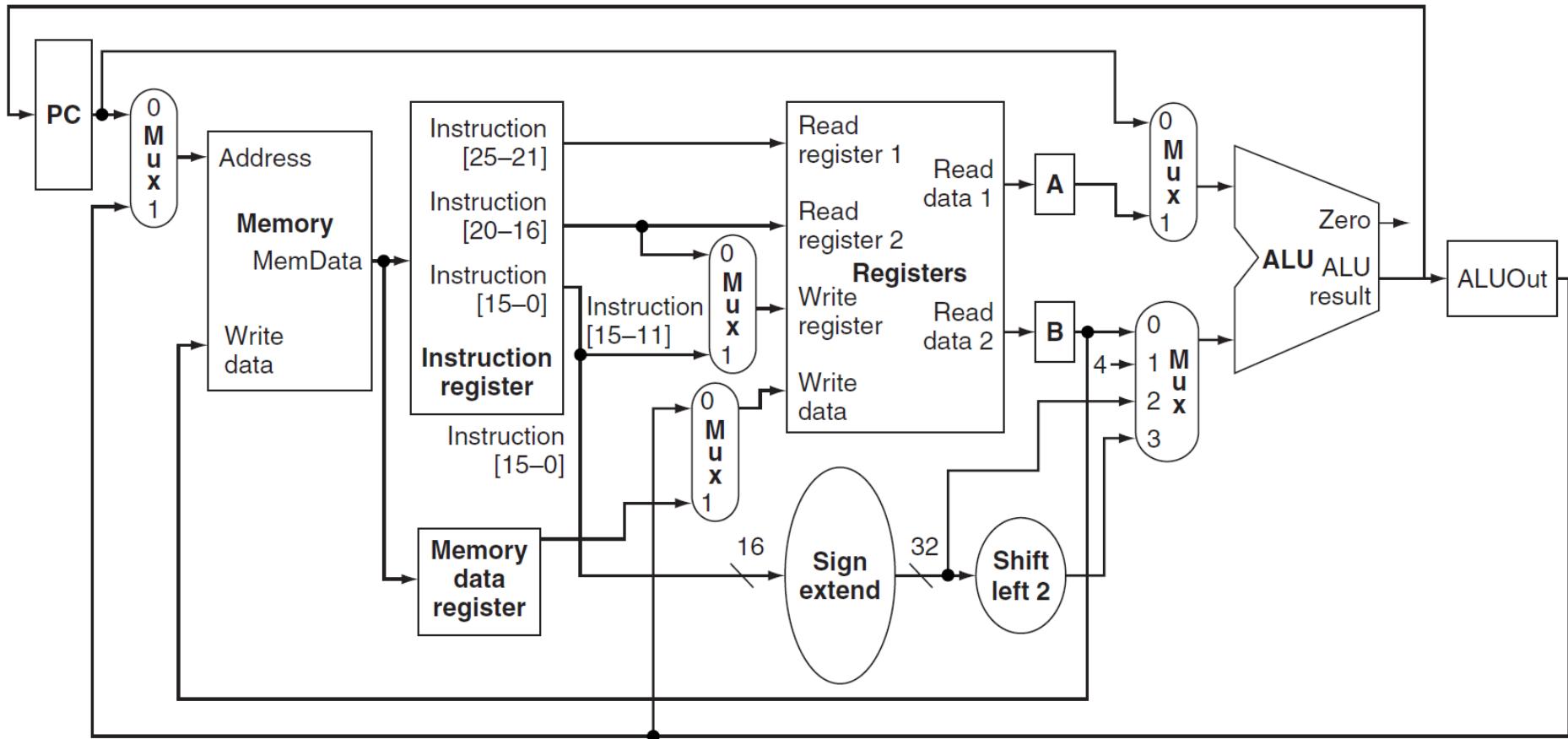


Multi-Cycle Data-Path – high level view

[1]

- Added registers (not visible to the programmer):
 - IR – Instruction Register; MDR – Memory Data Register
 - A, B – register file read data registers; ALUOut – ALU output register.
 - Data used by subsequent instructions are stored in programmer visible registers (i.e., register file, PC) or memory.
- Memory and ALU reused

Multi-Cycle CPU Design – Step 2

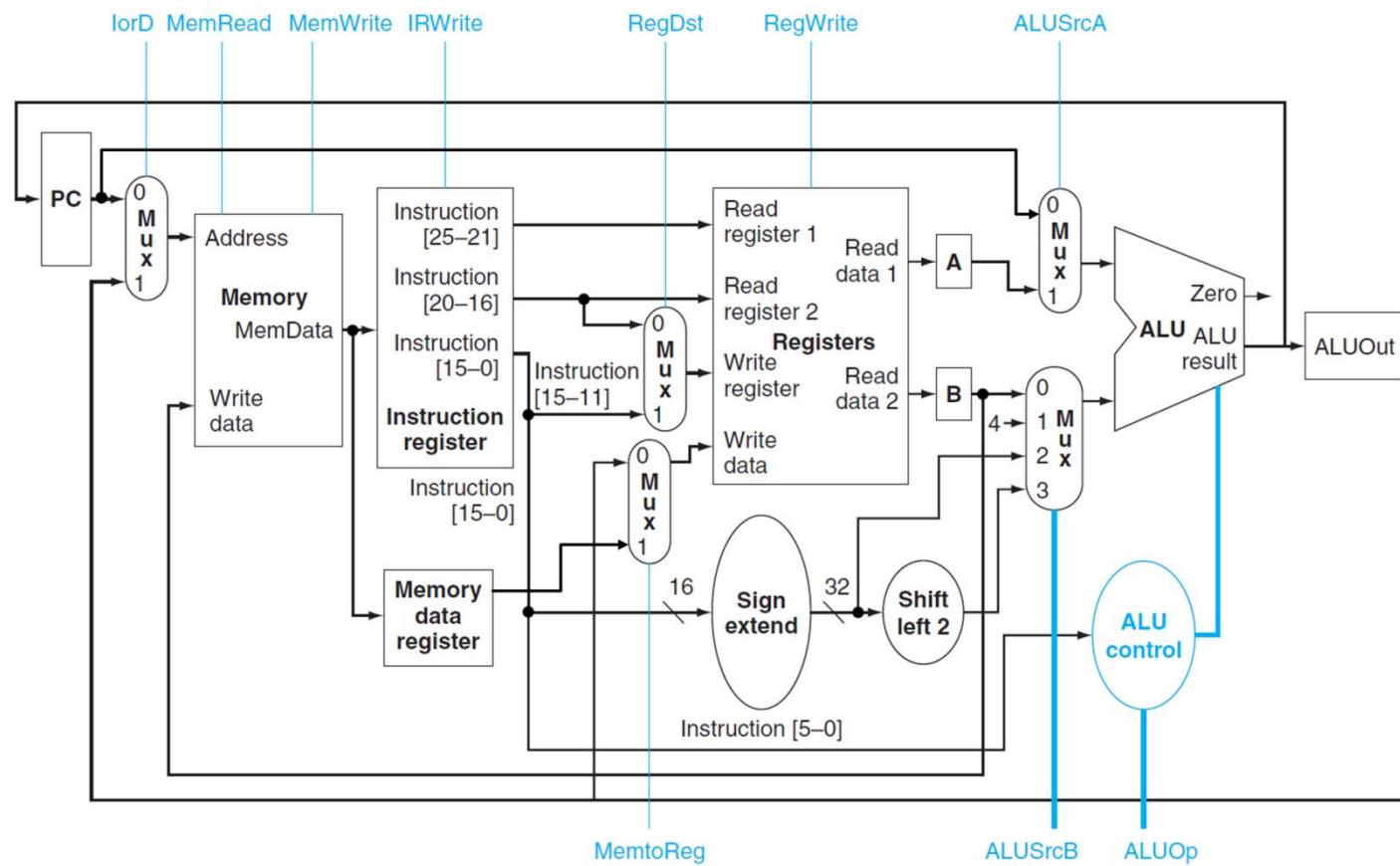


Multi-Cycle Data-Path derived from Single-Cycle MIPS

[1]



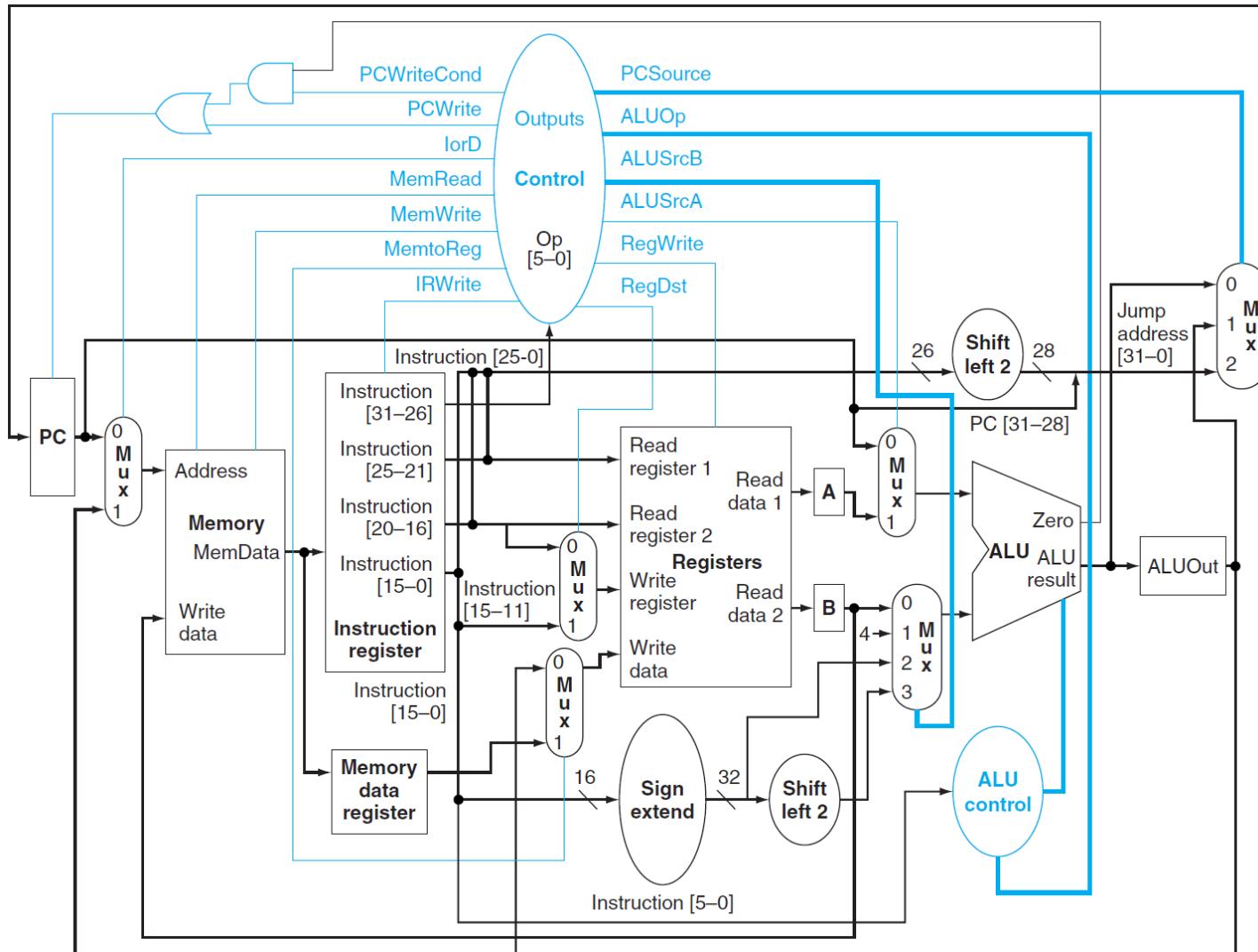
- Step 3: RTL + Components → Data-Path
 - We connect the components to build the Data-Path, and specify the Control Signals
 - Instruction Register (IR): IR[25:21] → rs, IR[20:16] → rt, IR[15:11] → rd



Multi-Cycle Data-Path with Control Signals

[1]

Multi-Cycle CPU Design – Step 3



[1]

Multi-Cycle Data-Path with Control Unit



Multi-Cycle CPU Design – Step 3



| Signal name | Effect when deasserted (=0) | Effect when asserted (=1) |
|--------------------|---|---|
| RegDst | The register destination number for the write register comes from the rt field (instruction bits 20:16) | The register destination number for the write register comes from the rd field (instruction bits 15:11) |
| RegWrite | None | The register on the write register input is written with the value on the Write data input |
| ALUSrcA | The first ALU operand is the PC (default) | The first ALU operand is register A (i.e. R[rs]) |
| MemRead | None (default) | Content of memory specified by the address input are put on the memory data output |
| MemWrite | None (default) | Memory contents specified by the address inputs is replaced by the value on the Write data input |
| MemtoReg | The value fed to the register write data input comes from ALUOut register (default) | The value fed to the register write data input comes from data memory register (MDR) |
| IorD | The PC is used to supply the address to the memory unit (default) | The ALUOut register is used to supply the address to the memory unit |
| IRWrite | None (default) | The output of the memory is written into the Instruction Register (IR) |
| PCWrite | None (default) | The PC is written; the source is controlled by PC source |
| PCWriteCond | None (default) | The PC is written if the Zero output of the ALU is also active |

The Meaning of the 1-bit Control Signals



Multi-Cycle CPU Design – Step 3



| Signal Name | Value(Binary) | Effect |
|-------------|---------------|--|
| ALUOp | 00 | The ALU performs an add operation |
| | 01 | The ALU performs a subtract operation |
| | 10 | The function field of the instruction determines the ALU operation (R-Type) |
| ALUSrcB | 00 | The second input of the ALU comes from the B register |
| | 01 | The second input of the ALU is the constant 4 |
| | 10 | The second input of the ALU is the sign-extended 16-bit immediate filed of the instruction in IR |
| | 11 | The second input of the ALU is the sign-extended 16-bit immediate field of IR shifted left 2 bits |
| PCSource | 00 | Output of the ALU (PC + 4) is sent to the PC for writing |
| | 01 | The content of the ALUOut (the branch target address) is sent to the PC for writing |
| | 10 | The jump target address (IR[25:0] shifted left 2 bits and concatenated with PC+4[31:28]) is sent to the PC for writing |

The Meaning of the 2-bit Control Signals



Multi-Cycle CPU Design – Step 4



- Step 4: Data path + Abstract RTL → Concrete RTL
 - For the multi cycle data path we write the RTL codes of the basic instructions to establish the necessary Control Signal settings
 - Instructions from ISA perspective
 - RTL Abstract
 - Specifies the instruction independent of a concrete implementation
 - Example: arithmetic, R-type instruction
$$RF[rd] \leftarrow RF[rs] \text{ op } RF[rt]$$
 - RTL Concrete
 - Describes the execution phases of the instruction for a given implementation
 - Example: arithmetic, R-type instruction
$$\begin{aligned} T0 &\rightarrow IR \leftarrow M[PC] \\ T1 &\rightarrow A \leftarrow RF[rs], B \leftarrow RF[rt] \\ T2 &\rightarrow ALUOut \leftarrow A \text{ op } B \\ T3 &\rightarrow RF[rd] \leftarrow ALUOut \end{aligned}$$
- We forgot an important part of the definition!
- PC \leftarrow PC + 4



Multi-Cycle CPU Design – Step 4



IorD MemRead MemWrite IRWrite

RegDst

RegWrite

ALUSrcA

add \$rd, \$rs, \$rt

Abstract RTL:

$$RF[rd] \leftarrow RF[rs] + RF[rt], \\ PC \leftarrow PC + 4$$

Concrete RTL:

| | |
|------------|---|
| T0 → | $IR \leftarrow M[PC]$, $PC \leftarrow PC + 4$; |
| T1 → | $A \leftarrow RF[rs]$, $B \leftarrow RF[rt]$; |
| ADD & T2 → | $ALUOut \leftarrow A + B$; |
| ADD & T3 → | $RF[rd] \leftarrow ALUOut$; |

[1]

MemtoReg

ALUSrcB

ALUOp

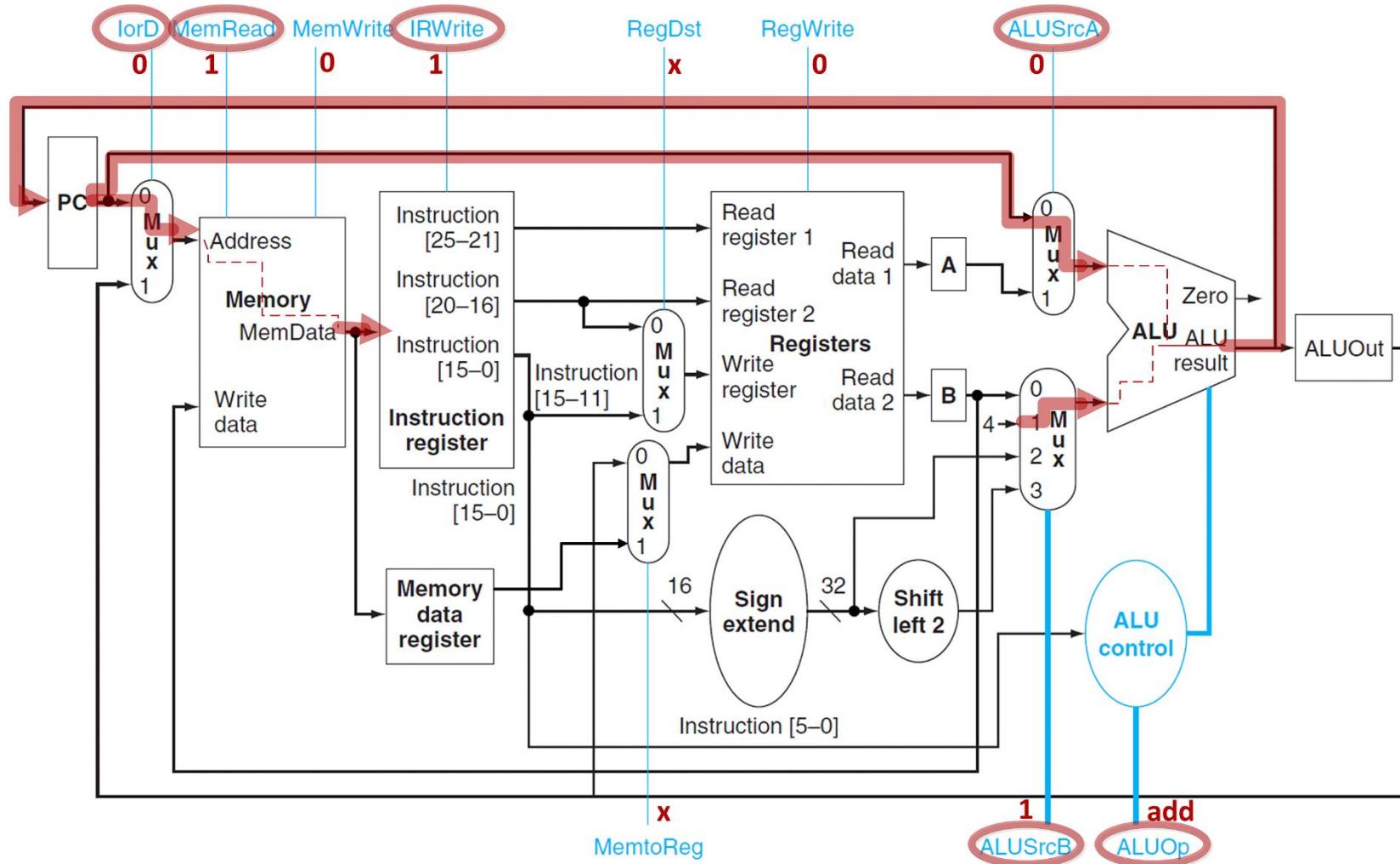
| | IorD | Mem Read | Mem Write | IR Write | Reg Dst | Mem toReg | Reg Write | Ext Op | ALU SrcA | ALU SrcB | ALU Op |
|----|------|----------|-----------|----------|---------|-----------|-----------|--------|----------|----------|--------|
| T0 | 0 | 1 | 0 | 1 | x | x | 0 | x | 0 | 1 | add |
| T1 | x | 0 | 0 | 0 | x | x | 0 | x | x | x | x |
| T2 | x | 0 | 0 | 0 | x | x | 0 | x | 1 | 0 | func |
| T3 | x | 0 | 0 | 0 | 1 | 0 | 1 | x | x | x | x |

Note: the operation is defined by the function field

Multi-Cycle CPU Design – Step 4



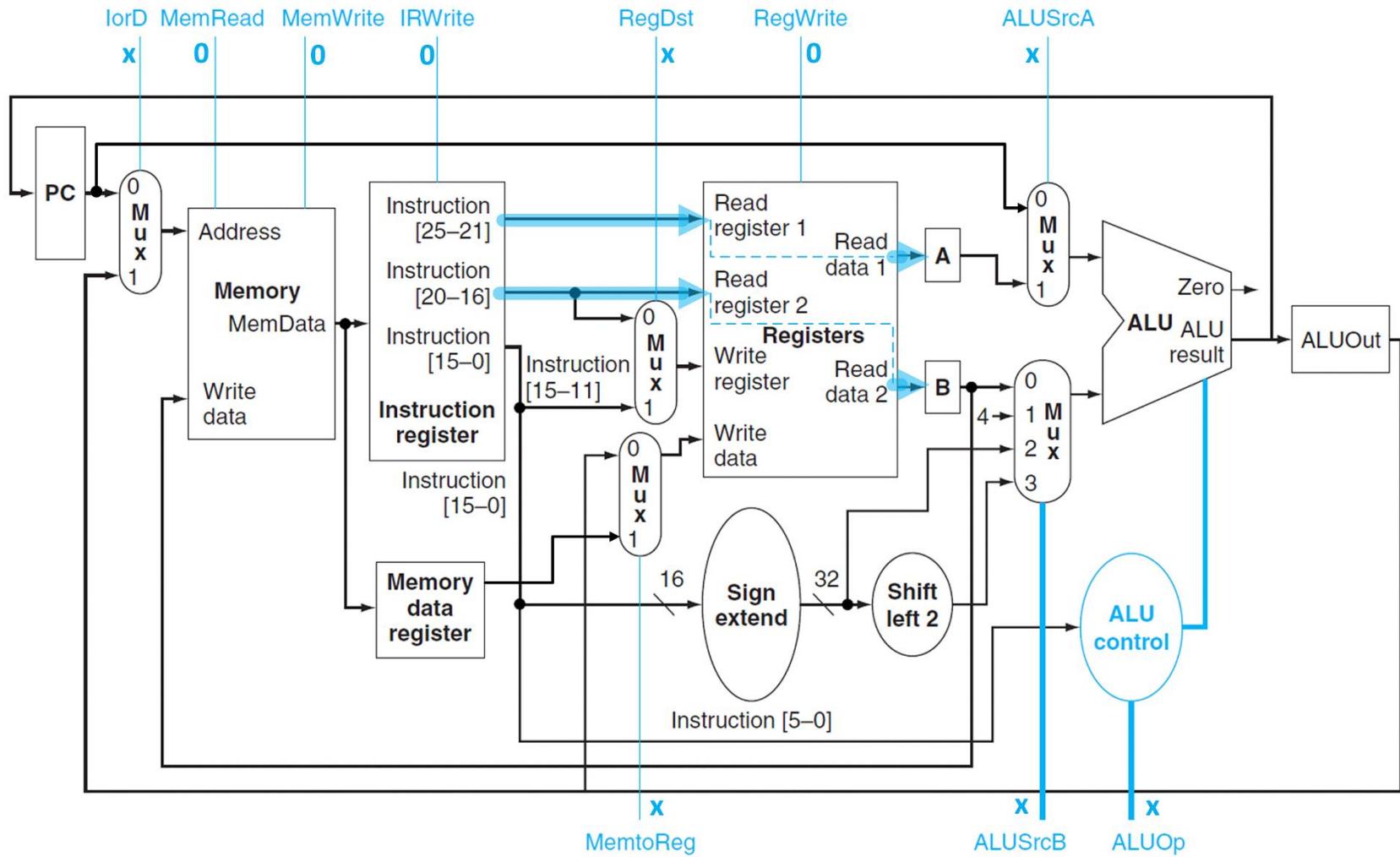
Add: $T_0 \rightarrow IR \leftarrow M[PC]$, $PC \leftarrow PC + 4$;



Multi-Cycle CPU Design – Step 4



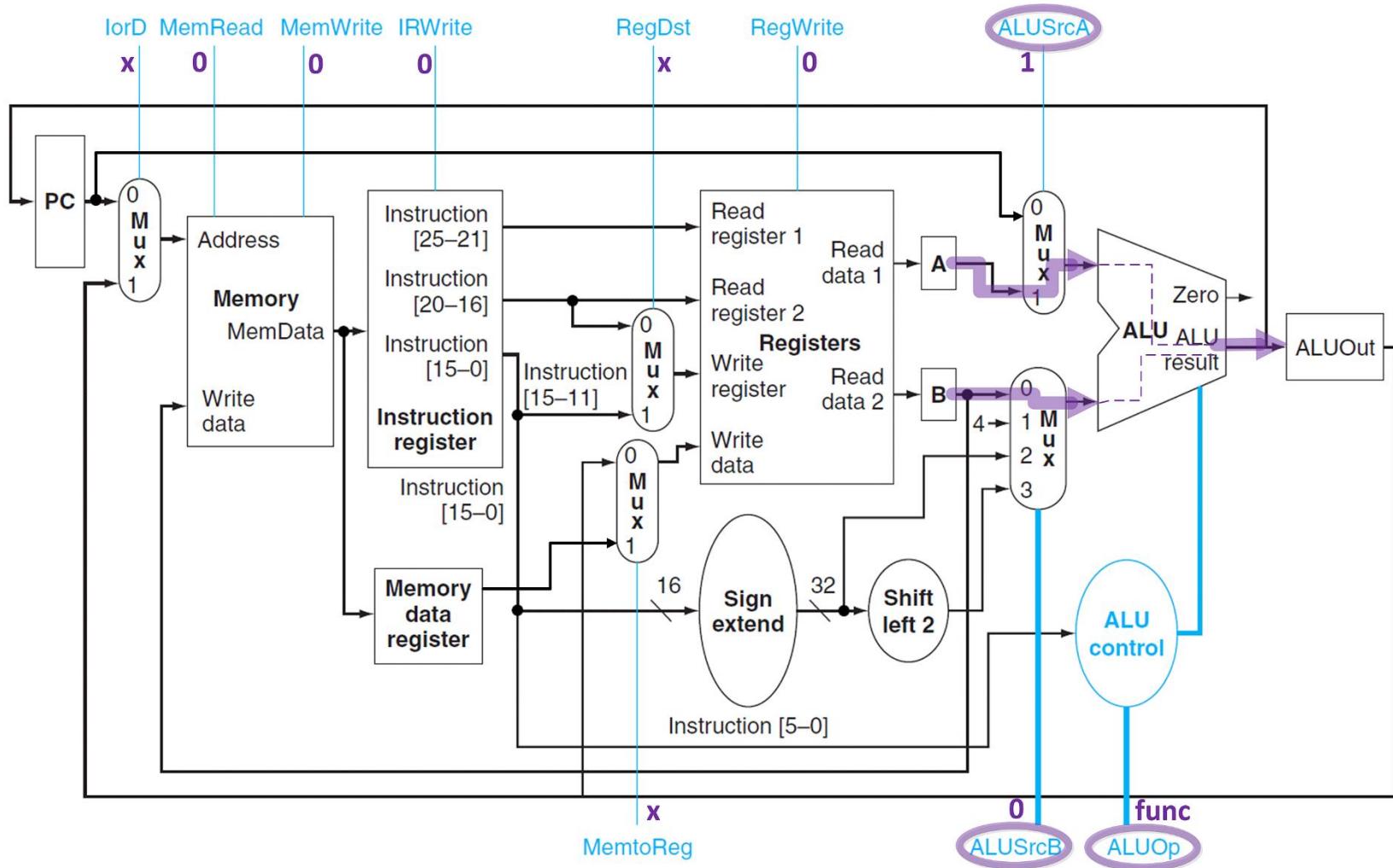
Add: $T1 \rightarrow A \leftarrow R[rs], B \leftarrow R[rt];$



Multi-Cycle CPU Design – Step 4



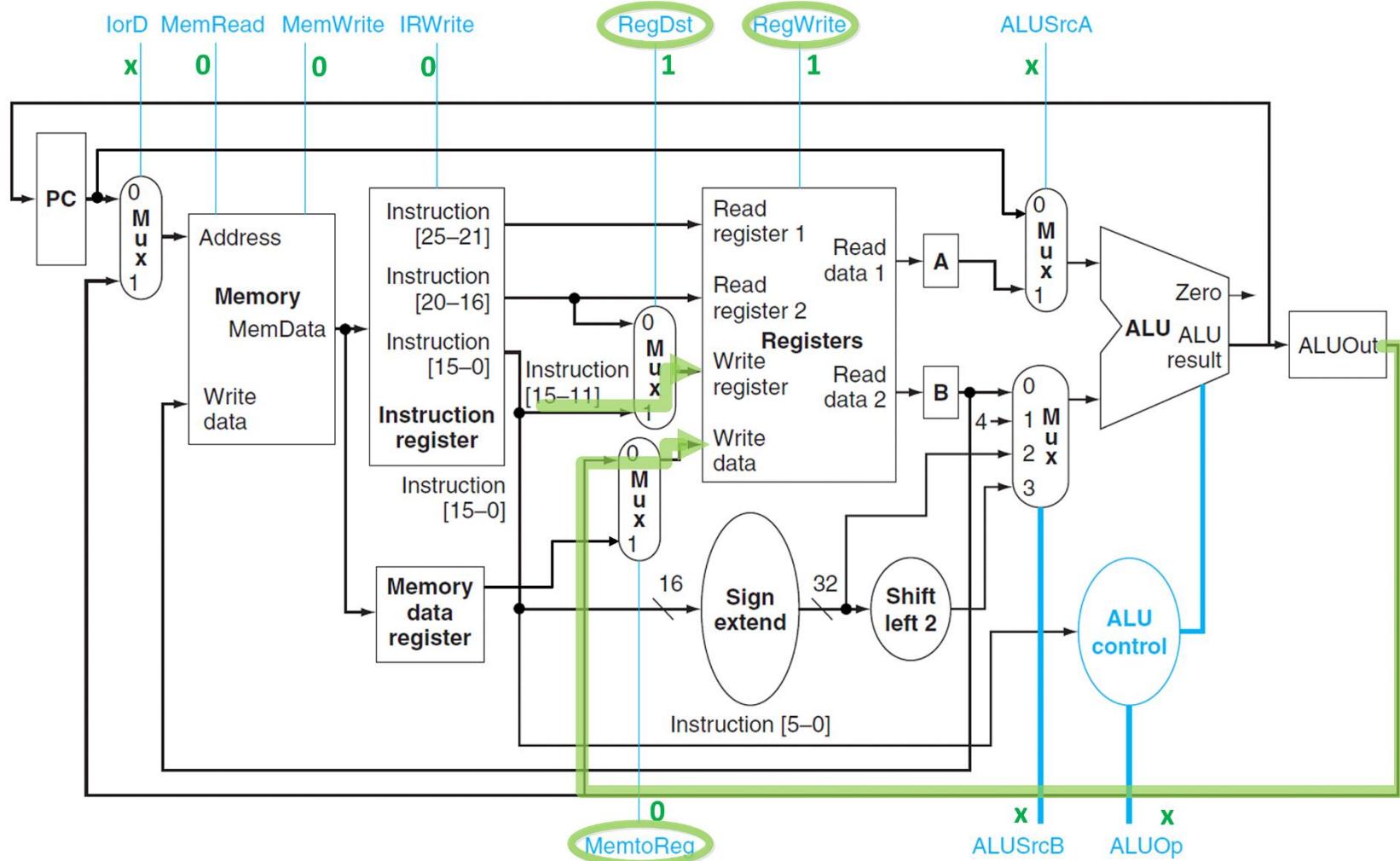
Add: ADD & T2 → ALUOut ← A + B;



Multi-Cycle CPU Design – Step 4



Add: ADD & T3 → R[rd] ← ALUOut;



Multi-Cycle CPU Design – Step 4

IorD MemRead MemWrite IRWrite

RegDst

RegWrite

ALUSrcA

ori \$rs, \$rt, imm

Abstract RTL:

$RF[rt] \leftarrow RF[rs] \mid Z_Ext(imm)$,
 $PC \leftarrow PC + 4$

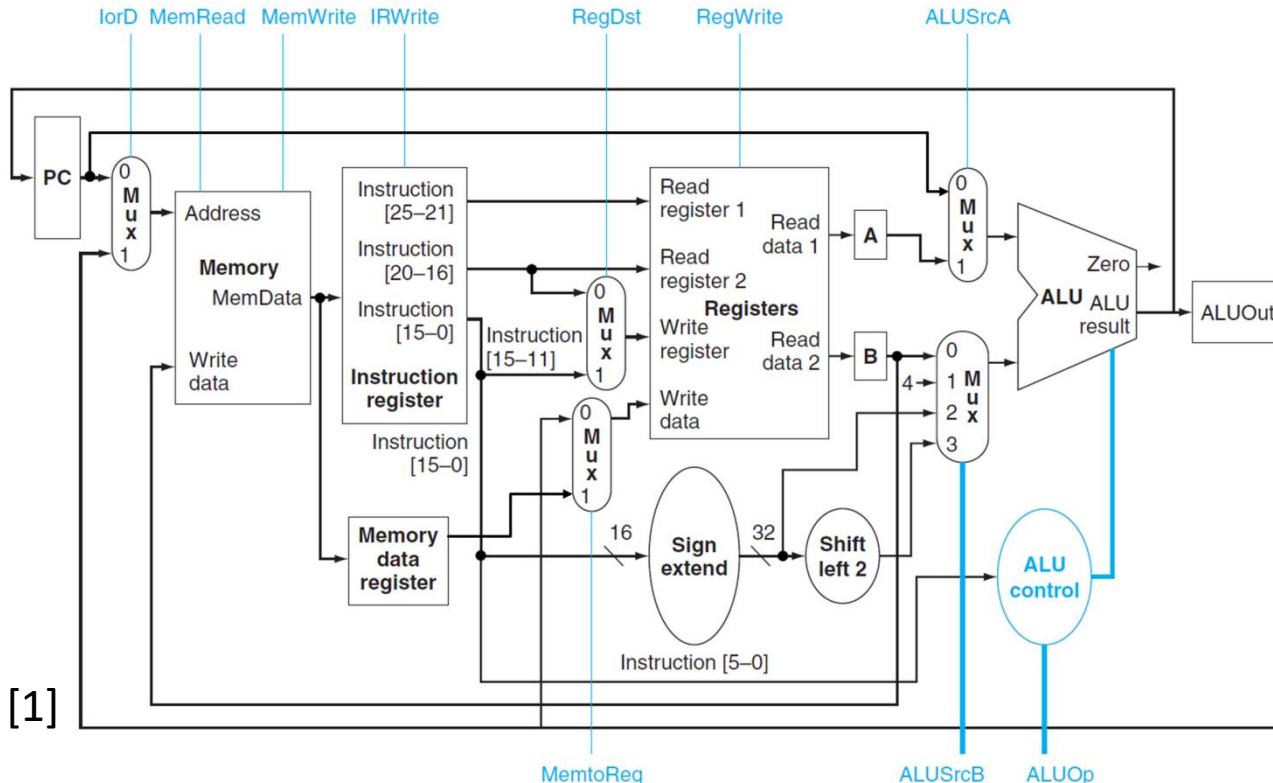
Concrete RTL:

T0 \rightarrow $IR \leftarrow M(PC)$,
 $PC \leftarrow PC + 4$;

T1 \rightarrow $A \leftarrow RF[rs]$,
 $B \leftarrow RF[rt]$;

ORI & T2 \rightarrow $ALUOut \leftarrow A \mid Z_Ext(imm)$;

ORI & T3 \rightarrow $RF[rt] \leftarrow ALUOut$;



MemtoReg

ALUSrcB

ALUOp

| | IorD | Mem Read | Mem Write | IR Write | Reg Dst | Mem toReg | Reg Write | Ext Op | ALU SrcA | ALU SrcB | ALU Op |
|----|------|----------|-----------|----------|---------|-----------|-----------|--------|----------|----------|--------|
| T0 | 0 | 1 | 0 | 1 | x | x | 0 | x | 0 | 1 | add |
| T1 | x | 0 | 0 | 0 | x | x | 0 | x | x | x | x |
| T2 | x | 0 | 0 | 0 | x | x | 0 | 0 | 1 | 2 | or |
| T3 | x | 0 | 0 | 0 | 0 | 0 | 1 | x | x | x | x |

Note: the operation is defined by the opcode field

Multi-Cycle CPU Design – Step 4



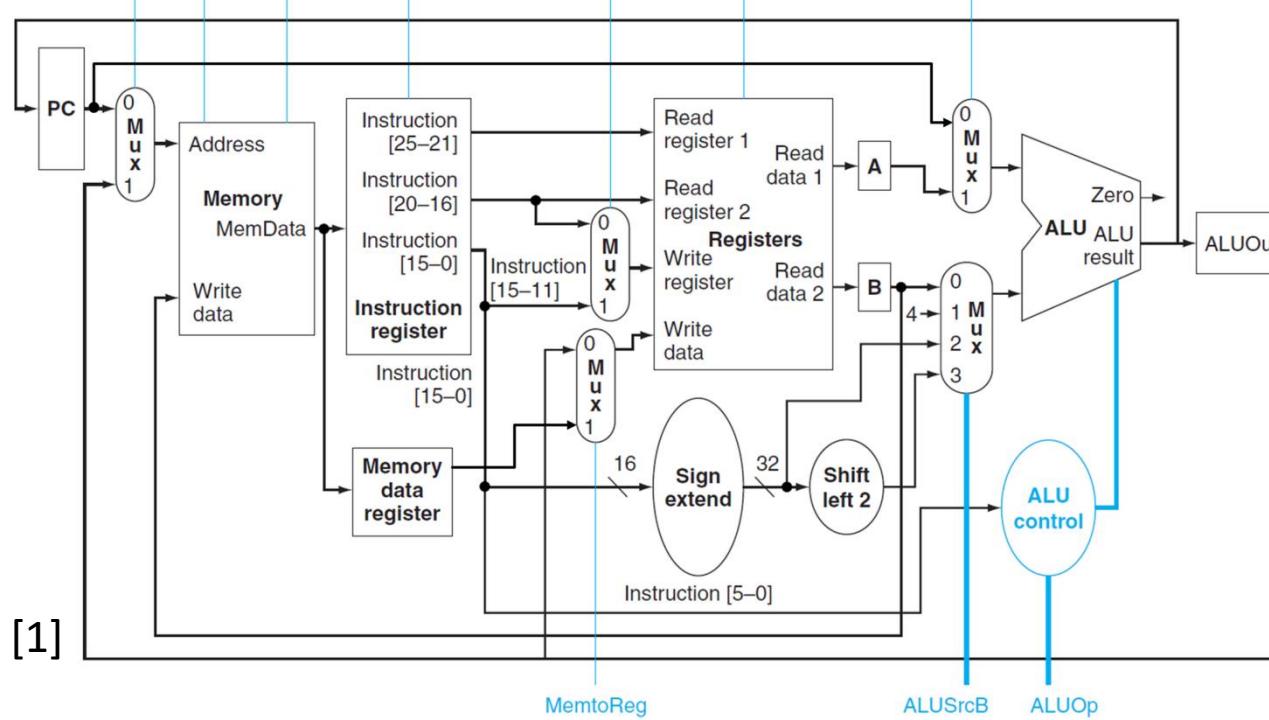
IorD MemRead MemWrite IRWrite

RegDst

RegWrite

ALUSrcA

lw \$rt, imm(\$rs)



MemtoReg

ALUSrcB

ALUOp

| | IorD | Mem Read | Mem Write | IR Write | Reg Dst | Mem toReg | Reg Write | Ext Op | ALU SrcA | ALU SrcB | ALU Op |
|----|------|----------|-----------|----------|---------|-----------|-----------|--------|----------|----------|--------|
| T0 | 0 | 1 | 0 | 1 | x | x | 0 | x | 0 | 1 | add |
| T1 | x | 0 | 0 | 0 | x | x | 0 | x | x | x | x |
| T2 | x | 0 | 0 | 0 | x | x | 0 | 1 | 1 | 2 | add |
| T3 | 1 | 1 | 0 | 0 | x | x | 0 | x | x | x | x |
| T4 | x | 0 | 0 | 0 | 0 | 1 | 1 | x | x | x | x |



Multi-Cycle CPU Design – Step 4



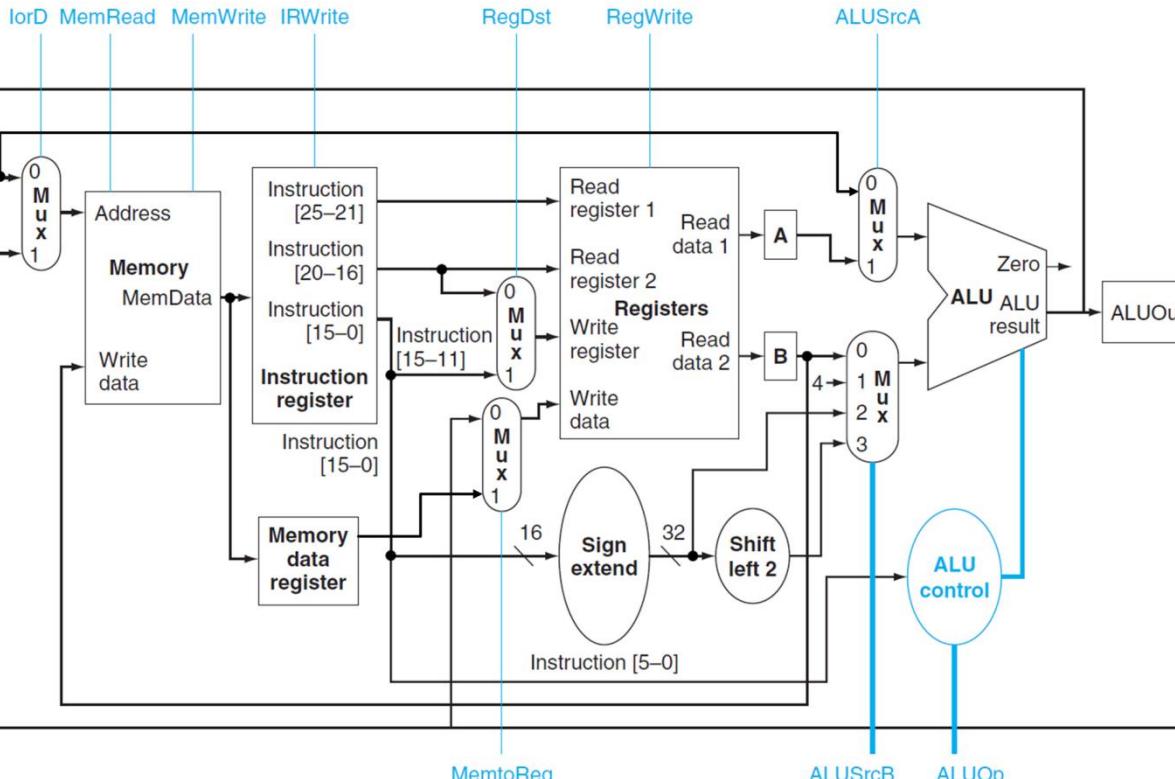
IorD MemRead MemWrite IRWrite

RegDst

RegWrite

ALUSrcA

sw \$rt, imm(\$rs)



Abstract RTL:

$M[RF[rs] + S_Ext(imm)] \leftarrow RF[rt],$
 $PC \leftarrow PC + 4$

Concrete RTL:

$T_0 \rightarrow IR \leftarrow M[PC],$
 $PC \leftarrow PC + 4;$

$T_1 \rightarrow A \leftarrow RF[rs],$
 $B \leftarrow RF[rt];$

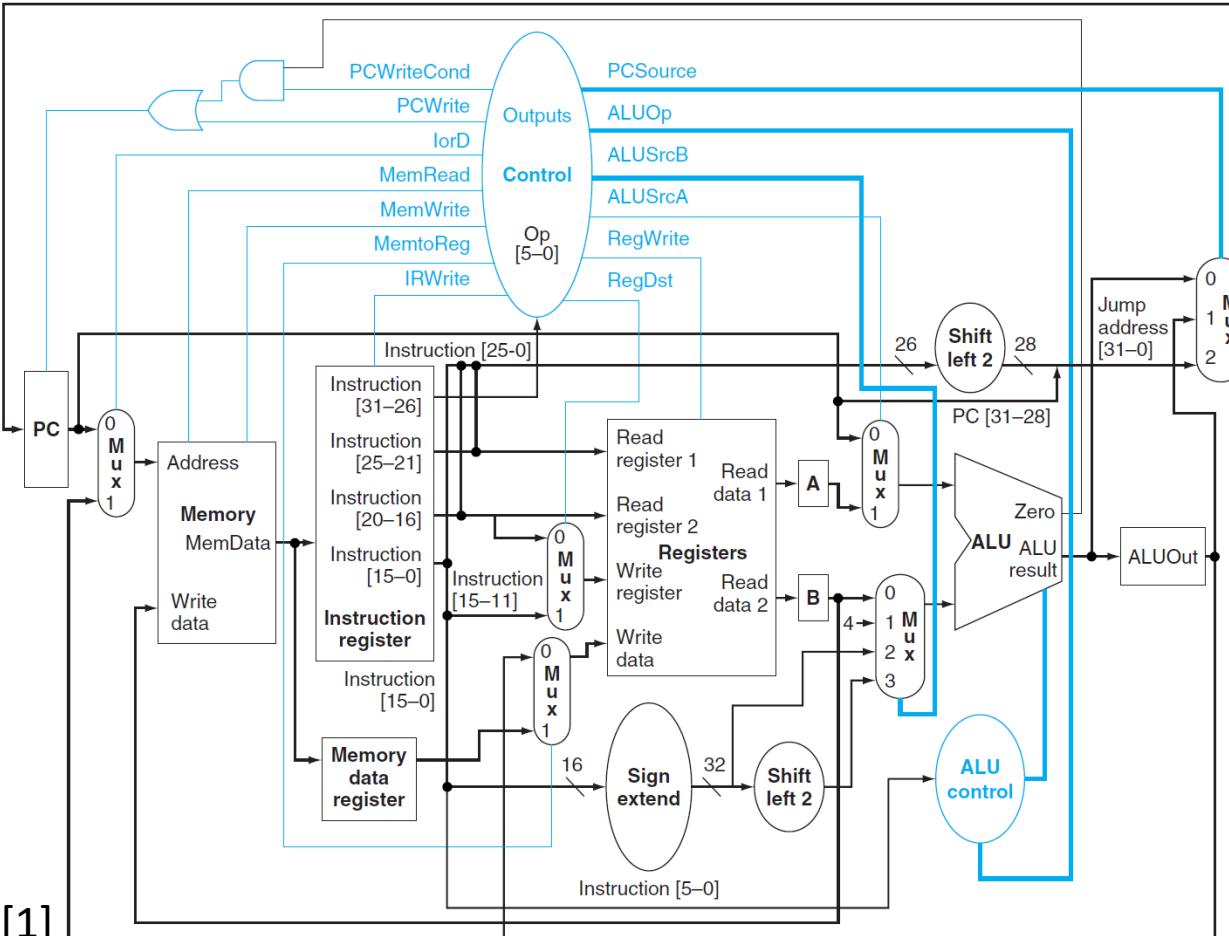
$SW \& T_2 \rightarrow ALUOut \leftarrow A + S_Ext(imm);$

$SW \& T_3 \rightarrow M[ALUOut] \leftarrow B$

| | IorD | Mem Read | Mem Write | IR Write | Reg Dst | Mem toReg | Reg Write | Ext Op | ALU SrcA | ALU SrcB | ALU Op |
|----|------|----------|-----------|----------|---------|-----------|-----------|--------|----------|----------|--------|
| T0 | 0 | 1 | 0 | 1 | x | x | 0 | x | 0 | 1 | add |
| T1 | x | 0 | 0 | 0 | x | x | 0 | x | x | x | x |
| T2 | x | 0 | 0 | 0 | x | x | 0 | 1 | 1 | 2 | add |
| T3 | 1 | 0 | 1 | 0 | x | x | 0 | x | x | x | x |



Multi-Cycle CPU Design – Step 4



beq \$rt, \$rs, imm

Abstract RTL:

```
If(RF[rs] == RF[rt]) then
    PC ← PC + 4 + S_Ext(imm) << 2
else
    PC ← PC + 4
```

Concrete RTL:

| | |
|------|-----------------------------|
| T0 → | IR ← M[PC], PC ← PC + 4; |
|------|-----------------------------|

| | |
|------|----------------------------|
| T1 → | A ← RF[rs], B ← RF[rt]; |
|------|----------------------------|

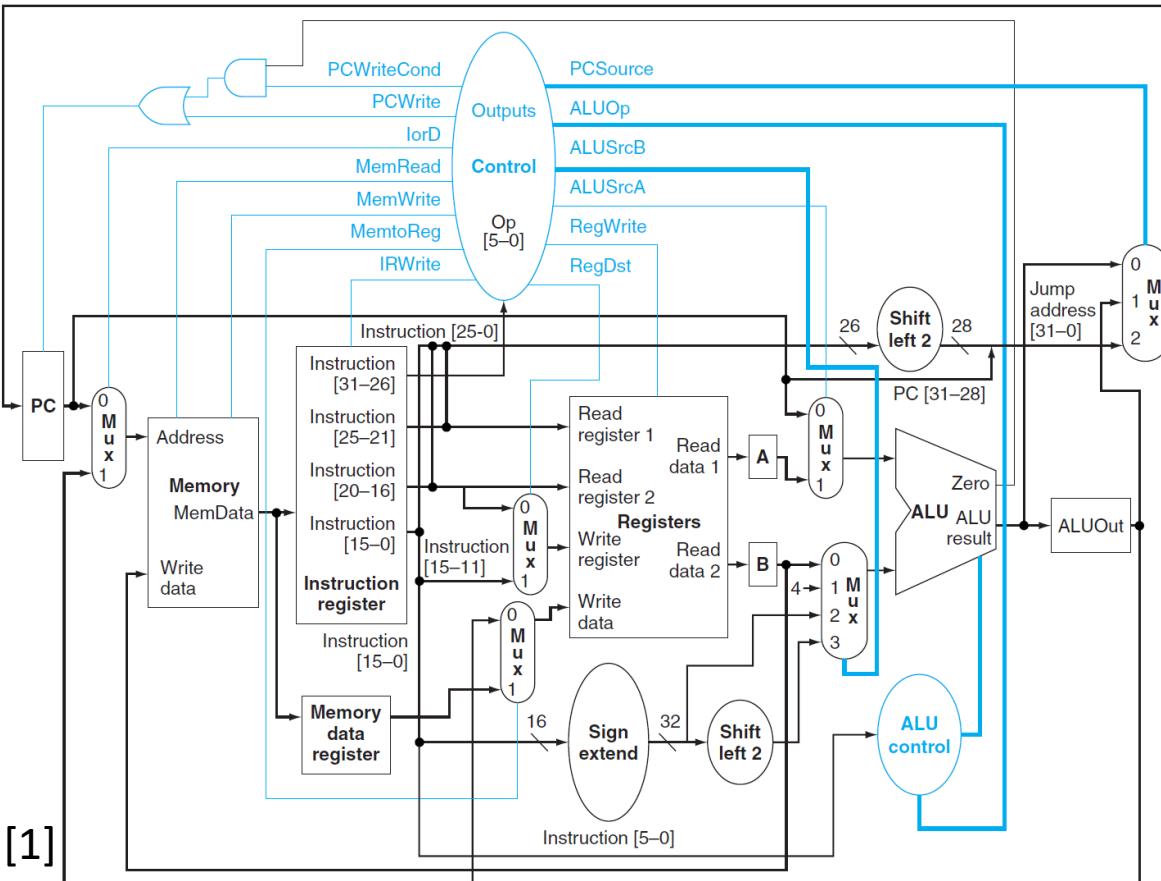
| | |
|------------|-----------------------------------|
| BEQ & T2 → | ALUOut ← PC + S_Ext(imm) << 2; |
|------------|-----------------------------------|

| | |
|---------------------|--------------|
| BEQ & T3 (A == B) → | PC ← ALUOut; |
|---------------------|--------------|

Note: T1 and T2 can be executed in parallel, in the same clock period!



Multi-Cycle CPU Design – Step 4



beq \$rt, \$rs, imm

Abstract RTL:

```
If(RF[rs] == RF[rt]) then
    PC ← PC + 4 + S_Ext(imm) << 2
else
    PC ← PC + 4
```

Concrete RTL:

| | |
|--|--|
| T0 → | $IR \leftarrow M[PC],$ $PC \leftarrow PC + 4;$ |
| T1 → | $A \leftarrow RF[rs], B \leftarrow RF[rt],$ $ALUOut \leftarrow PC + S_Ext(imm) << 2;$ |
| BEQ & T2 (A == B) → $PC \leftarrow ALUOut;$ | |
| jmp → Homework | |

| | IorD | Mem Read | Mem Write | IR Write | Reg Dst | Mem toReg | Reg Write | Ext Op | ALU SrcA | ALU SrcB | ALU Op | PC Src | PC WrCd | PC Wr |
|----|------|----------|-----------|----------|---------|-----------|-----------|--------|----------|----------|--------|--------|---------|-------|
| T0 | 0 | 1 | 0 | 1 | x | x | 0 | x | 0 | 1 | add | 0 | 0 | 1 |
| T1 | x | 0 | 0 | 0 | x | x | 0 | 1 | 0 | 3 | add | x | 0 | 0 |
| T2 | x | 0 | 0 | 0 | x | x | 0 | x | 1 | 0 | sub | 1 | 1 | 0 |



Multi-Cycle CPU Design – Summary



- Five Execution Phases
 - Instruction Fetch
 - Instruction Decode and Register Fetch
 - Execution, Memory Address Computation, or Branch / Jump Completion
 - Memory Access or Arithmetical – Logical instruction completion
 - Write-back
- Instructions take from 3 to 5 clock cycles
- In one clock cycle all operations are done in parallel, not sequential!
 - $T_0 \rightarrow IR \leftarrow M[PC]$ and $PC \leftarrow PC+4$ are done simultaneously
- Between Clock T1 and Clock T2 the control unit will select the next step in accordance to the instruction type



Multi-Cycle CPU Design – Summary

| Step / Cycle | Action for R-type instructions | Action for ORI instruction | Action for memory reference instructions | Action for branches | Action for jumps |
|--|---|---|--|---|---|
| T0: Instruction Fetch | $IR \leftarrow M[PC]$ $PC \leftarrow PC + 4$ | | | | |
| T1: Instruction decode / register Fetch | $A \leftarrow RF[IR[25:21]]$ $B \leftarrow RF[IR[20:16]]$ $ALUOut \leftarrow PC + S_Ext(IR[15:0] \ll 2)$ | | | | |
| T2: Execution, address computation, branch / jump completion | $ALUOut \leftarrow A \text{ op } B$ | $ALUOut \leftarrow A \text{ OR } Z_Ext(Imm16)$ | $ALUOut \leftarrow A + S_Ext(IR[15:0])$ | If ($A == B$) $PC \leftarrow ALUOut$ | $PC \leftarrow PC[31:28] IR[25:0] \ll 2$ |
| T3: Memory access or R-type completion | $RF[IR[15:11] \leftarrow ALUOut$ | $RF[IR[20:16]] \leftarrow ALUOut;$ | LW: $MDR \leftarrow M[ALUOut]$ SW: $M[ALUOut] \leftarrow B$ | | |
| T4: Memory read completion | | | LW: $RF[IR[20:16]] \leftarrow MDR$ | | |



Multi-Cycle CPU Design – Control Signals



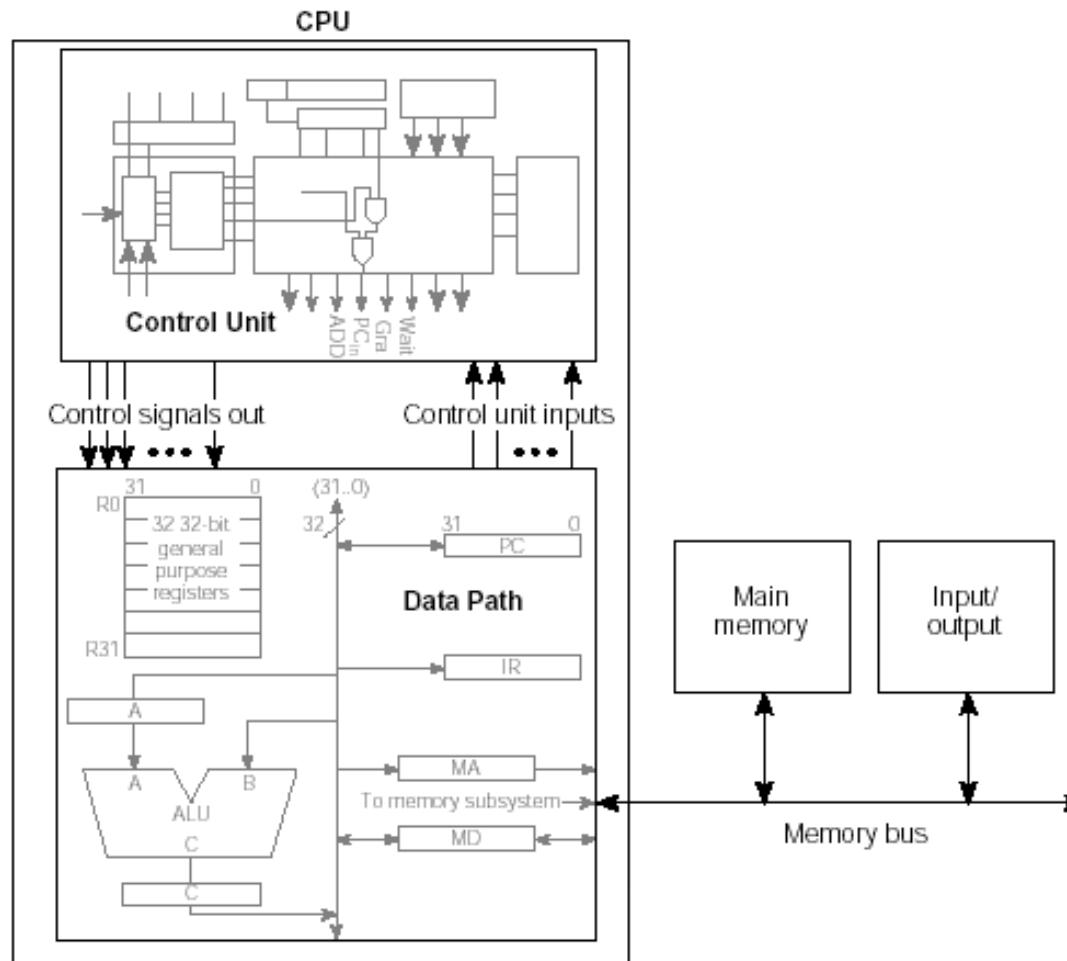
| T | IorD | Mem Read | Mem Write | IR Write | Reg Dst | Mem toReg | Reg Write | Ext Op | ALU SrcA | ALU SrcB | ALU Op | PC Src | PC WrCd | PC Wr | |
|----|------|----------|-----------|----------|---------|-----------|-----------|--------|----------|----------|--------|--------|---------|-------|--------|
| T0 | 0 | 1 | 0 | 1 | x | x | 0 | x | 0 | 1 | add | 0 | 0 | 1 | IF |
| T1 | x | 0 | 0 | 0 | x | x | 0 | 1 | 1 | 3 | add | x | 0 | 0 | ID |
| T2 | x | 0 | 0 | 0 | x | x | 0 | x | 1 | 0 | fun | x | 0 | 0 | Ex R-T |
| T3 | x | 0 | 0 | 0 | 1 | 0 | 1 | x | x | x | x | x | 0 | 0 | Wb R-T |
| T2 | x | 0 | 0 | 0 | x | x | 0 | 0 | 1 | 2 | or | x | 0 | 0 | Ex ORI |
| T3 | x | 0 | 0 | 0 | 0 | 0 | 1 | x | x | x | x | x | 0 | 0 | Wb ORI |
| T2 | x | 0 | 0 | 0 | x | x | 0 | 1 | 1 | 2 | add | x | 0 | 0 | Ex LW |
| T3 | 1 | 1 | 0 | 0 | x | x | 0 | x | x | x | x | x | 0 | 0 | M LW |
| T4 | x | 0 | 0 | 0 | 0 | 1 | 1 | x | x | x | x | x | 0 | 0 | Wb LW |
| T2 | x | 0 | 0 | 0 | x | x | 0 | 1 | 1 | 2 | add | x | 0 | 0 | Ex SW |
| T3 | 1 | 0 | 1 | 0 | x | x | 0 | x | x | x | x | x | 0 | 0 | M SW |
| T2 | x | 0 | 0 | 0 | x | x | 0 | x | x | x | sub | 1 | 1 | 0 | Ex BEQ |
| T2 | x | 0 | 0 | 0 | x | x | 0 | x | x | x | x | 2 | 0 | 1 | Ex J |

Table 1: The Values of the Control Signals in each Clock Cycle

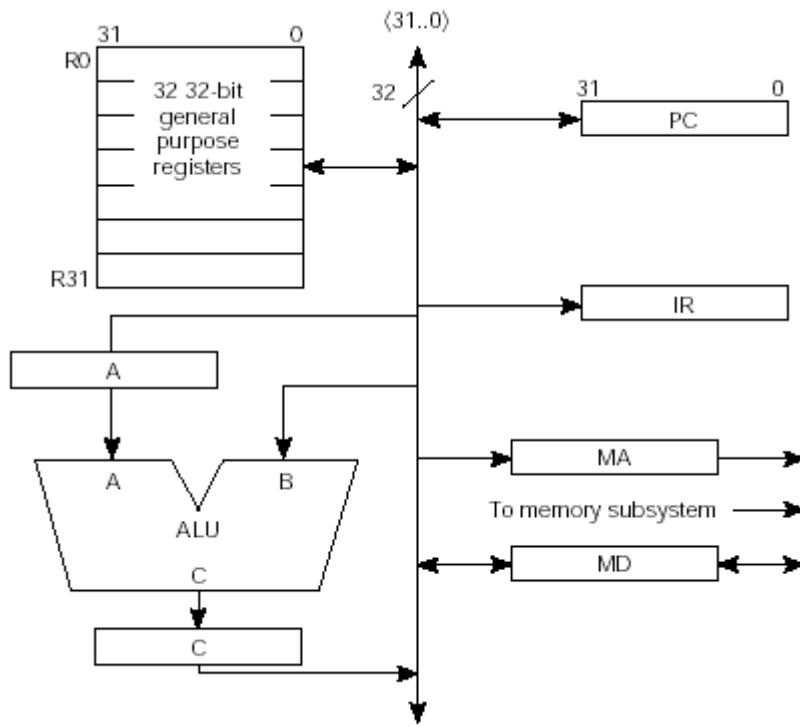
- Execution phases: IF, ID, Ex – Execute, M – Memory, Wb – Write back
- Instructions: R – R-type, LW – Load, SW – Store, BEQ – Branch , J – Jump , ORI – I-type
- ExtOp: 1/0 → 1 – arithmetic, 0 – logical operations



- 1-Bus Multi-Cycle Data-Path



Data-Path – Control Unit – Memory Interfacing



1-BUS SRC (simple RISC Computer)
Block Diagram [2]

ALU connected to the BUS through A and C registers

One bus connecting most registers allows many different RTs, but only one at a time
→ replaces multiplexors

add \$rd, \$rs, \$rt

Abstract RTL:

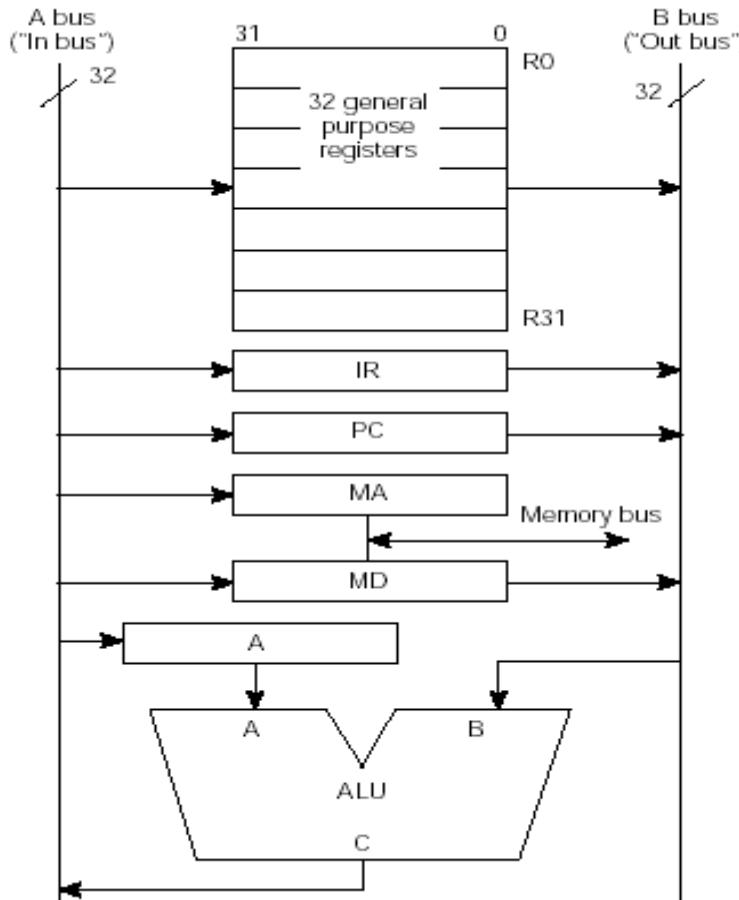
IF: $IR \leftarrow M[PC]$, $PC \leftarrow PC + 4$;
Add: $RF[rd] \leftarrow RF[rs] + RF[rt]$;

Concrete RTL:

IF: $T_0 \rightarrow MA \leftarrow PC$, $C \leftarrow PC + 4$;
 $T_1 \rightarrow MD \leftarrow M[MA]$, $PC \leftarrow C$;
 $T_2 \rightarrow IR \leftarrow MD$;

Ex: $T_3 \rightarrow A \leftarrow RF[rs]$;
 $T_4 \rightarrow C \leftarrow A + RF[rt]$;
 $T_5 \rightarrow RF[rd] \leftarrow C$

- Special ALU operation for $PC + 4$ in T_0
- add takes 3 concrete RTs (T_3, T_4, T_5)



2-BUS SRC – Block Diagram [2]

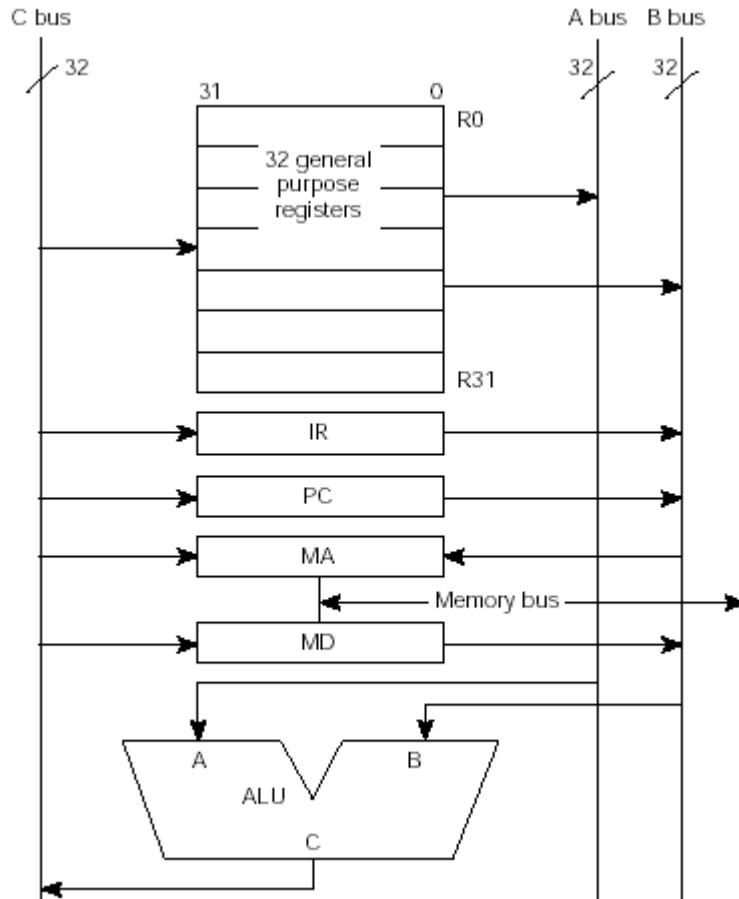
- Bus A carries data going into registers
- Bus B carries data coming from registers
- ALU function $C = B$ (Pass B)
 - is used for all simple register transfers

add \$rd, \$rs, \$rt

Concrete RTL:

IF: $T_0 \rightarrow MA \leftarrow PC;$
 $T_1 \rightarrow PC \leftarrow PC + 4, MD \leftarrow M[MA];$
 $T_2 \rightarrow IR \leftarrow MD;$

Ex: $T_3 \rightarrow A \leftarrow RF[rs];$
 $T_4 \rightarrow RF[rd] \leftarrow A + RF[rt];$



3-BUS SRC – Block Diagram [2]

- A-bus is ALU operand 1
- B-bus is ALU operand 2
- C-bus is ALU output
- Note: MA input connected to the C and B-buses

add \$rd, \$rs, \$rt

Concrete RTL:

IF: $T_0 \rightarrow MA \leftarrow PC, MD \leftarrow M[MA], PC \leftarrow PC + 4$
 $T_1 \rightarrow IR \leftarrow MD;$

Ex: $T_2 \rightarrow RF[rd] \leftarrow RF[rs] + RF[rt];$

- In step T0:
 - PC moves to MA over bus B and goes through the ALU (INC 4 operation)
 - to reach PC again by way of bus C
- PC must be edge-triggered
- MA must be a transparent latch – the address is propagated to the memory unit in T0



Problems – Homework



- Implement other instructions for the Mux based multi-cycle MIPS CPU and for the bus based MIPS CPUs (1, 2 or 3 busses)
 - add, sub, and, or, lw, sw, beq, j, addi, andi, ori
 - sll, srl, sra, sllv, srav, srlv
 - slt, slti
 - bne , bgez, bltz,...
 - jr, jal
 -
- Implement new instructions for the Mux based multi-cycle MIPS CPU and for the bus based MIPS CPUs (1, 2 or 3 busses)
 - LWR, SWR (sums two registers to obtain the memory address)
 - LWA, SWA (uses a single register to obtain the memory address)



References



1. D. A. Patterson, J. L. Hennessy, “Computer Organization and Design: The Hardware/Software Interface”, 5th edition, ed. Morgan–Kaufmann, 2013.
2. D. A. Patterson and J. L. Hennessy, “Computer Organization and Design: A Quantitative Approach”, 5th edition, ed. Morgan-Kaufmann, 2011.
3. MIPS32™ Architecture for Programmers, Volume I: “Introduction to the MIPS32™ Architecture”.
4. MIPS32™ Architecture for Programmers Volume II: “The MIPS32™ Instruction Set”.



Computer Architecture

Lecturer: Mihai Negru

2nd Year, Computer Science

Lecture 7: Multi-Cycle CPU Design (2) Control Unit Design

<http://users.utcluj.ro/~negrum/>



Multi-Cycle Processor Design



- Step-by-step Processor Design → Multi cycle MIPS
 - Step 1: ISA → Abstract RTL
 - Step 2: Components of the Data-Path
 - Step 3: RTL + Components → Data-Path
 - Step 4: Data-Path + Abstract RTL → Concrete RTL
 - **Step 5: Concrete RTL → Control**
- Mux-based multi-cycle data-path designed in the previous lecture



Multi-Cycle CPU Design – Summary



- Five Execution Phases
 - Instruction Fetch
 - Instruction Decode and Register Fetch
 - Execution, Memory Address Computation, or Branch / Jump Completion
 - Memory Access or Arithmetical – Logical instruction completion
 - Write-back
- Instructions take from 3 to 5 clock cycles
- In one clock cycle all operations are done in parallel, not sequential!
 - $T_0 \rightarrow IR \leftarrow M[PC]$ and $PC \leftarrow PC+4$ are done simultaneously
- Between Clock T1 and Clock T2 the control unit will select the next step in accordance to the instruction type



Multi-Cycle CPU Design – Control Signals



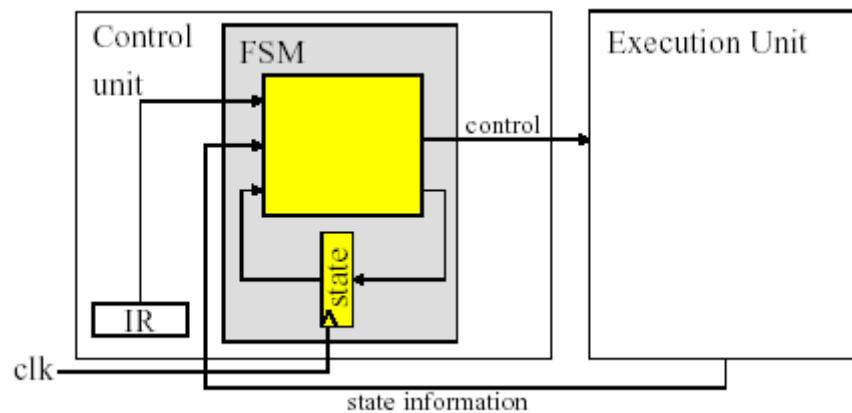
| T | IorD | Mem Read | Mem Write | IR Write | Reg Dst | Mem toReg | Reg Write | Ext Op | ALU SrcA | ALU SrcB | ALU Op | PC Src | PC WrCd | PC Wr | |
|----|------|----------|-----------|----------|---------|-----------|-----------|--------|----------|----------|--------|--------|---------|-------|--------|
| T0 | 0 | 1 | 0 | 1 | x | x | 0 | x | 0 | 1 | add | 0 | 0 | 1 | IF |
| T1 | x | 0 | 0 | 0 | x | x | 0 | 1 | 1 | 3 | add | x | 0 | 0 | ID |
| T2 | x | 0 | 0 | 0 | x | x | 0 | x | 1 | 0 | fun | x | 0 | 0 | Ex R-T |
| T3 | x | 0 | 0 | 0 | 1 | 0 | 1 | x | x | x | x | x | 0 | 0 | Wb R-T |
| T2 | x | 0 | 0 | 0 | x | x | 0 | 0 | 1 | 2 | or | x | 0 | 0 | Ex ORI |
| T3 | x | 0 | 0 | 0 | 0 | 0 | 1 | x | x | x | x | x | 0 | 0 | Wb ORI |
| T2 | x | 0 | 0 | 0 | x | x | 0 | 1 | 1 | 2 | add | x | 0 | 0 | Ex LW |
| T3 | 1 | 1 | 0 | 0 | x | x | 0 | x | x | x | x | x | 0 | 0 | M LW |
| T4 | x | 0 | 0 | 0 | 0 | 1 | 1 | x | x | x | x | x | 0 | 0 | Wb LW |
| T2 | x | 0 | 0 | 0 | x | x | 0 | 1 | 1 | 2 | add | x | 0 | 0 | Ex SW |
| T3 | 1 | 0 | 1 | 0 | x | x | 0 | x | x | x | x | x | 0 | 0 | M SW |
| T2 | x | 0 | 0 | 0 | x | x | 0 | x | x | x | sub | 1 | 1 | 0 | Ex BEQ |
| T2 | x | 0 | 0 | 0 | x | x | 0 | x | x | x | x | 2 | 0 | 1 | Ex J |

Table 1: The Values of the Control Signals in each Clock Cycle

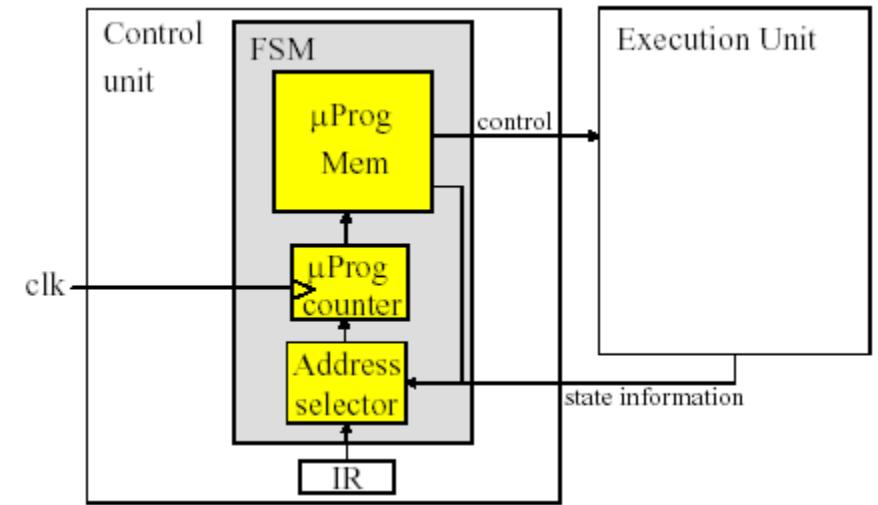
- Execution phases: IF, ID, Ex – Execute, M – Memory, Wb – Write back
- Instructions: R – R-type, LW – Load, SW – Store, BEQ – Branch , J – Jump , ORI – I-type
- ExtOp: 1/0 → 1 – arithmetic, 0 – logical operations



- Control Unit implementation
 - Hardwired
 - Micro-Programmed
- Hardwired vs. Micro-Programmed
 - Hardwired units are faster
 - Hardwired design is complex if large
 - Bugs in hardwired design cannot be fixed in field
 - Emulation is easy with micro-coding



Hardwired Control Unit



Micro-Programmed Control Unit



Multi-Cycle CPU Design – Step 5

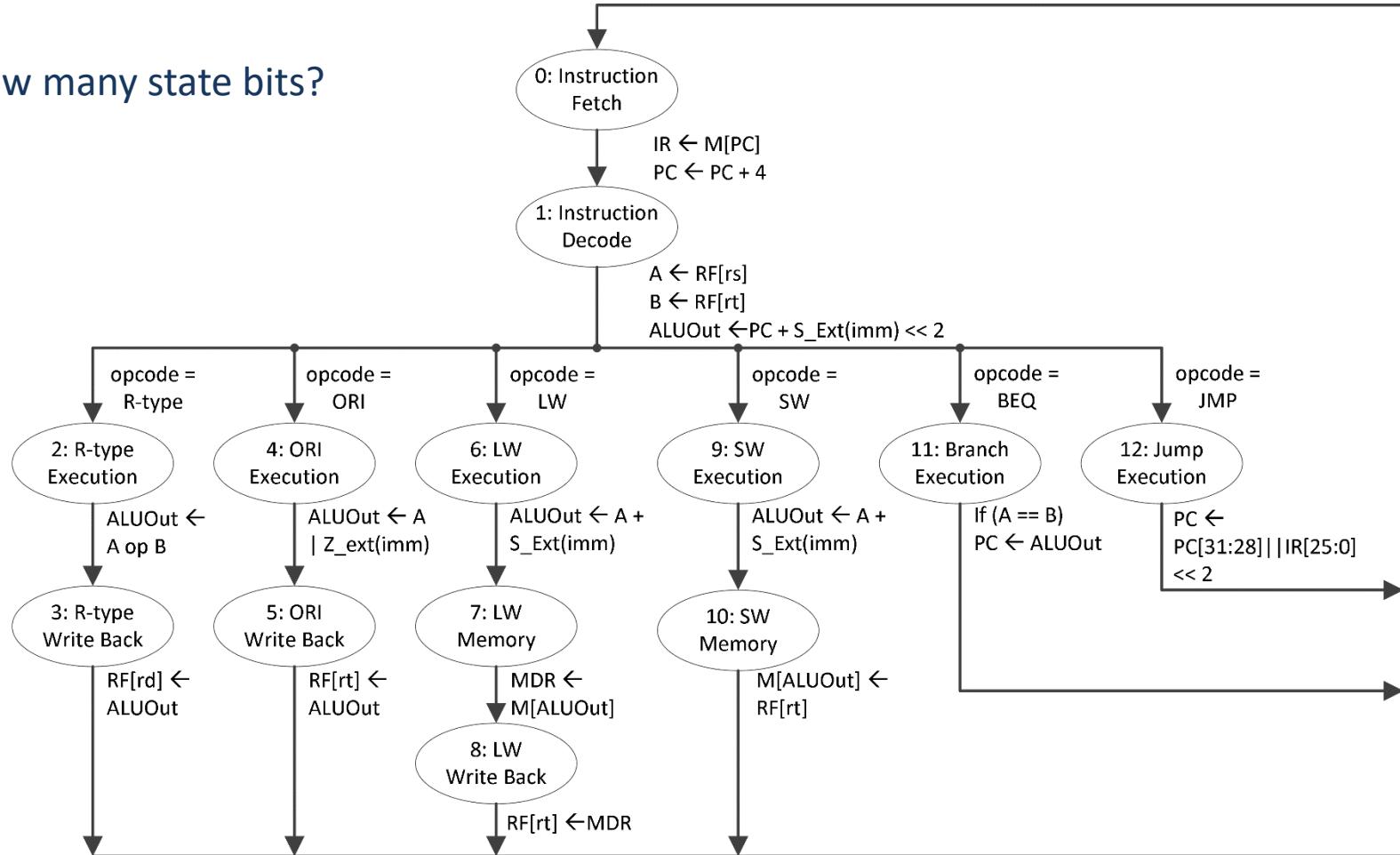


- Control Unit implementation can be derived from specification
 - Hardwired
 - Micro-Programming
- Hardwired control unit
 - We can specify the Control Unit as a finite state machine
 - We'll use a Moore machine (output based only on current state)
 - Value of control signals is dependent upon:
 - What instruction is being executed
 - Which step is being performed
 - Each entry in the **Control Signals Values Table** is a state in our FSM



- Finite State Machine for MIPS-lite Multi-Cycle CPU: FSM 1
 - FSM1 is based on the Control Signals Values table (table 1)

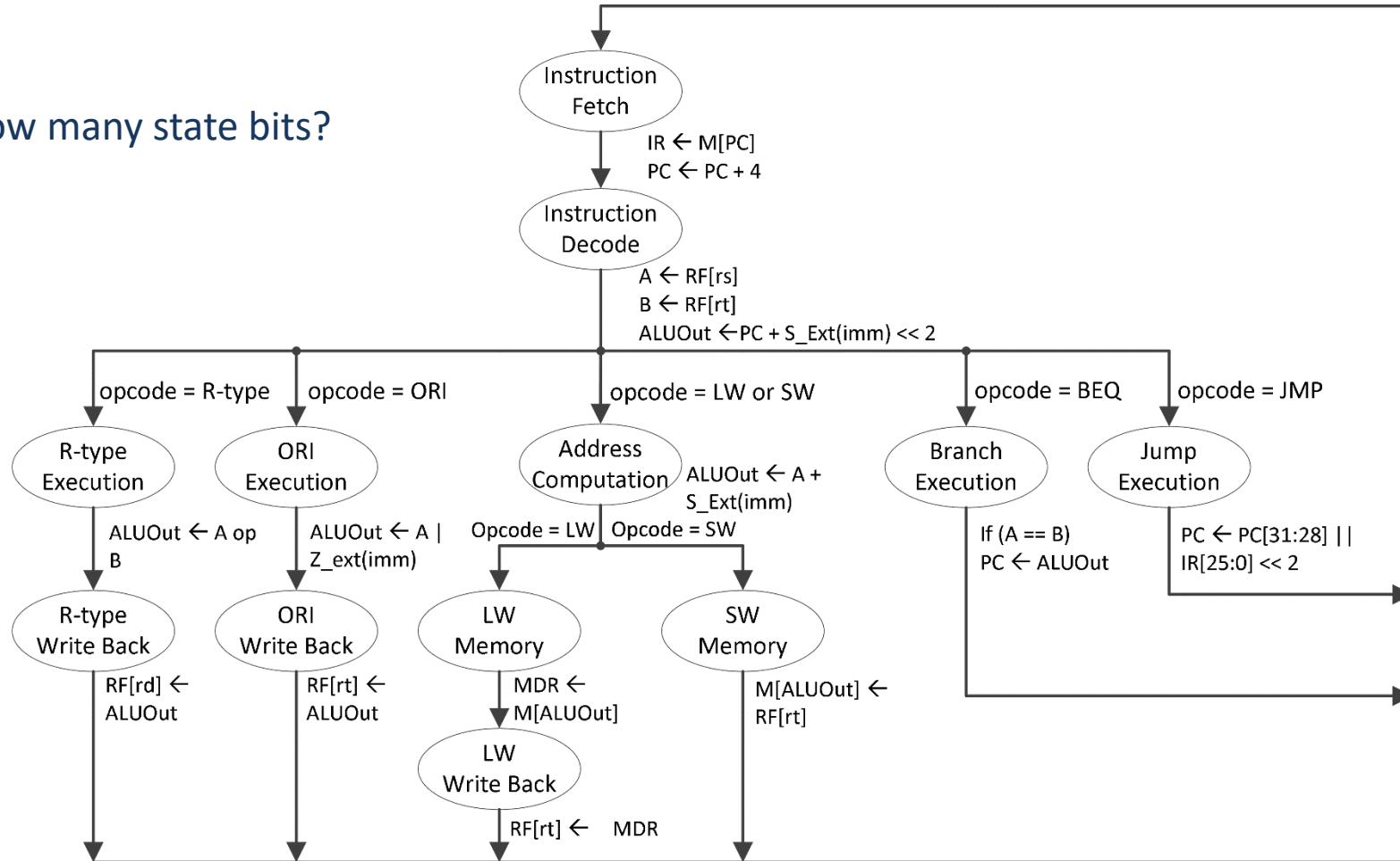
How many state bits?





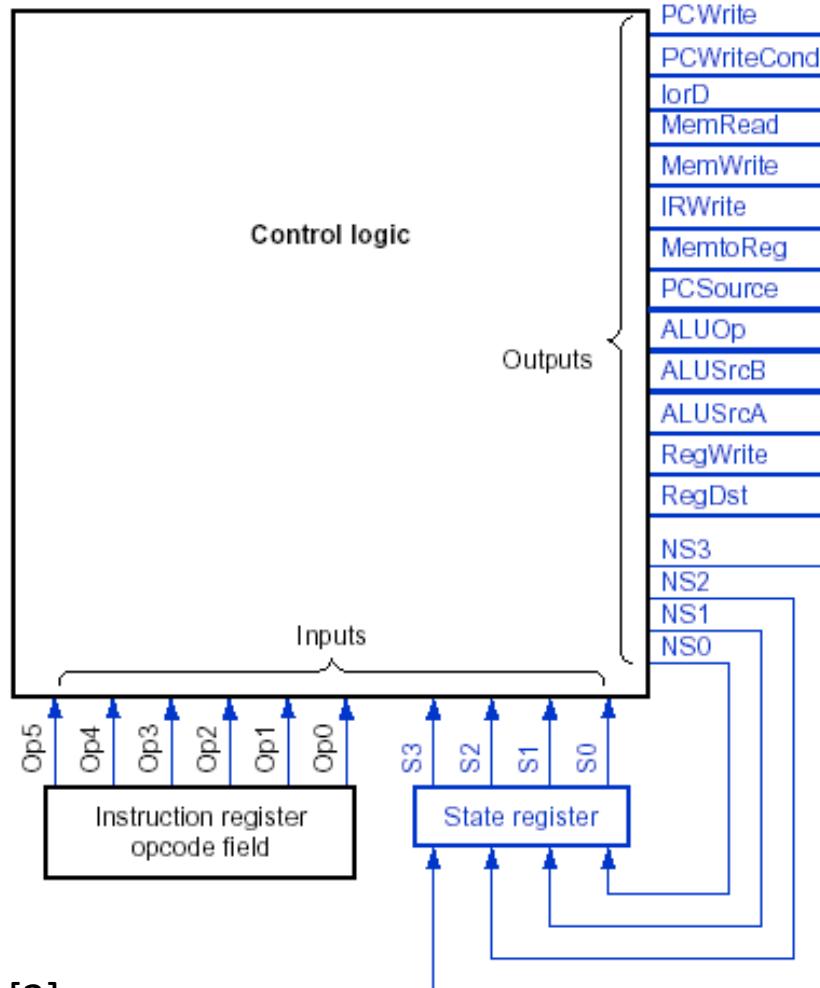
- Optimized FSM1 – unify Execution for LW and SW: FSM 2

How many state bits?





- Hardwire implementation for MIPS Multi-Cycle Control Unit

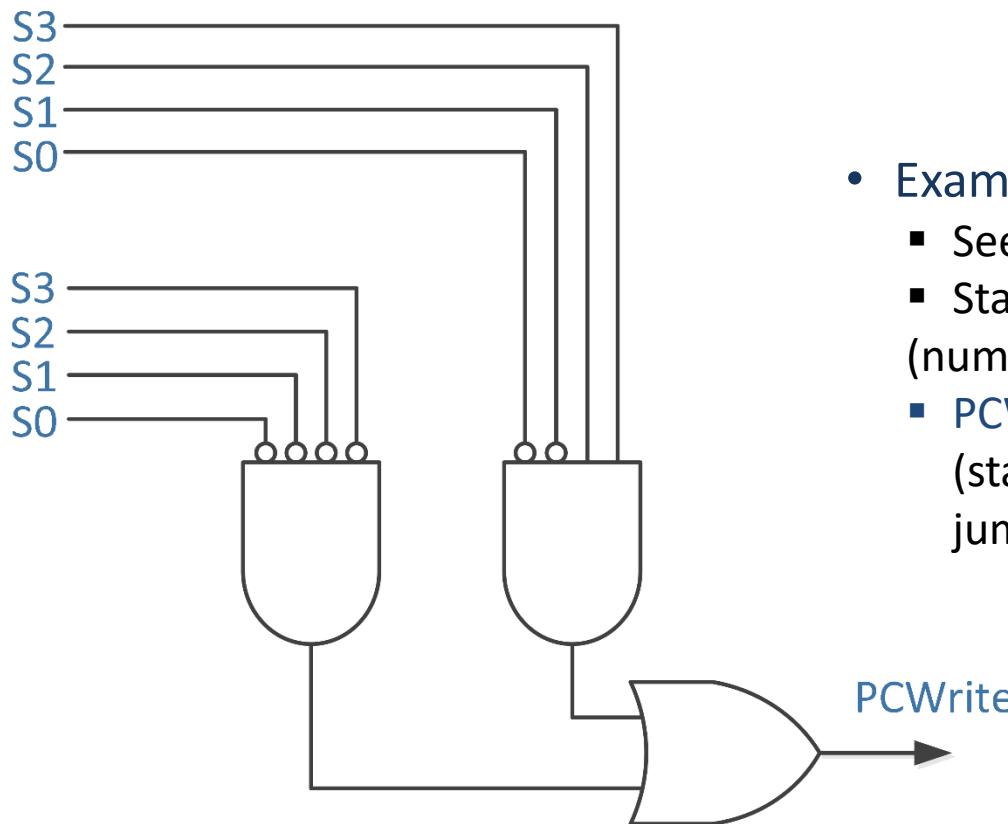


[2]

- Control Unit
 - State register to hold the state
 - Control logic
 - assigns the values for the control signals in each state
 - Current state + opcode field decode the next state
- Possible implementations:
 - Programmable Logic Array (PLA) to implement the control logic
 - One Flip-Flop per state to implement the FSM
 - Sequence/Jump Counter + Decoder to implement the FSM
 - ...



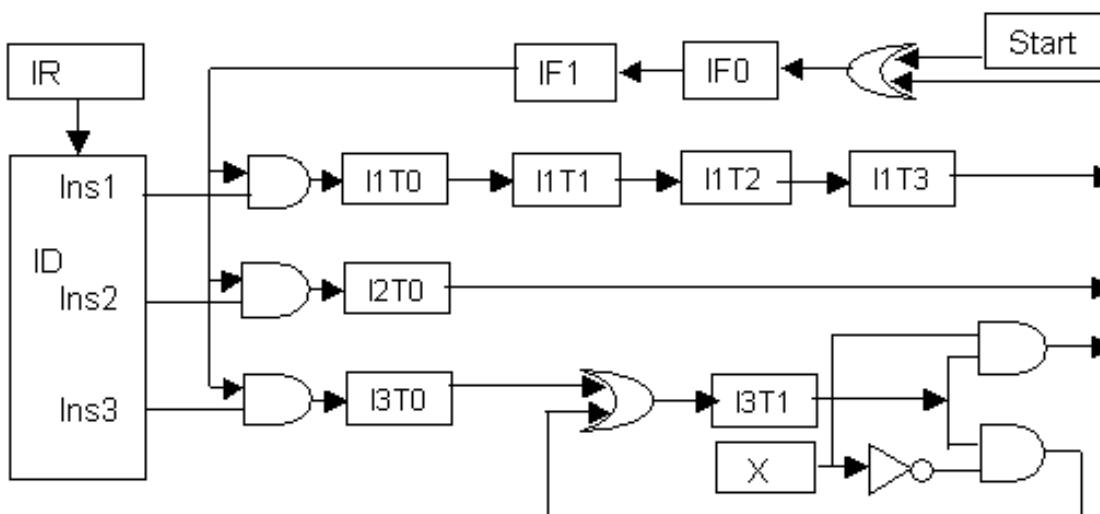
- PLA (programmable logic array) based control logic
 - A net of **AND** gates; the outputs are connected by **OR** gates
 - Every control signal is set by the current state
 - **Next state** derived from **current state** and the **opcode** field



- Example – PCWrite :
 - See table 1, FSM1.
 - States encoded on 4 bits (numbering top → down, left → right)
 - PCWrite must be set in Instruction Fetch (state 0 = “0000”) and when executing a jump (state 12 = “1100”)



- State encoding Types
 - Compact binary encoded FSM
 - The state numbers depend on instruction decode and execution phases
 - → Hard to extend or modify
 - One flip-flop per state (One-Hot encoding)

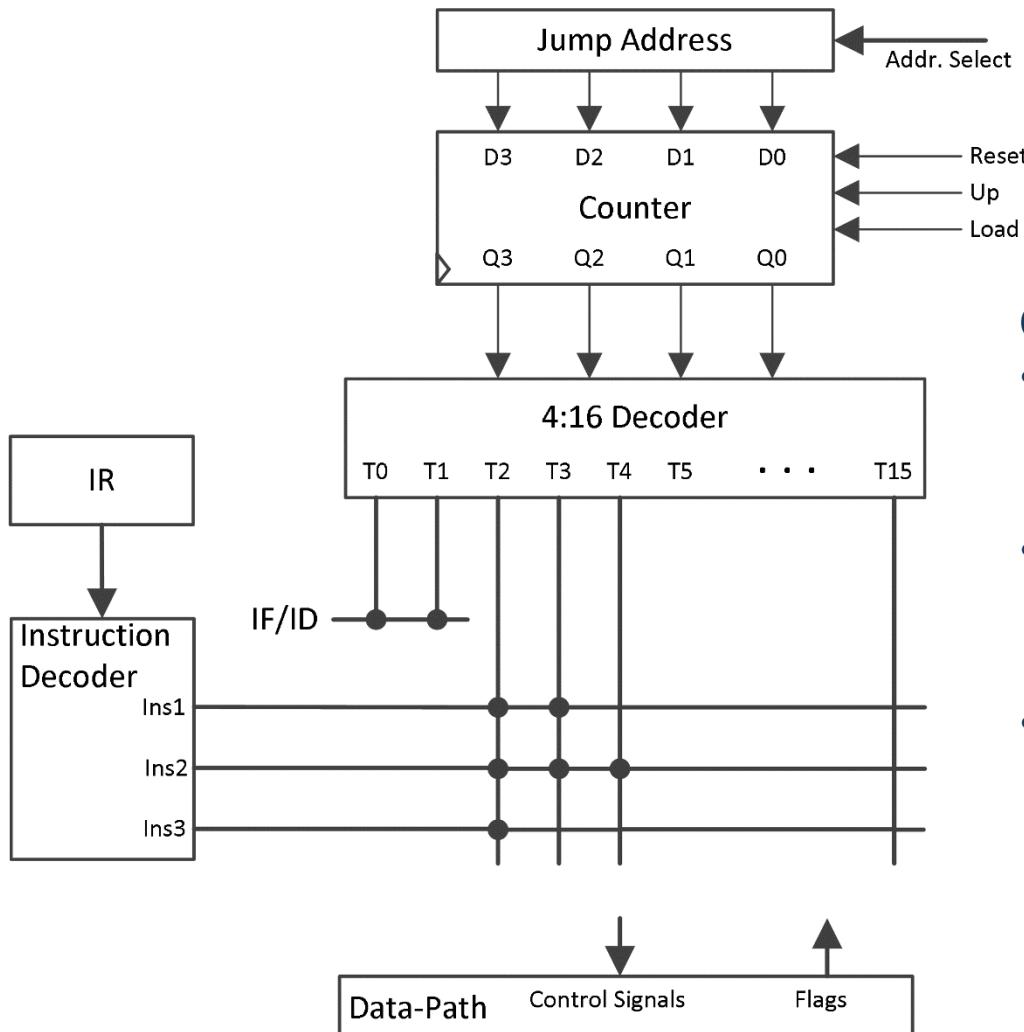


One Flip-Flop per State Control Unit

- Start: Synchronized, a single clock period duration signal
- X: Condition Flag
- IF0, IF1: Instruction Fetch and decode execution phases
- IjTi: Instruction execution phases
- The instruction decode and execution phases are independent
- → Easy to extend or modify



Hardwired Control Unit – Sequence Counter



- IF, ID: T0 and T1
- After IF, InsX & T2 determines the start of a new instruction execution
- After completion → Reset → T0 and IF for the new instruction

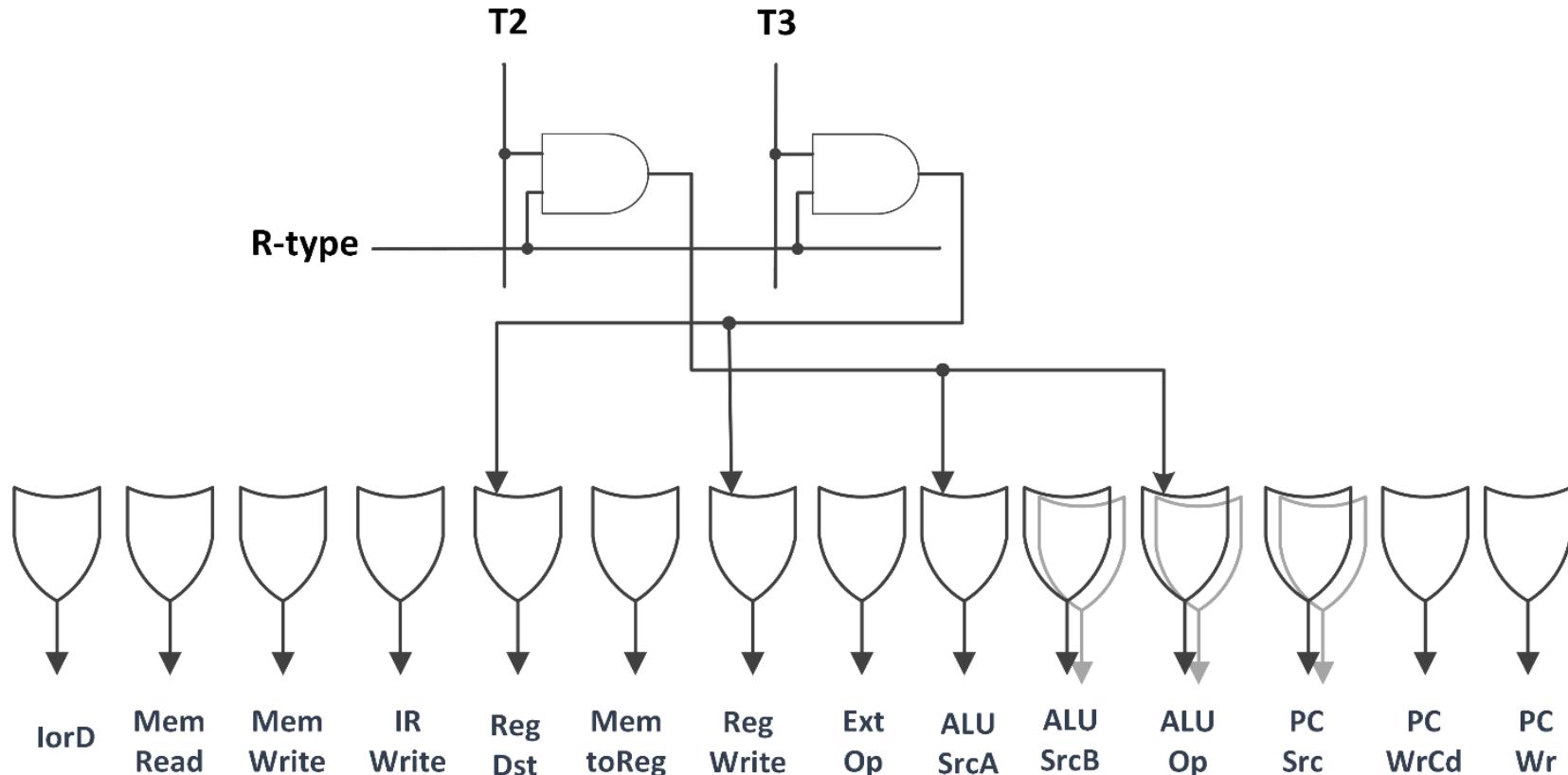
Counter Operation

- **Reset:** Counter Reset
 - selects the T0 column through the Decoder
- **Up:** Count Up validation
 - Implicitly asserted
 - Load and Reset have priority
- **Load:** Counter Load for micro-jumps inside the counter
 - The micro-jump addresses are selected by the currently active control signals from the control matrix

Reduction of states possible – depends on implementation



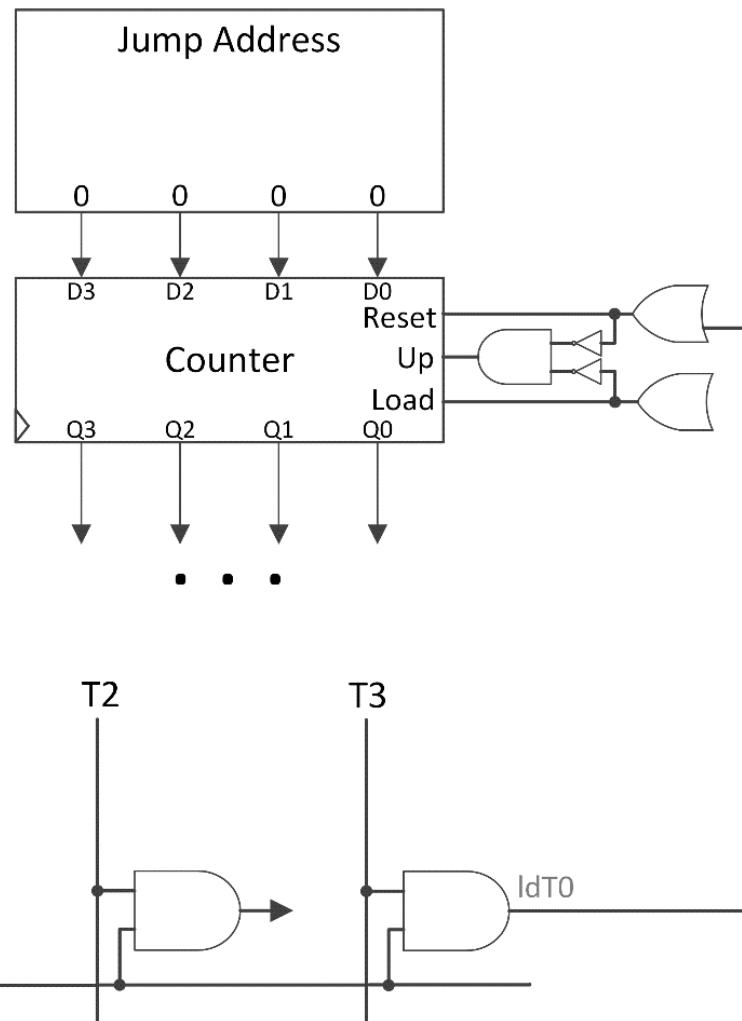
- Sequence/Jump Counter Control Unit
 - Example for R-type instructions
 - Sequencing: T2 – implicit Up, T3 – Reset (jump to IF – T0)



Hardwired Control Unit Decode Logic for R-type instruction



- Sequencing logic example: R-type Instructions



- T2 – implicit Up
- T3 – Reset (jump to IF – T0)

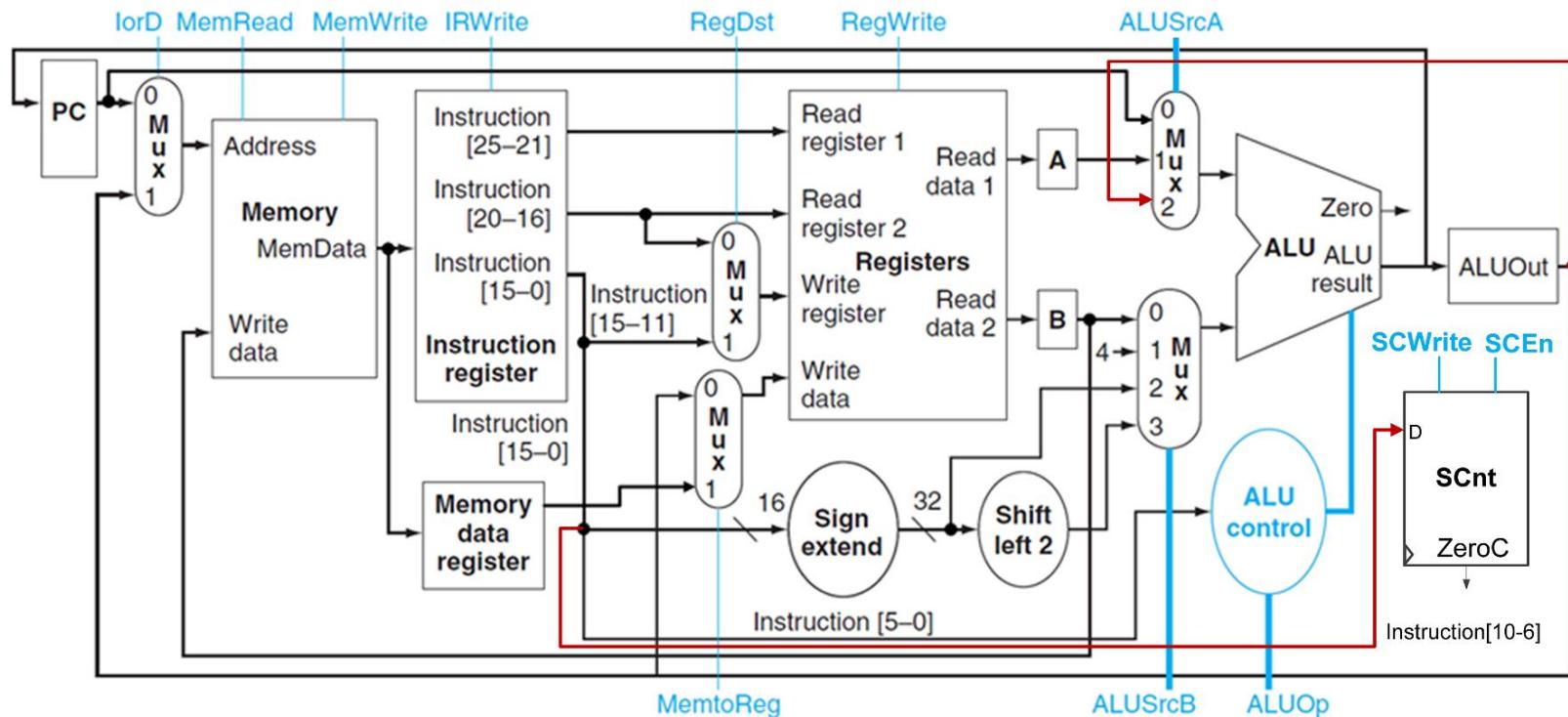


Hardwired Control Unit – Sequence Counter



- Example – extend for instructions that take a variable number of clock cycles
- **R-type: sll \$rd, \$rs, sa** RTL Abstract: $RF[rd] \leftarrow RF[rs] \ll sa$
 - Suppose ALU performs only 1-bit shifts
 - We must extend the data-path in order to allow
 - A mechanism of counting the clock cycles: dedicated counter **SCnt** with parallel load (sa is loaded when **SCWrite** is asserted), count down enable (**SCen**), 1-bit zero detector signal (**ZeroC**)
 - A connection from **ALUOut** to one of the **ALU inputs**, in order to repeat the shifting process (as long as it is necessary)
 - The control unit will treat this R-type instruction separately (dedicated line from Instruction Decoder, to the sequence counter)

Hardwired Control Unit – Sequence Counter

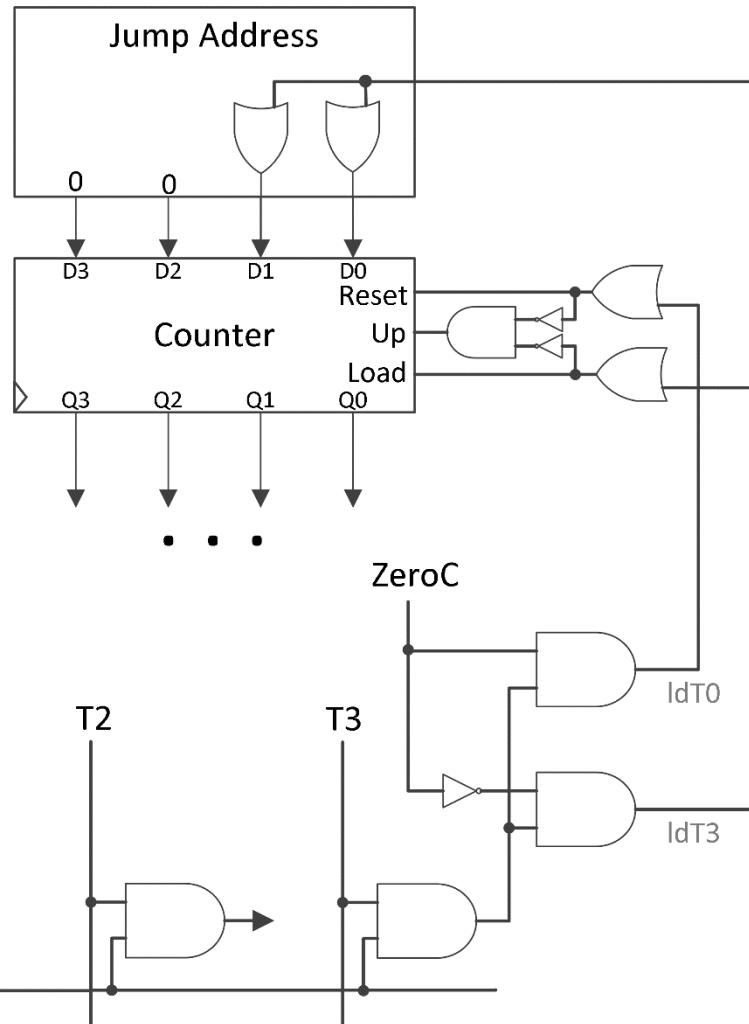


| | |
|---------------------------------------|--|
| $T_0 \rightarrow$ | $IR \leftarrow M[PC]$, $PC \leftarrow PC + 4$; |
| $T_1 \rightarrow$ | $A \leftarrow RF[rs]$, $B \leftarrow RF[rt]$; $SCnt \leftarrow sa$ |
| $SLL \& T_2 \& ZeroC = 0 \rightarrow$ | $ALUOut \leftarrow A \ll 1$; $SCnt \leftarrow SCnt - 1$ |
| $SLL \& T_2 \& ZeroC = 1 \rightarrow$ | $ALUOut \leftarrow A$; |
| $SLL \& T_3 \& ZeroC = 0 \rightarrow$ | $ALUOut \leftarrow ALUOut \ll 1$; $SCnt \leftarrow SCnt - 1$; repeat (load T3) |
| $SLL \& T_3 \& ZeroC = 1 \rightarrow$ | $R[rd] \leftarrow ALUOut$; load T0 |

What are the values of the control signals in each state?



Hardwired Control Unit – Sequence Counter



T2: Implicit Up

T3: ($ZSC = 0$) \rightarrow IdT3; Repeat the same state
($ZSC = 1$) \rightarrow IdT0; Reset (go to IF – T0)

Shift Left Logical: SLL – sequencing



Multi-Cycle CPU Design – Step 5

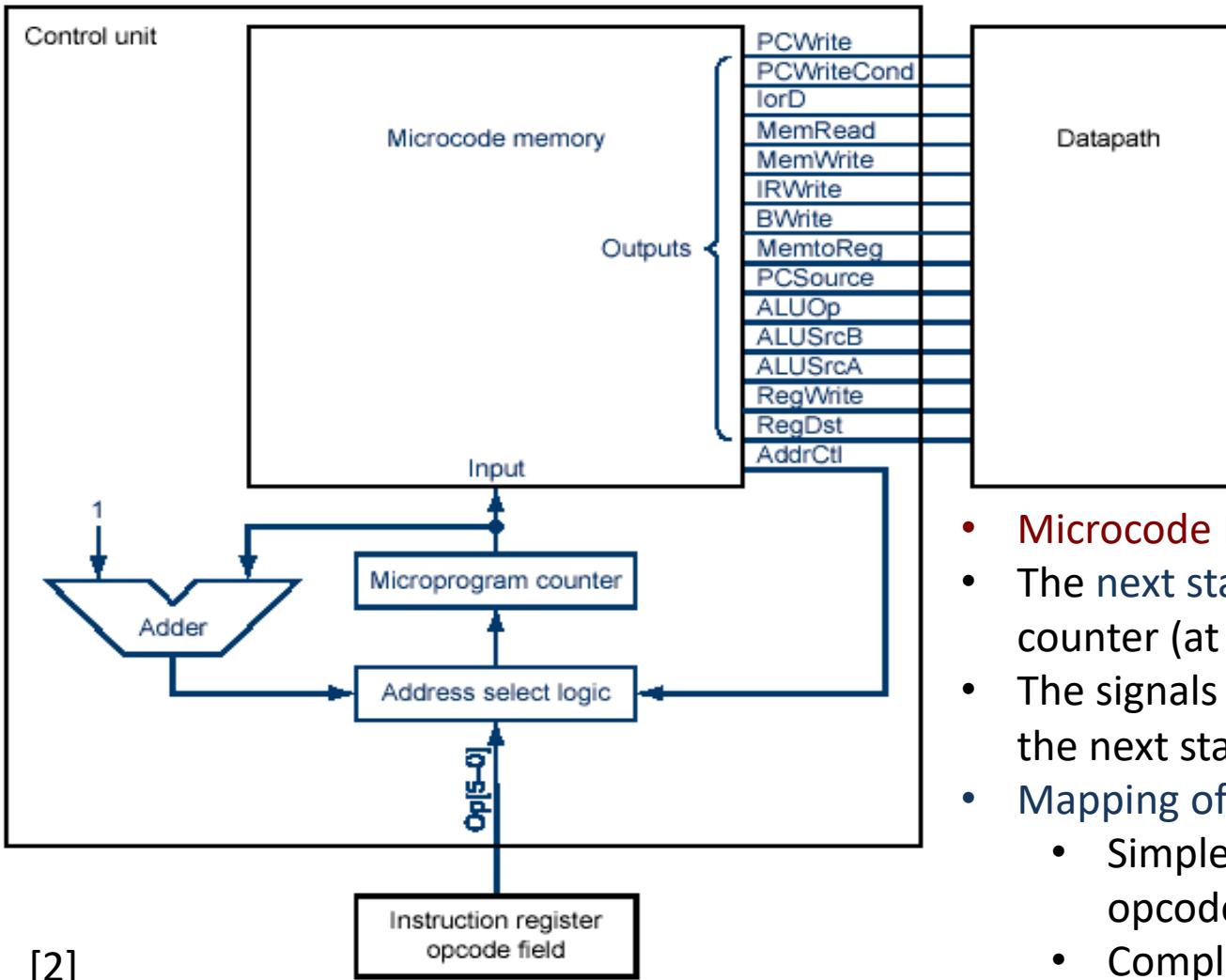


- Disadvantages of Using Hardwire Control Units
 - The complexity of the FSM depends on Data-Path complexity
 - Real processors are complex: over 100 instructions with 1 to 20 cycles.
 - → Use micro-programming
- Micro-Programmed Control Unit
 - Appropriate if hundreds of opcodes, modes, cycles, etc.
 - Control signals are specified symbolically using **μinstructions**
 - Possible **Sequencing Capabilities** in a Micro-Programmed Control Unit
 - Incrementing of the control address register
 - Unconditional and conditional branches
 - Mapping from the machine instruction's operation code to the address of the corresponding μinstruction sequence in control memory
 - A facility for subroutine call and return

Micro-Programmed Control Unit



- Micro-Programmed Control Unit Block Diagram



- **Microcode Memory = Control Memory**
- The **next state** is computed using a counter (at least in some states)
- The signals labeled **AddrCtl** control how the next state is determined → MUX
- **Mapping of instructions to μ operations**
 - Simple – concatenation between opcode and address
 - Complex – ROM or PLA



Micro-Programmed Control Unit



- Variations on Micro-Programming
 - “Horizontal” micro-code → Example on next slides
 - Each data-path control signal is represented by a bit in the μ instruction
 - Horizontal μ code has wider μ instructions
 - Multiple parallel operations per μ instruction
 - Fewer steps per instruction
 - Simple encoding → more bits
 - “Vertical” micro-code
 - Encode fields to save ROM space
 - Vertical μ code has narrower μ instructions
 - Typically a single data-path operation per μ instruction
 - Separate μ instructions for branches
 - More steps per instruction
 - More compact → less bits
 - Nano-coding
 - Two level microprogramming
 - Tries to combine best of horizontal and vertical μ -code



Micro-Programmed Control Unit



- Micro-Programming Pros and Cons
 - Ease of design
 - Flexibility
 - Easy to adapt to changes in organization, timing, technology
 - **Can make changes late in design cycle, or even in the field**
 - Can implement very powerful instruction sets (just more control memory)
 - Generality
 - Can implement multiple instruction sets on same machine
 - Can tailor instruction set to application requirements
 - Compatibility
 - Many organizations, same instruction set
 - Slow (ROM is no longer faster than RAM)



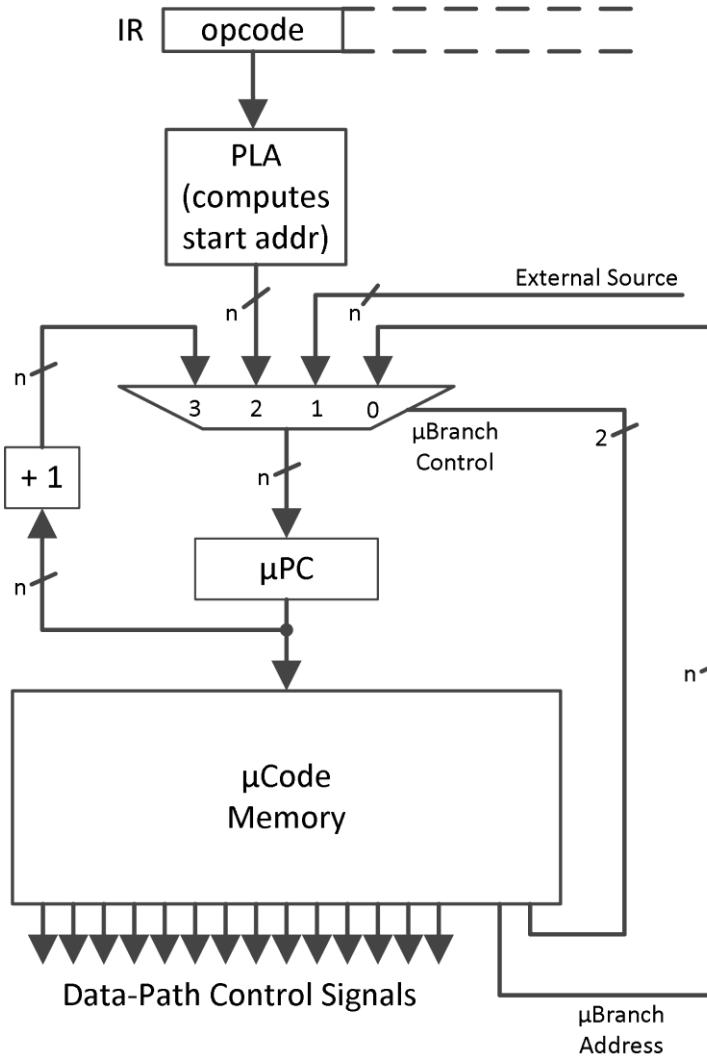
Micro-Programmed Control Unit



- Designing a μ instruction set
 1. Start with list of control signals
 2. Group signals together that make sense (vs. random): called **fields**
 3. Places fields in some logical order (e.g., ALU operation & ALU operands first and μ instruction sequencing last)
 4. Create a symbolic legend for the μ instruction format, showing name of field values and how they set the control signals
 5. To minimize the width, encode in the same field operations that will not be used at the same time (vertical micro-programming)



- Micro-Programmed Control Unit Example for FSM1



μ Instruction fields

| Data-Path Control Signals | μ Branch Address | μ Branch Control |
|---------------------------|----------------------|----------------------|
|---------------------------|----------------------|----------------------|

- The data-path control signals are defined in table 1
- The μ Branch address: 4 bits (13 states)
- The μ Branch control: select signal for the multiplexer – 2 bits
- μ Branch control
 - 00 → μ Branch address
 - 01 → External Source (not used in this example)
 - 10 → The first μ instruction address for the mapped operation code
 - 11 → Next sequential μ instruction address



Micro-Programmed Control Unit – V1

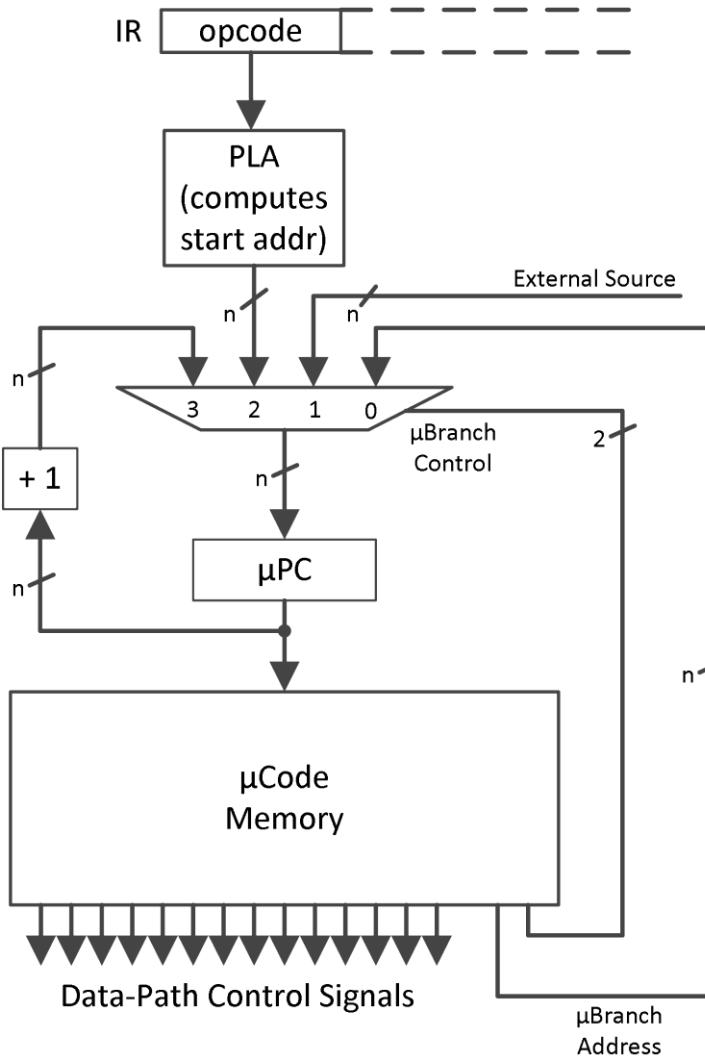


| Addr. | IorD | MemRead | MemWrite | IRWrite | RegDst | Mem toReg | Reg Write | ExtOp | ALUSrcA | ALUSrcB | ALUOp | PCSrc | PcWrCd | PCWr | μ Branch Address | μ Branch Control | Execution Phase |
|-------|------|---------|----------|---------|--------|-----------|-----------|-------|---------|---------|-------|-------|--------|------|----------------------|----------------------|-----------------|
| No. | 1b | 1b | 1b | 1b | 1b | 1b | 1b | 1b | 1b | 2b | 2b | 2b | 1b | 1b | 4b | 2b | |
| 00 | 0 | 1 | 0 | 1 | x | x | 0 | x | 0 | 1 | add | 0 | 0 | 1 | x | 3 | IF |
| 01 | x | 0 | 0 | 0 | x | x | 0 | 1 | 1 | 3 | add | x | 0 | 0 | x | 2 | ID |
| 02 | x | 0 | 0 | 0 | x | x | 0 | x | 1 | 0 | fun | x | 0 | 0 | x | 3 | Ex R-T |
| 03 | x | 0 | 0 | 0 | 1 | 0 | 1 | x | x | x | x | x | 0 | 0 | 0000 | 0 | Wb R-T |
| 04 | x | 0 | 0 | 0 | x | x | 0 | 0 | 1 | 2 | or | x | 0 | 0 | x | 3 | Ex ORI |
| 05 | x | 0 | 0 | 0 | 0 | 0 | 1 | x | x | x | x | x | 0 | 0 | 0000 | 0 | Wb ORI |
| 06 | x | 0 | 0 | 0 | x | x | 0 | 1 | 1 | 2 | add | x | 0 | 0 | x | 3 | Ex LW |
| 07 | 1 | 1 | 0 | 0 | x | x | 0 | x | x | x | x | x | 0 | 0 | x | 3 | M LW |
| 08 | x | 0 | 0 | 0 | 0 | 1 | 1 | x | x | x | x | x | 0 | 0 | 0000 | 0 | Wb LW |
| 09 | x | 0 | 0 | 0 | x | x | 0 | 1 | 1 | 2 | add | x | 0 | 0 | x | 3 | Ex SW |
| 10 | 1 | 0 | 1 | 0 | x | x | 0 | x | x | x | x | x | 0 | 0 | 0000 | 0 | M SW |
| 11 | x | 0 | 0 | 0 | x | x | 0 | x | x | x | sub | 1 | 1 | 0 | 0000 | 0 | Ex Br |
| 12 | x | 0 | 0 | 0 | x | x | 0 | x | x | x | x | 2 | 0 | 1 | 0000 | 0 | Ex J |

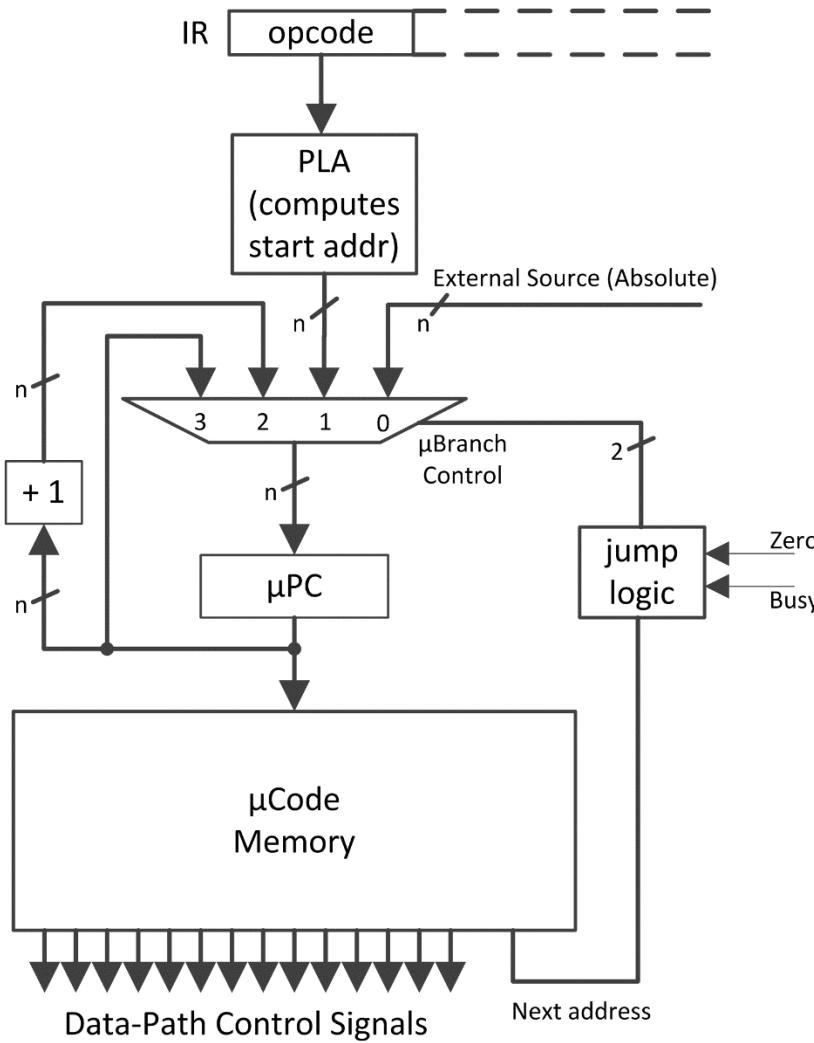
The μ Code Memory for the selected instructions

- Horizontal micro instruction length: **23 bits**
- The addresses correspond to the order in table 1
- For the selected instructions the μ Branch field is used only for IF μ Address generation
- If we connect to the External Source the IF μ Address \rightarrow shorter μ Instruction length

- Micro-Programmed Control Unit Example for FSM1



- **μ Branch control**
 - 00 → μ Branch address
 - 01 → External Source (not used in this example)
 - 10 → The first μ instruction address for the mapped operation code
 - 11 → Next sequential μ instruction address
- **Disadvantage:** the select logic does not depend on state bits => one cannot implement more complex instructions (SLL from our previous example)
- **Possible variation** – different inputs to the sequencing Multiplexer and conditional logic for the select signal of the multiplexer



μ -Programmed Control Unit, V2

Differences from V1:

- μ PC and μ PC+1 inputs
- The μ Branch address and μ Branch control replaced by Next address
- External – absolute address of IF from μ Code Memory
- Sequencing based on Jump Logic – considering the data-path status signals and Next address

μ Control values

| | |
|----------|--|
| next | $\rightarrow \mu$ PC+1 |
| spin | \rightarrow if (busy) then μ PC else μ PC+1 |
| fetch | \rightarrow absolute |
| dispatch | \rightarrow PLA = MAP(opcode) or a decoder |
| feqz | \rightarrow if (zero) then absolute else μ PC+1 (fetch if equal zero) |
| fnez | \rightarrow if (zero) then μ PC+1 else absolute (fetch if not equal zero) |



Micro-Programmed Control Unit



- Micro-Programmed Control Unit Possible Types
 - V1: The Sequencer's MUX inputs: $\mu\text{PC}+1$, MAP (Opcode), External Source, $\mu\text{Branch Address}$. Unconditional MUX Selection Logic from the $\mu\text{instructions}$.
 - V2: The Sequencer's MUX inputs: MAP (opcode), External Source (absolute), $\mu\text{PC}+1$, μPC . Conditional MUX Selection Logic: Condition selection code from the $\mu\text{instructions}$ and external condition flag evaluation logic.
 - V3: The Sequencer's MUX inputs: $\mu\text{PC}+1$, MAP1 (opcode), MAP2 (Opcode), Instruction Fetch $\mu\text{Address}$. Unconditional MUX Selection Logic from the $\mu\text{instructions}$ (Vertical Micro-Programming).
 - V4: The Sequencer's MUX inputs: $\mu\text{PC}+1$, MAP(Opcode), External Source, $\mu\text{Branch Address}$. Conditional MUX Selection Logic: Condition selection code from the $\mu\text{instructions}$ and external condition flag evaluation logic.
 -



Multi-Cycle CPU Design – Exceptions



- Definition: Event causes unexpected transfer of control
- Types:
 - Exceptions[overflow] → generated inside the processor
 - Interrupts [I/O] → associated with external events
 - MIPS Exceptions: undefined instruction or arithmetic overflows
 - Exception Detection – How to discover exception?
 - Exception Handling – What to do?
- Exception Detection:
 - Undefined Instruction
 - Add an **IllegalOp State** to the FSM
 - Every unknown instruction (undefined opcode) → transitions to the **Exception State**
 - Arithmetic Overflow
 - ALU has overflow detection logic → create a new **Overflow State** to handle overflow



Multi-Cycle CPU Design – Exceptions



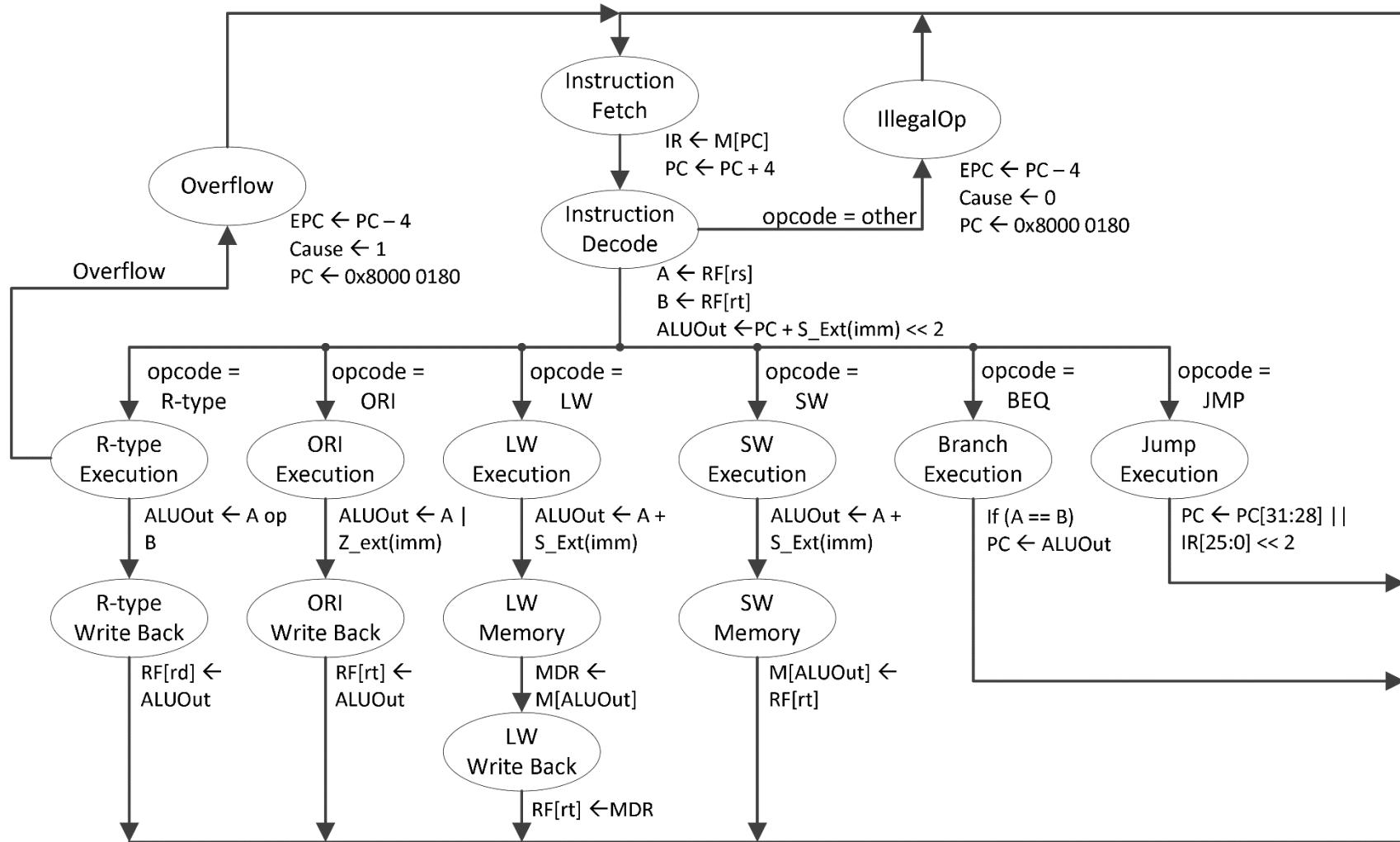
- Exception handling
 - Two Techniques: EPC / Cause Registers and Vectored Interrupts
 - Vectored Interrupts
 - Each exception has a distinct address A_E associated with it
 - Exception detected $\rightarrow A_E$ for that exception written to PC
 - EPC / Cause: MIPS
 - Exception Program Counter Register – EPC (32 bits): stores the address of the offending instruction: $EPC \leftarrow (PC + 4 - 4) = PC$ (use ALU for subtract 4)
 - Cause Register – 32 bits: stores the cause of the exception:
 - undefined instruction: $Cause \leftarrow 0$
 - arithmetic overflows: $Cause \leftarrow 1$
 - Exception detected
 - \rightarrow Address of instruction saved in EPC
 - \rightarrow Cause register stores an exception type code
 - Exception handler acts on Cause, tries to restart execution at instruction pointed to by EPC



Multi-Cycle CPU Design – Exceptions



- FSM1 with Exceptions handling states



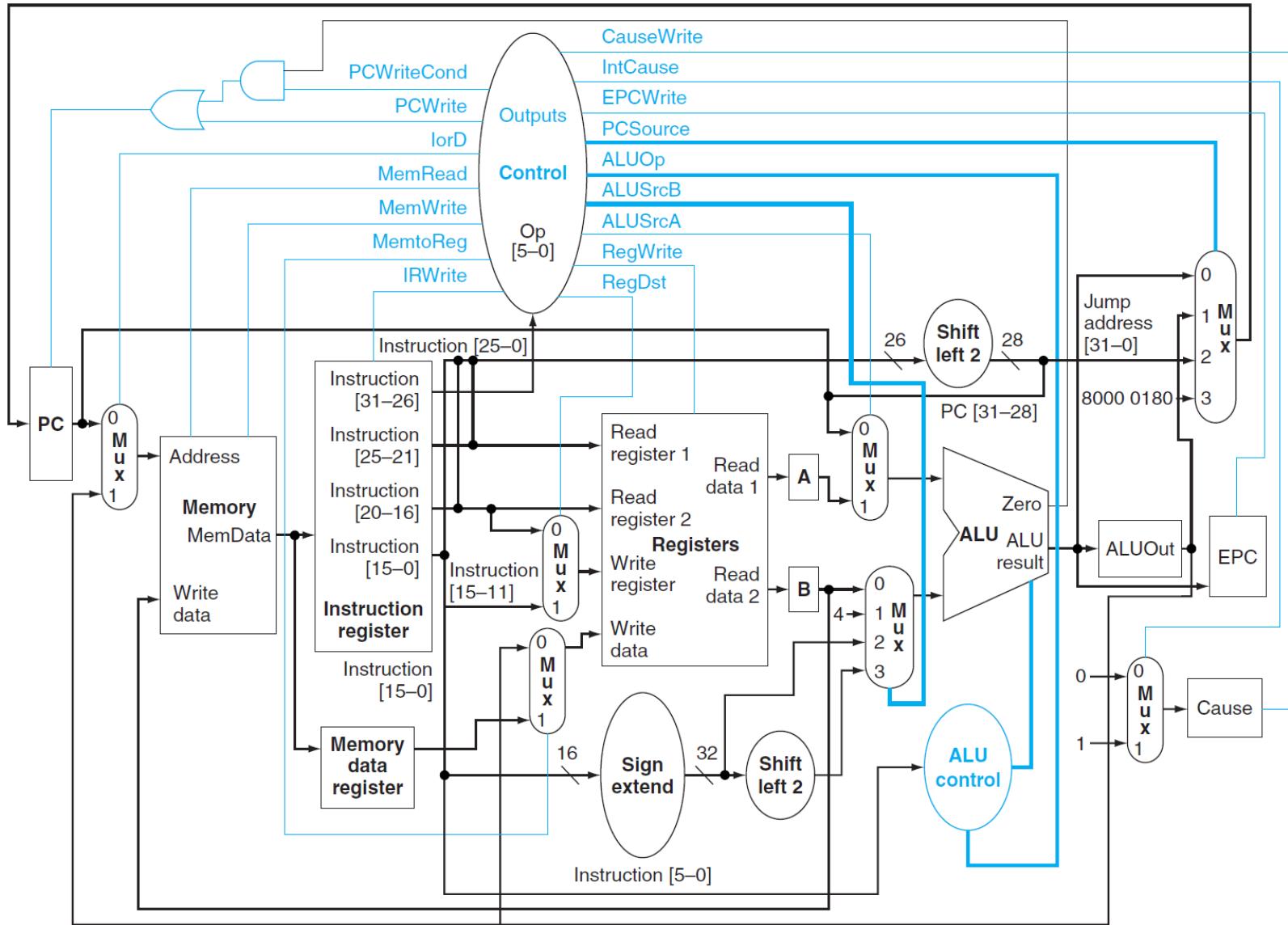


Multi-Cycle CPU Design – Exceptions



- Modifications to MIPS Multi-Cycle Data-Path for Exceptions
 - New Registers – EPC and Cause (32-bit)
 - New Control Signals – EpcWrite, CauseWrite
 - New Control Line: IntCause
 - 0 for undefined instruction
 - 1 for overflow
 - New Mux input for PCsource Signal (3)
 - PC inputs
 - PC + 4
 - Branch Target Address
 - Jump Address
 - Additional input: $A_E = 8000\ 0180_{16}$ in MIPS
- Recall: ALU overflow detection already “installed”

Multi-Cycle CPU Design – Exceptions



[1]



Problems – Homework



- 1-Bus, 2-Bus and 3-Bus based MIPS implementation of the instructions
 - add, sub, and, or, lw, sw, beq, j, addi, andi, ori
 - sll, srl, sra, sllv, srav, srlv
 - slt, slti
 - bne , bgez, bltz,...
 - jr, jal
 -
- Design the control unit in order to reflect the new instructions
 - Hardwired control unit
 - Micro-programmed control unit



References



1. D. A. Patterson, J. L. Hennessy, “Computer Organization and Design: The Hardware/Software Interface”, 3rd edition, ed. Morgan–Kaufmann, 2005.
2. Vincent P. Heuring, Harry F. Jordan, “Computer Systems Design and Architecture”, 2nd Edition
3. CODE3e – Appendix D: Mapping Control to Hardware
4. Microprogramming – *Arvind* Computer Science & Artificial Intelligence Lab M.I.T., Based on the material prepared by Arvind and Krste Asanovic, September 21, 2005
5. MIPS32™ Architecture for Programmers, Volume I: “Introduction to the MIPS32™ Architecture”.
6. MIPS32™ Architecture for Programmers Volume II: “The MIPS32™ Instruction Set”.



Computer Architecture

Lecturer: Mihai Negru

2nd Year, Computer Science

Lecture 8: Pipeline CPU Design

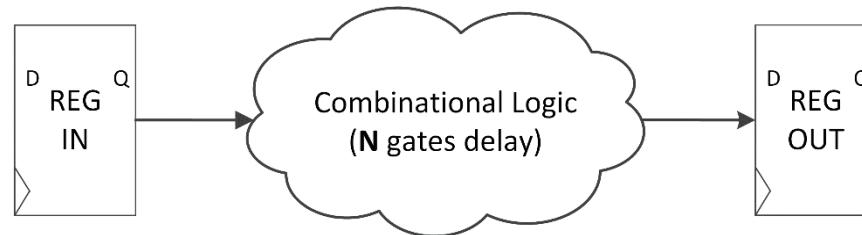
<http://users.utcluj.ro/~negrum/>



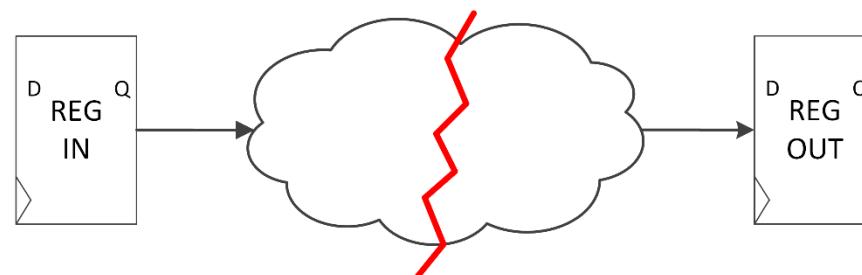
The problem with long critical paths



- The critical path limits the clock frequency for the circuit



- The clock period must cover at least the equivalent N gates delay (plus the setup/hold time of the registers)
- Time period for a new result: $\approx N \times (\text{delay/gate}) = D$
- Throughput (number of results/time interval): $\text{Throughput}_{\text{init}} = 1/D$
- Solution to reduce the clock period (greater frequency):
 - Partition the critical path with synchronous storage elements (registers)

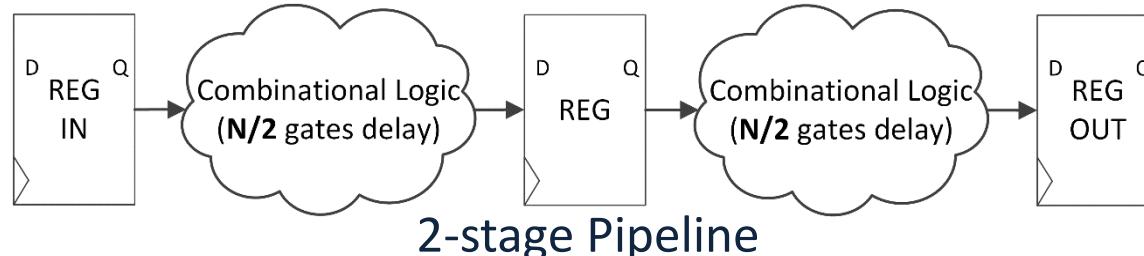




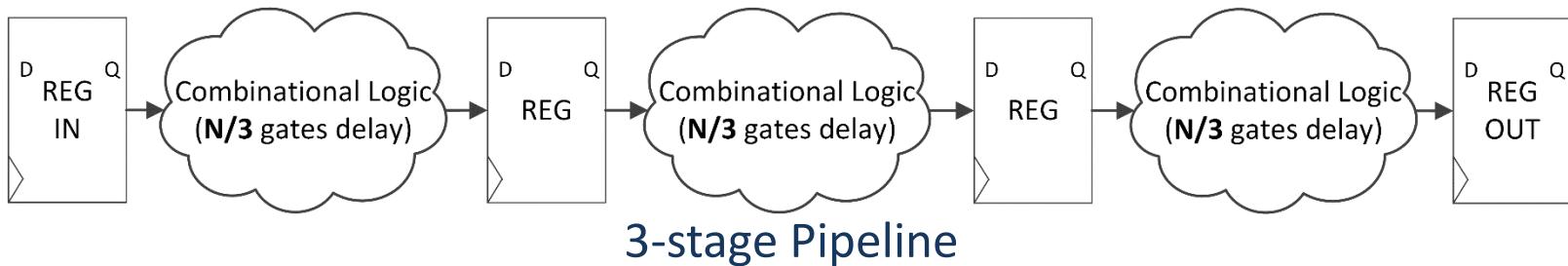
Partitioning the Critical Path



- Pipeline: a set of data processing elements (**stages**) connected in series; the output of one stage is the input of the next one



- The clock period must cover at least the equivalent $N/2$ gates delay
- Time period for a new result: $\approx (N/2) \times (\text{delay/gate}) = D/2$
- Potential throughput is double: $2/D = 2*(1/D) = 2*\text{Throughput}_{\text{init}}$



- The clock period must cover at least the equivalent $N/3$ gates delay
- Time period for a new result: $\approx (N/3) \times (\text{delay/gate}) = D/3$
- Potential throughput is double: $3/D = 3*(1/D) = 3*\text{Throughput}_{\text{init}}$



Partitioning the Critical Path

Performance vs. Cost



- Does the throughput grow indefinitely?
 - No – the intermediate registers introduce supplemental delays (cumulative)
- If the duration without pipeline is D and the duration introduced by an intermediate register is D_{reg} → throughput for $k / k+1$ stages:

$$Throughput(k) = \frac{1}{\frac{D}{k} + (k-1) \cdot D_{reg}}$$

$$Throughput(k+1) = \frac{1}{\frac{D}{k+1} + k \cdot D_{reg}}$$

- Throughput decreases ($Throughput(k+1) < Throughput(k)$) when:

$$D_{reg} > \frac{D}{k} - \frac{D}{k+1}$$

- In reality problems of additional cost → balance between cost and performance in order to determine the optimum number of stages k_{opt} :
 - $k < k_{opt}$ – under-pipelined design
 - $k > k_{opt}$ – over-pipelined design



Pipelining – General Aspects



- Pipelines allow overlapping execution of multiple instructions, exploiting the ILP (instruction level parallelism)
- The pipeline increases the instruction throughput (the number of instructions that can be executed in a unit of time)
- Pipeline decreases the clock cycle time
 - Ideal case: duration without pipeline / pipeline stages
 - In reality:
 - The stages will not be perfectly balanced
 - The pipeline rate is limited by the slowest pipeline stage
 - Other delays may appear due to hazards (stalls)
 - The setup/hold times of the intermediate registers
 - Time to “fill” pipeline and time to “drain” – reduce speedup
- In general **the pipeline is not visible to the programmer** (compiler)
- Pipelining yields a reduction in the **average execution time per instruction.**



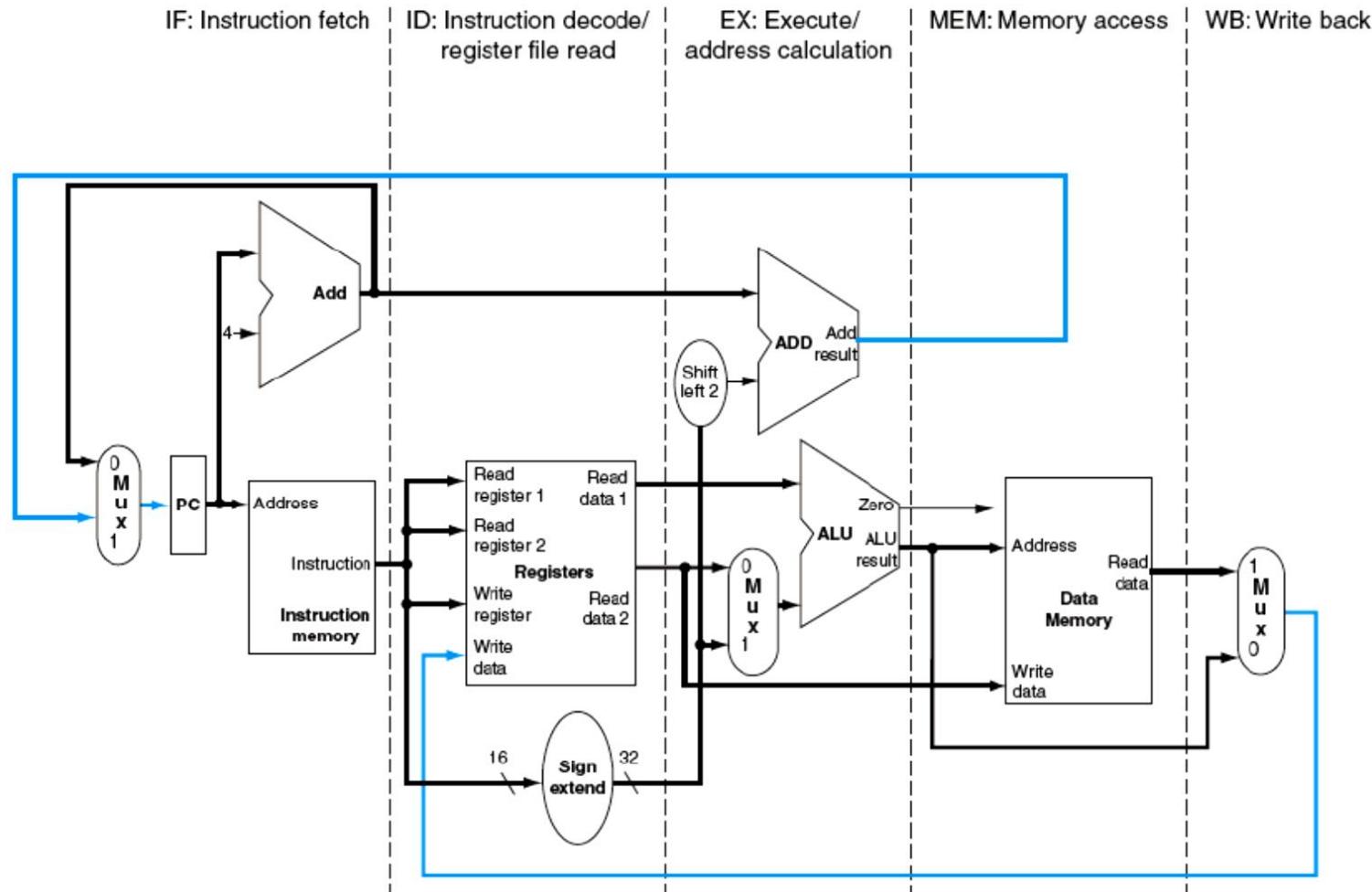
Pipelining – General Aspects



- Pipeline designer's goals
 - To balance the length of pipeline stages
 - To reduce the effects of the possible hazards
- ISA MIPS advantages for a pipeline implementation
 - All instructions are the same length
 - Just a few instruction formats, registers located in same place in instructions
 - Memory operands appear only in loads and stores
- What makes pipelining hard?
 - Structural hazards: suppose we had only one memory.
 - Control hazards: need to worry about branch/jump instructions
 - Data hazards: an instruction depends on a previous instruction
- We will build a simple pipeline CPU starting from the Single-Cycle CPU, look at the possible problems and find solutions



- Start from the single-cycle implementation, 5 stages





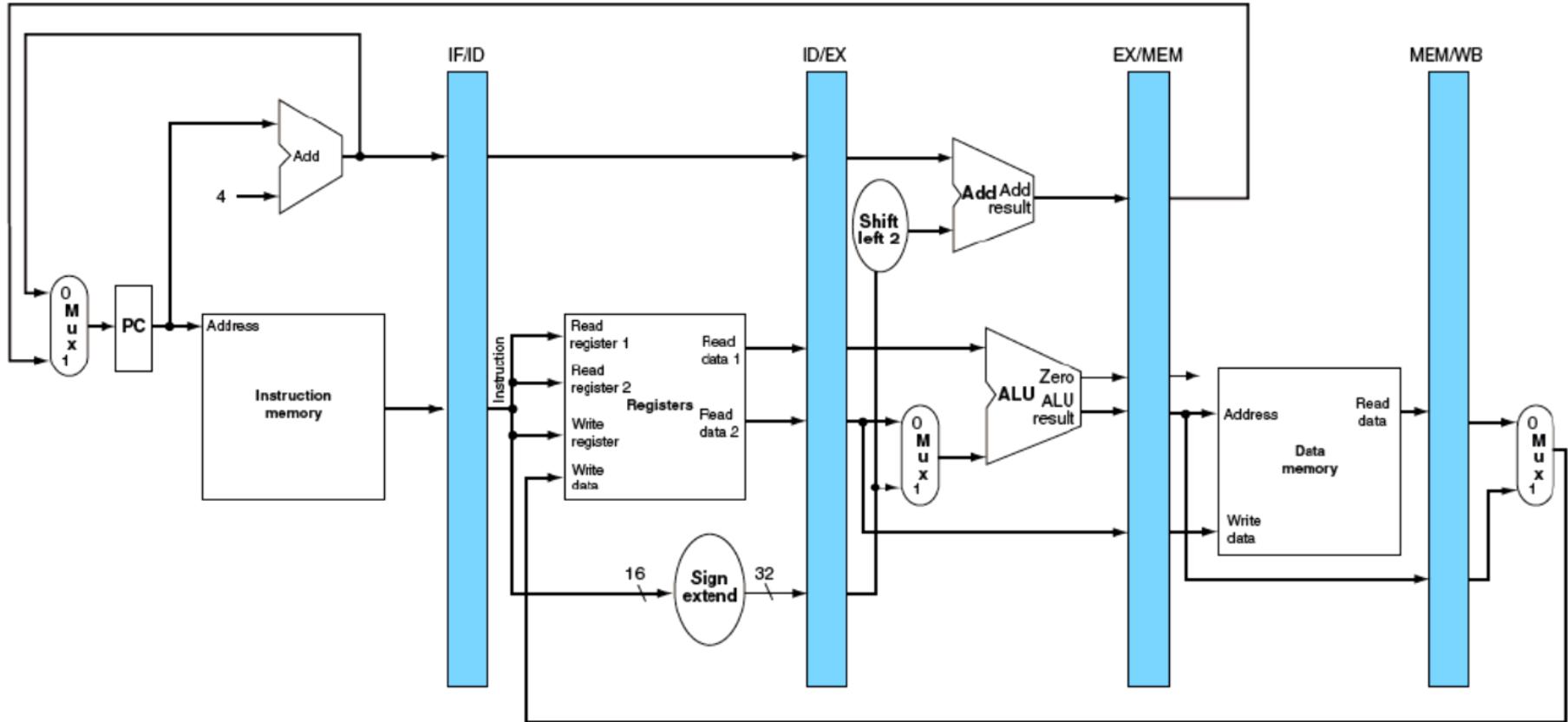
Pipeline CPU Design



- Basic idea
 - Single-cycle data path divided into IF, ID, EX, MEM, WB stages
 - Up to 5 instructions will be in execution during each clock cycle
 - Normal data flow is from left to right
 - The paths flowing from right to left (register write-back and PC on a branch) introduce complications into pipeline (forwarding, branch delay)
- How to actually split the data-path into stages?
 - Add registers between each pair of pipe stages.
 - The registers transfer data values and control information from one stage to the next.
 - PC: a pipeline register before the IF stage → one pipeline register for each stage. Practically PC is a part of the IF!



Pipeline CPU Design

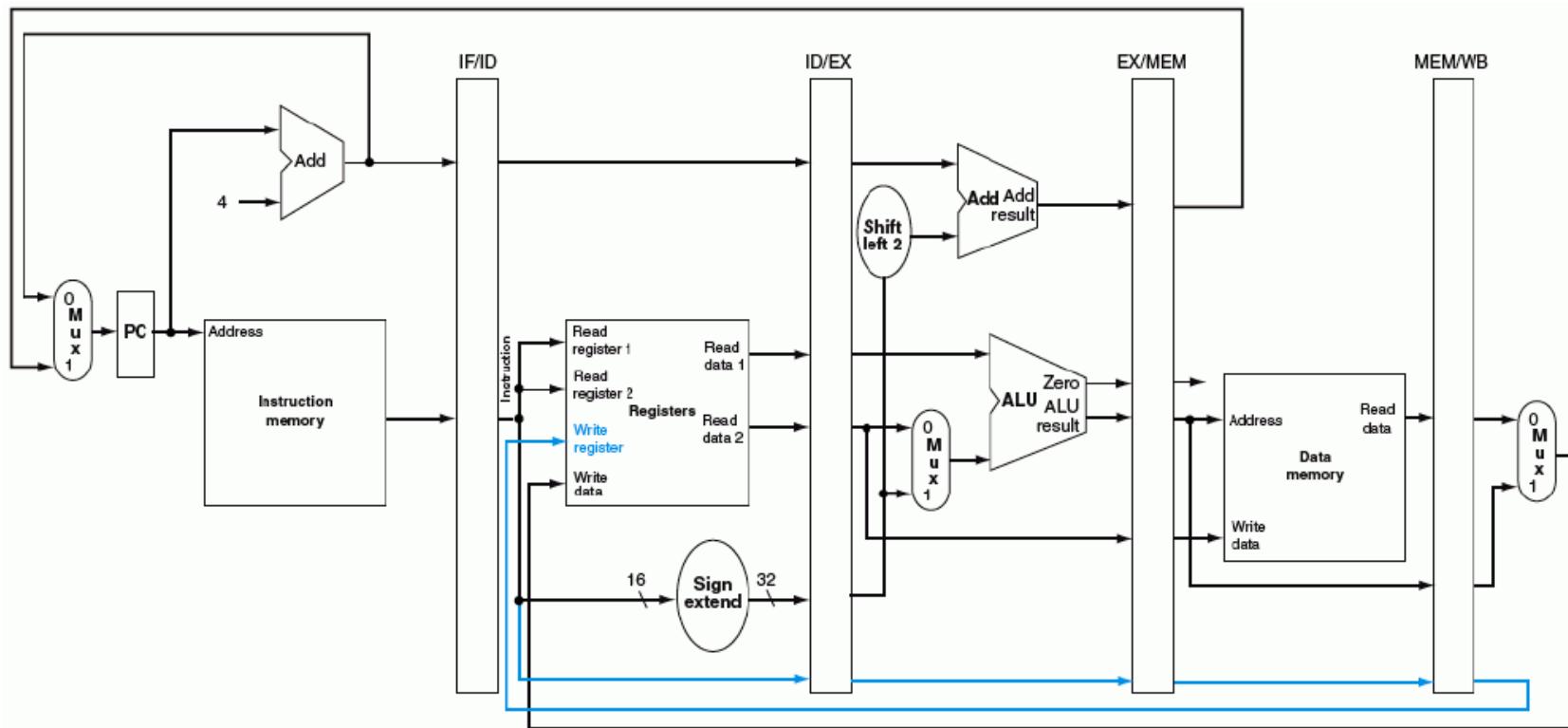


Single-Cycle data-path, with added registers between execution stages

- Can you find a problem even if there are no dependencies?
- What type of instructions present this problem?



- Answer:
 - Register File: write address and write data come from different instructions



Corrected Pipeline: The destination address and data are pipelined simultaneously



Actions in Pipeline Stages





Actions in Pipeline Stages



- EX – Execute Stage
 - Memory reference instructions (LW & SW): calculate the effective address
 - $\text{EX/MEM.ALUResult} \leftarrow \text{ID/EX.A} + \text{ID/EX.Imm}$ $(\text{RF}[rs] + \text{S_Ext(Imm)})$
 - Transmit Write data for SW
 - $\text{EX/MEM.WriteData} \leftarrow \text{ID/EX.B}$ $(\text{RF}[rt])$
 - Register-type ALU instructions:
 - $\text{EX/MEM.ALUResult} \leftarrow \text{ID/EX.A} (\text{ALUOp-func}) \text{ ID/EX.B}$ $(\text{RF}[rs] \text{ ALUOp-func } \text{R}[rt])$
 - Register-Immediate ALU instructions:
 - $\text{EX/MEM.ALUResult} \leftarrow \text{ID/EX.A} (\text{ALUOp-codop}) \text{ ID/EX.Imm}$ – Arithmetic
 - $\text{EX/MEM.ALUResult} \leftarrow \text{ID/EX.A} (\text{ALUOp-codop}) \text{ ID/EX.Imm}$ – Logical
 - Branch instructions: perform the equality test in ALU
 - $\text{EX/MEM.Zero} \leftarrow \text{ALUZero}$ $(\text{ID/EX.A} - \text{ID/EX.B})$
 - Compute the branch-target address in the additional Adder
 - $\text{EX/MEM.PC} \leftarrow \text{ID/EX.PC} + \text{ID/EX.Imm} \ll 2$ $(\text{PC} + 4 + \text{S_Ext(Imm)} \ll 2)$
 - The Write data address for Registers is transmitted
 - $\text{EX/MEM.WriteReg} \leftarrow \text{ID/EX.WriteRegRD or ID/EX.WriteRegRt}$



Actions in Pipeline Stages



- **MEM – Memory Stage**
 - LW: read data memory at the address computed in EX stage.
 - $\text{MEM/WB.LMD} \leftarrow \text{DM[EX/MEM.ALUResult]}$
 - LMD – Load Memory Data register
 - SW: write the value of the RF[rt] register into the data memory.
 - $\text{DM[EX/MEM.ALUResult]} \leftarrow \text{EX/MEM.WriteData}$
 - The Write data address for Registers is transmitted
 - $\text{MEM/WB.WriteReg} \leftarrow \text{EX/MEM.WriteReg}$
 - ALU result is transmitted
 - $\text{MEM/WB.ALUResult} \leftarrow \text{EX/MEM.ALUResult}$
 - BEQ: $(\text{EX/MEM.Zero}) \rightarrow \text{PC} \leftarrow \text{EX/MEM.PC}$ $(\text{PC} + 4 + \text{S_Ext(Imm)} \ll 2)$
- **WB – Write Back Stage**
 - Register – type and Register – Immediate ALU instructions:
 - Write the ALU result into the register file.
 - $\text{RF[MEM/WB.WriteReg]} \leftarrow \text{MEM/WB.ALUResult}$ $(\text{RF[rd]} \text{ or } \text{RF[rt]})$
 - LW: Write the data read from Data Memory into the Register File
 - $\text{RF[MEM/WB.WriteReg]} \leftarrow \text{MEM/WB.LMD}$ (RF[rt])
 - Write to Register File → in the **first half** of the clock cycle



Actions in Pipeline Stages



Pipeline RTL Summary

| | R – R (R-type) | R–I (I-type, aritm/logic) | LW | SW | BEQ |
|-----|--|--|--|---|--|
| IF | IF/ID.IR \leftarrow IM[PC] IF/ID.PC \leftarrow PC + 4 PC \leftarrow PC + 4 or (EX/MEM.Zero&Branch) \rightarrow PC \leftarrow EX/MEM.PC | | | | |
| ID | ID/EX.A \leftarrow RF[rs] ID/EX.B \leftarrow RF[rt] ID/EX.Imm \leftarrow S_Ext(Imm) or ID/EX.Imm \leftarrow Z_Ext(Imm) ID/EX.WriteRegRd \leftarrow rd, ID/EX.WriteRegRt \leftarrow rt ID/EX.PC \leftarrow IF/ID.PC | | | | |
| EX | EX/MEM.ALUResult \leftarrow ID/EX.A (ALUOp-Func) ID/EX.B | EX/MEM.ALUResult \leftarrow ID/EX.A (ALUOp-codop) ID/EX.Imm | EX/MEM.ALUResult \leftarrow ID/EX.A + ID/EX.Imm | EX/MEM.ALUResult \leftarrow ID/EX.A + ID/EX.Imm EX/MEM.WriteData \leftarrow ID/EX.B | EX/MEM.Zero \leftarrow ALUZero (ID/EX.A – ID/EX.B) EX/MEM.PC \leftarrow ID/EX.PC + ID/EX.Imm << 2 |
| | EX/MEM.WriteReg \leftarrow ID/EX.WriteRegRd or ID/EX.WriteRegRt | | | | |
| MEM | MEM/WB.ALUResult \leftarrow EX/MEM.ALUResult | MEM/WB.ALUResult \leftarrow EX/MEM.ALUResult | MEM/WB.LMD \leftarrow DM[EX/MEM. ALUresult] | DM[EX/MEM. ALUResult] \leftarrow EX/MEM.WriteData | EX/MEM.Zero \rightarrow PC \leftarrow EX/MEM.PC |
| | MEM/WB.WriteReg \leftarrow EX/MEM.WriteReg | | | | |
| WB | RF[MEM/WB.WriteReg] \leftarrow MEM/WB.ALUResult | RF[MEM/WB.WriteReg] \leftarrow MEM/WB.ALUResult | RF[MEM/WB.WriteReg] \leftarrow MEM/WB.LMD | | |

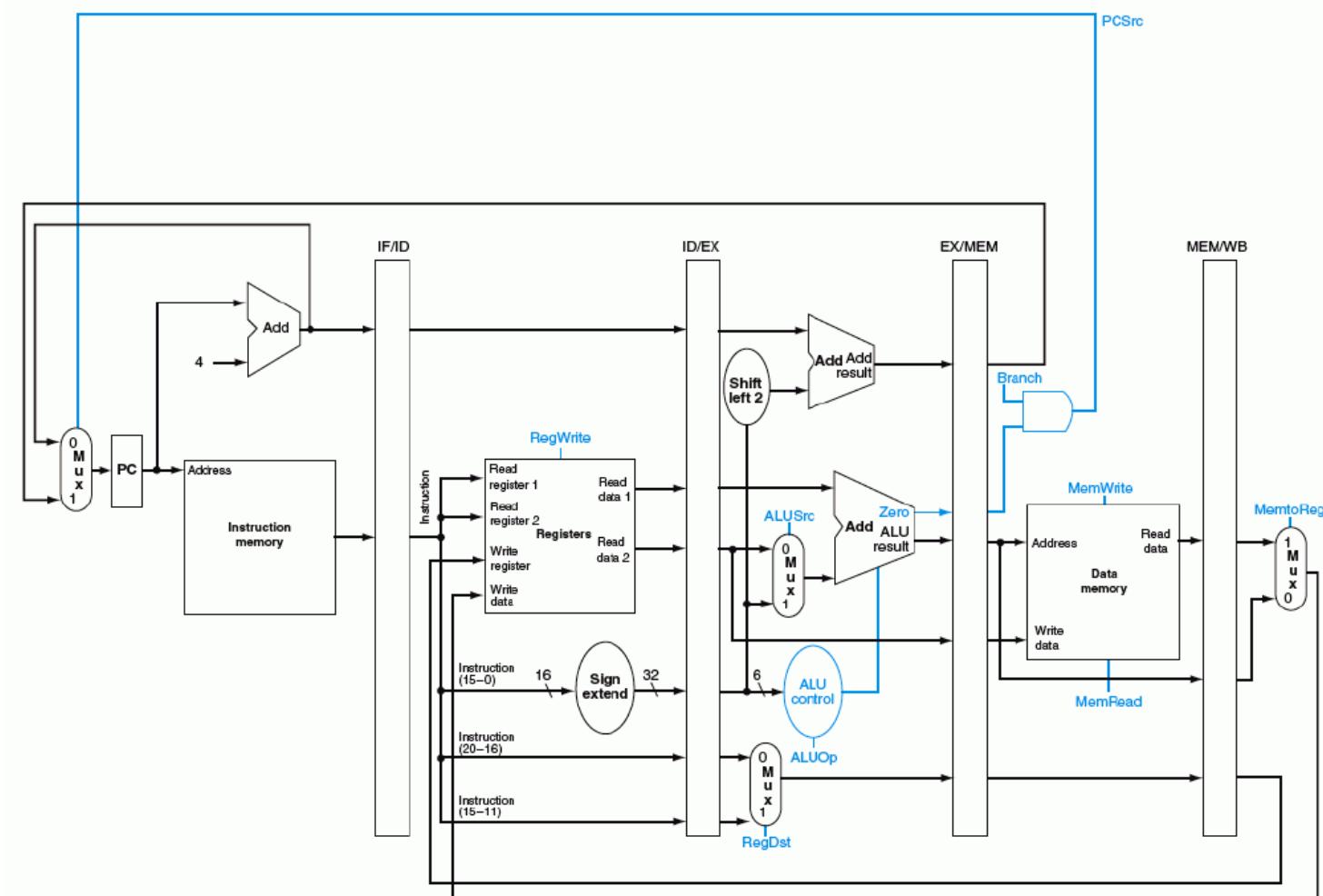
What else must be transferred along with the data in order for the instructions to work?



Pipeline CPU Design – Control



What needs to be controlled in each stage?



MIPS Pipeline with Control Signals



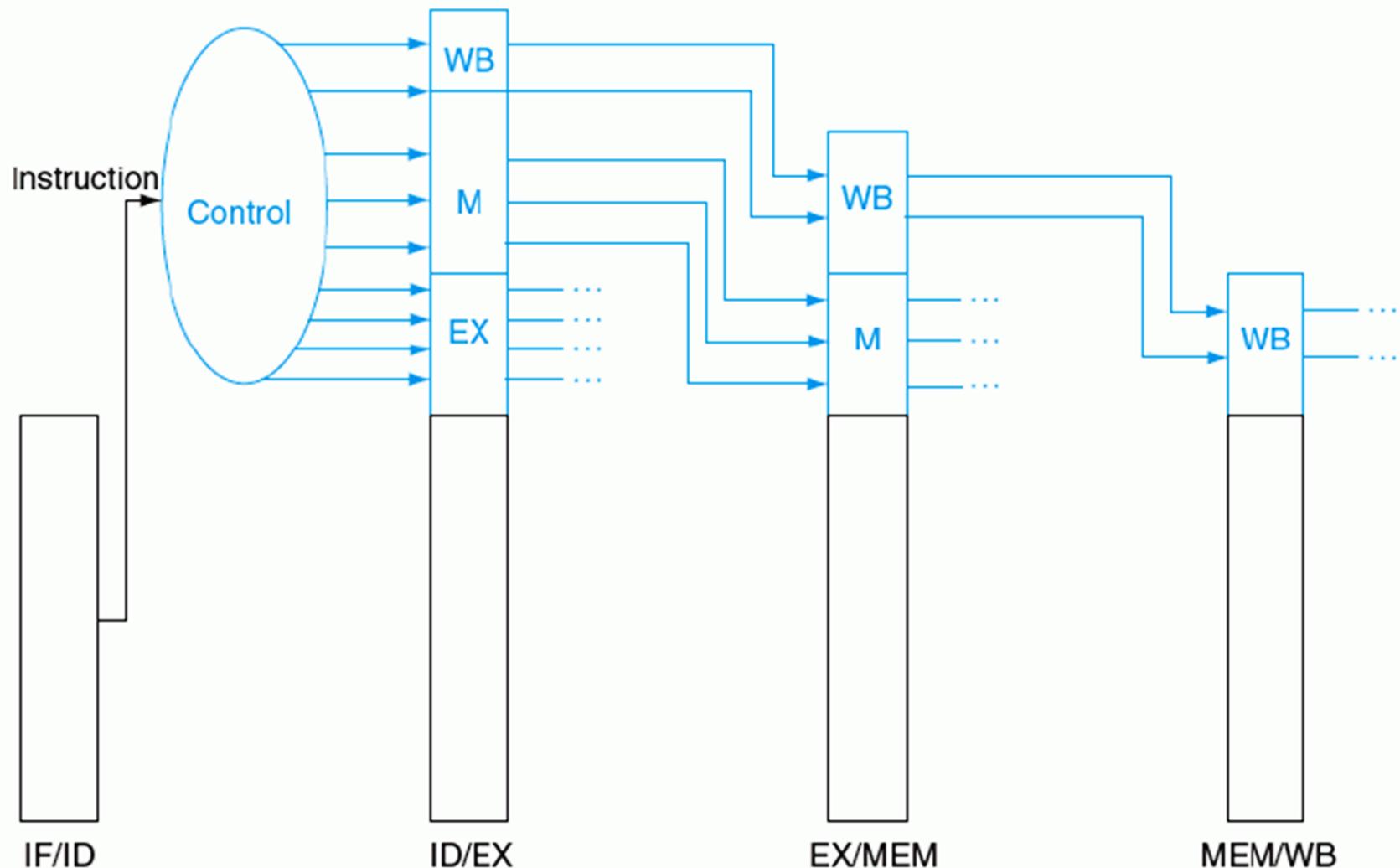
Pipeline CPU Design – Control



| Instr | EX Stage | | | MEM Stage | | | WB Stage | |
|--------|----------|-------|--------|-----------|---------|----------|----------|----------|
| | RegDst | ALUOp | ALUSrc | Branch | MemRead | MemWrite | RegWrite | MemtoReg |
| R-type | 1 | 10 | 0 | 0 | 0 | 0 | 1 | 0 |
| LW | 0 | 00 | 1 | 0 | 1 | 0 | 1 | 1 |
| SW | X | 00 | 1 | 0 | 0 | 1 | 0 | X |
| BEQ | X | 01 | 0 | 1 | 0 | 0 | 0 | X |

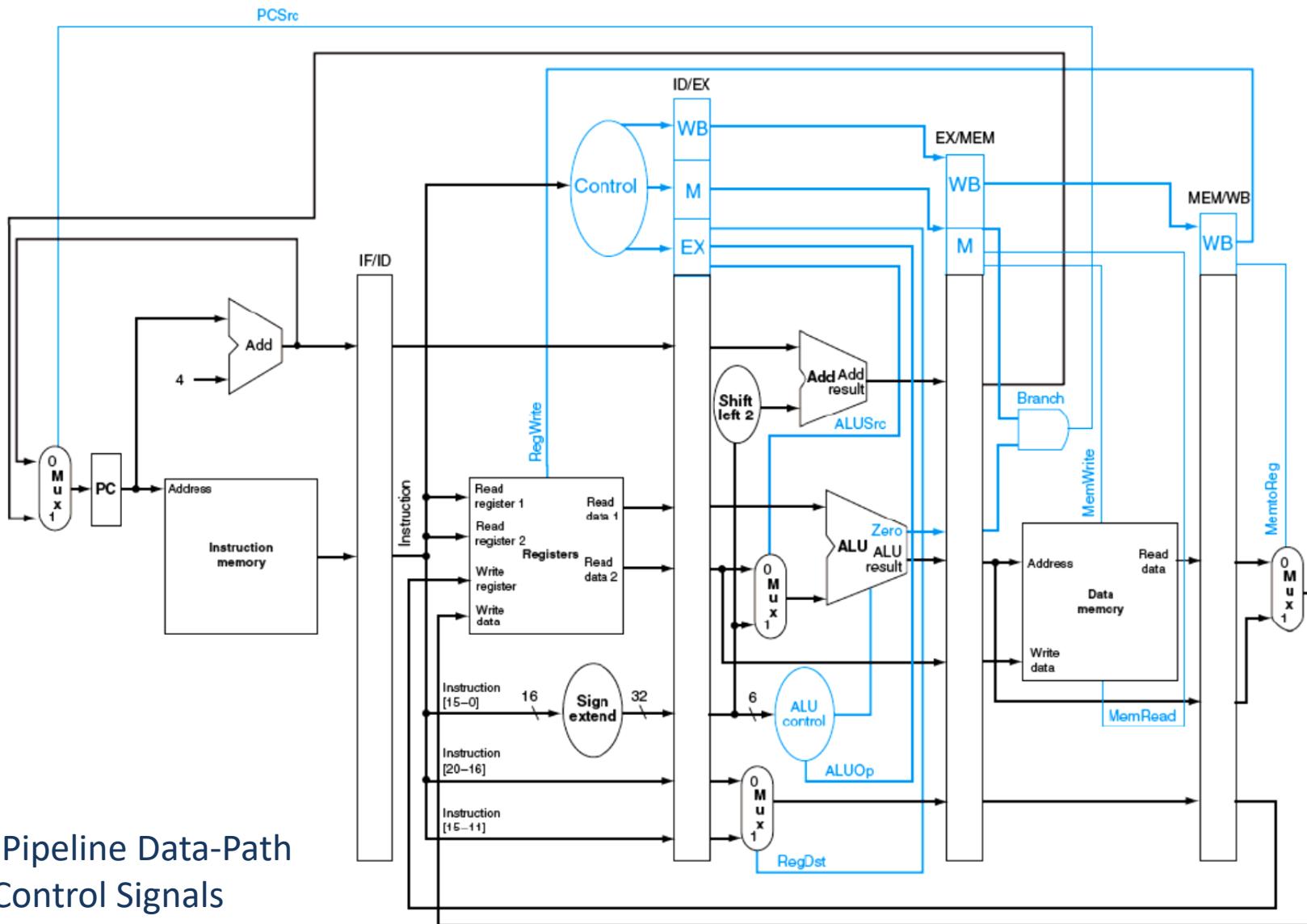
MIPS Pipeline Control Settings in different stages (IF and ID are common)

- Pipeline Control Signals must “travel” together with the intermediate results
 - The Main Control Unit generates the control signals during ID Stage
 - Control signals for EX Stage (RegDst, ALUOp, ALUSrc) are used 1 clock cycle later
 - Control signals for MEM (MemWrite, Branch, ...) are used 2 clock cycles later
 - Control signals for WB (MemtoReg, RegWrite) are used 3 clock cycles later
 - Pass control signals along just like the data from one stage to another



The control lines for the final three stages

Pipeline CPU Design – Control



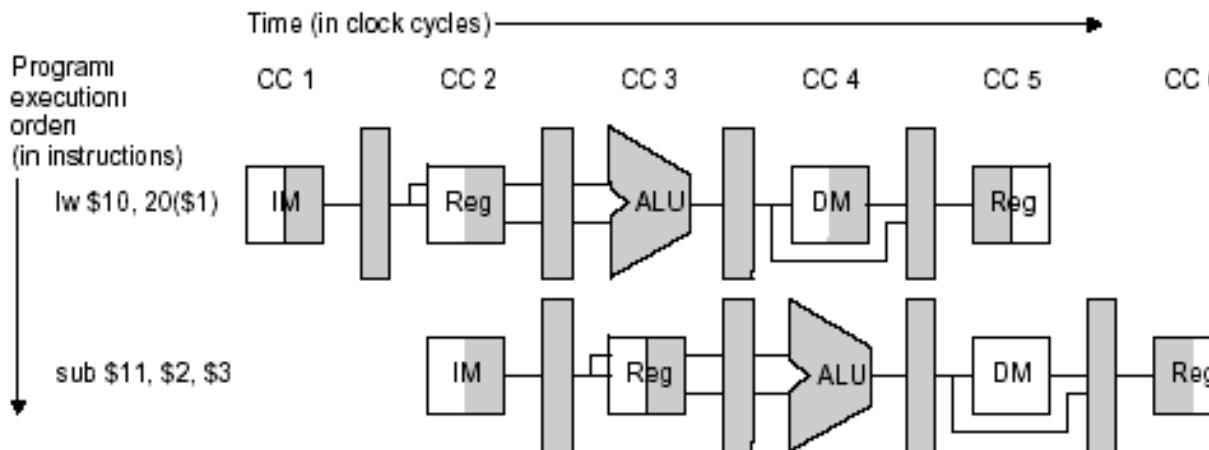
MIPS Pipeline Data-Path
with Control Signals



Pipeline Representations



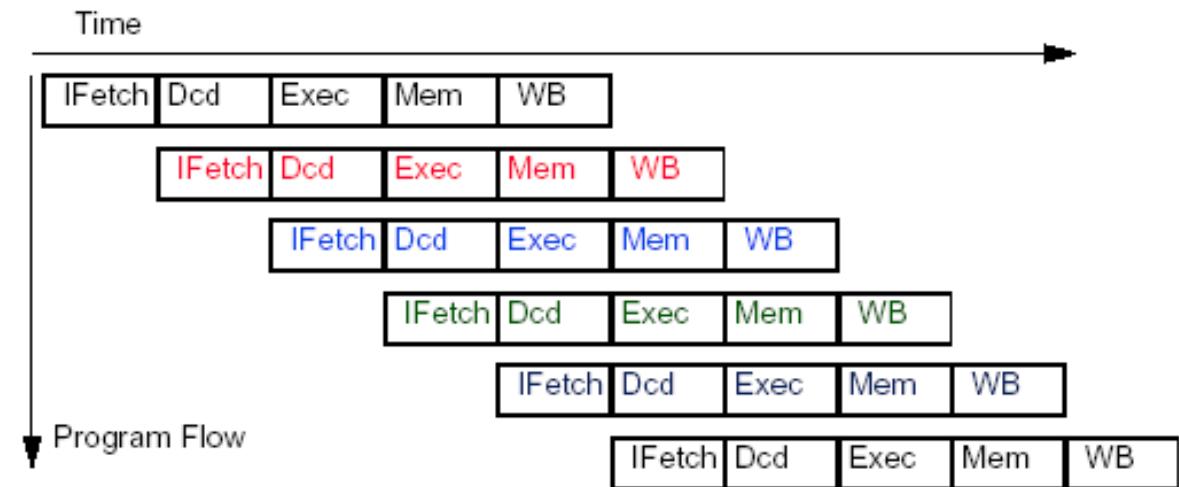
Graphically Representing Pipelines



Utility:

- How many cycles does it take to execute this code?
- What is the ALU doing during cycle 4?

Conventional Pipelined Execution Representation



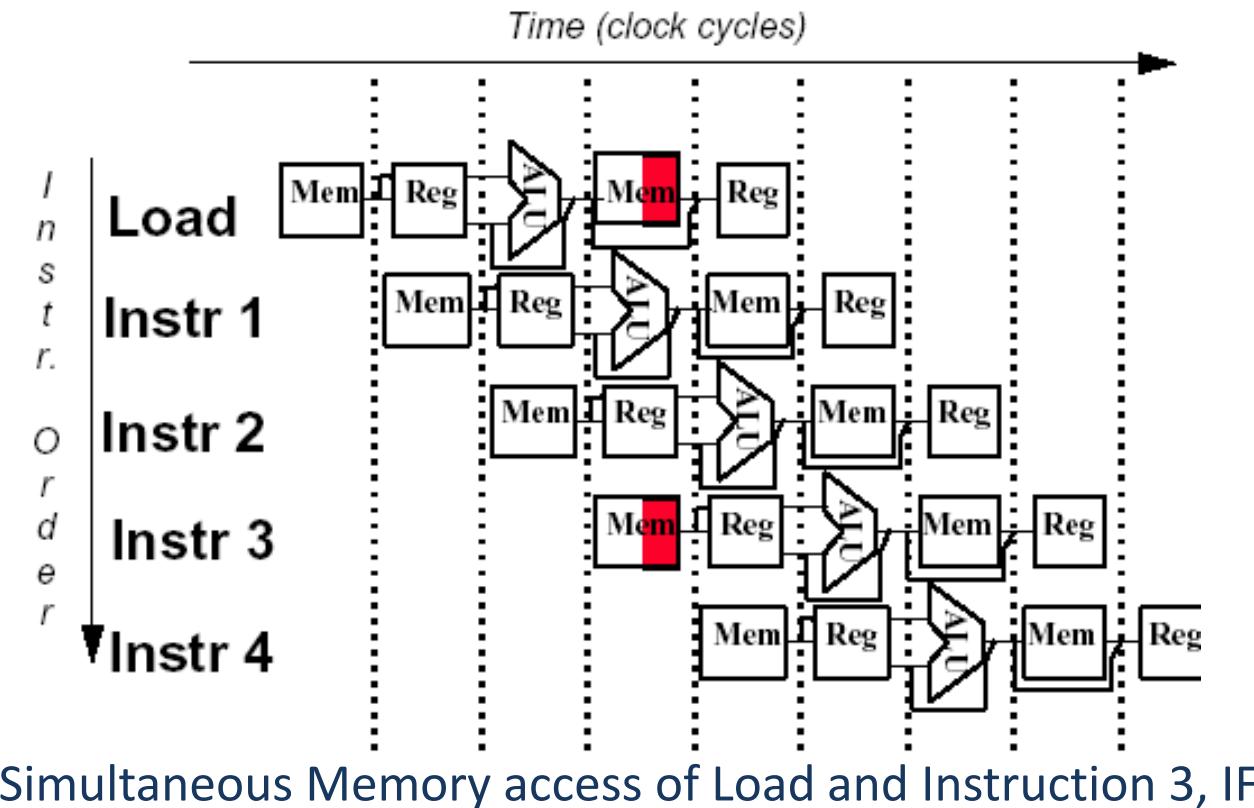


Pipeline CPU Design – Hazards



- Hazards – Situations in pipelining when the next instruction cannot be executed in the following clock cycle
- 3 types of Hazards
 - **Structural Hazards** (resource dependency) – two instructions attempt to use the same resource at the same time → resource constraints
 - **Data Hazards** (data dependency)
 - Attempt to use data before it is ready (data availability)
 - For an instruction in ID stage the source operand(s) are produced by a prior instruction still in the pipeline
 - **Control Hazards** (flow control)
 - Attempt to make a decision about program control flow before the condition has been evaluated and the new PC target address calculated
 - Pipelining of branches, jumps & other instructions that change the PC
- Can always resolve hazards by waiting
 - Pipeline control must detect the hazard and take action to resolve hazards
 - Common solution is to stall the pipeline until the hazard is resolved, inserting one or more “bubbles” (NoOps) in the pipeline

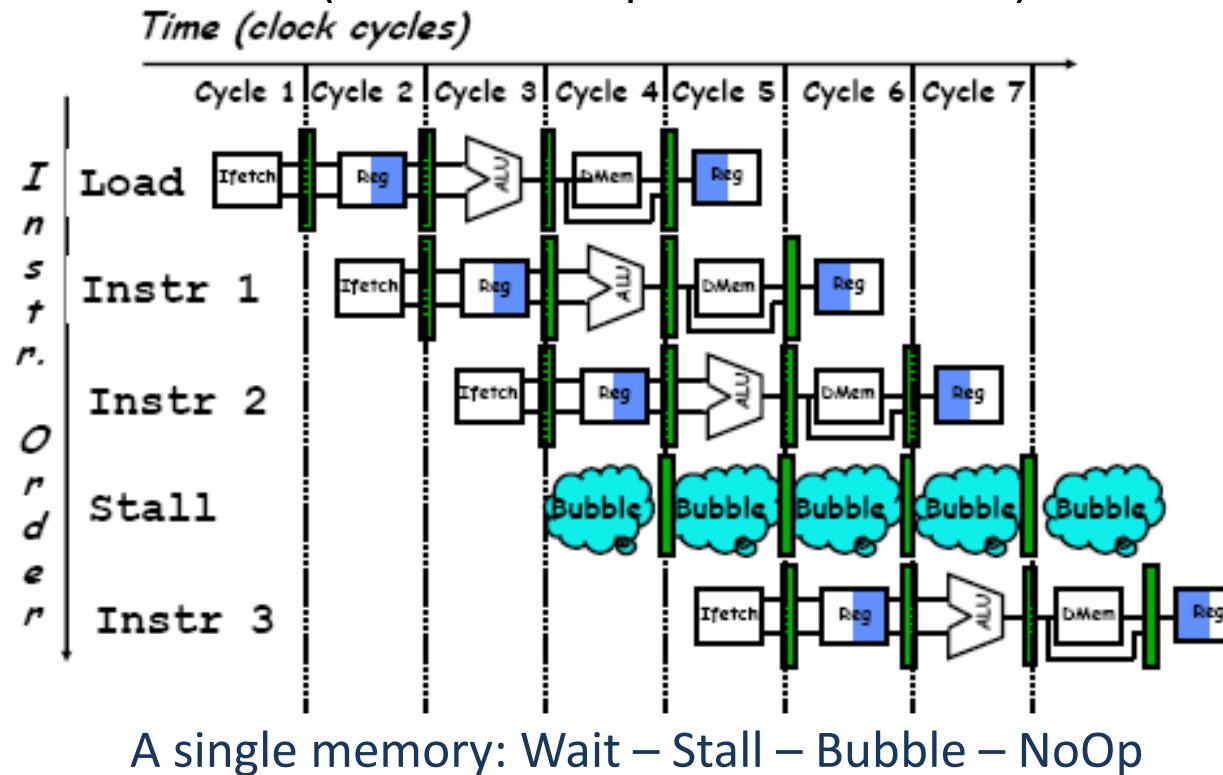
- Structural Hazards
 - Single Memory → Structural Hazard



Is there a solution?

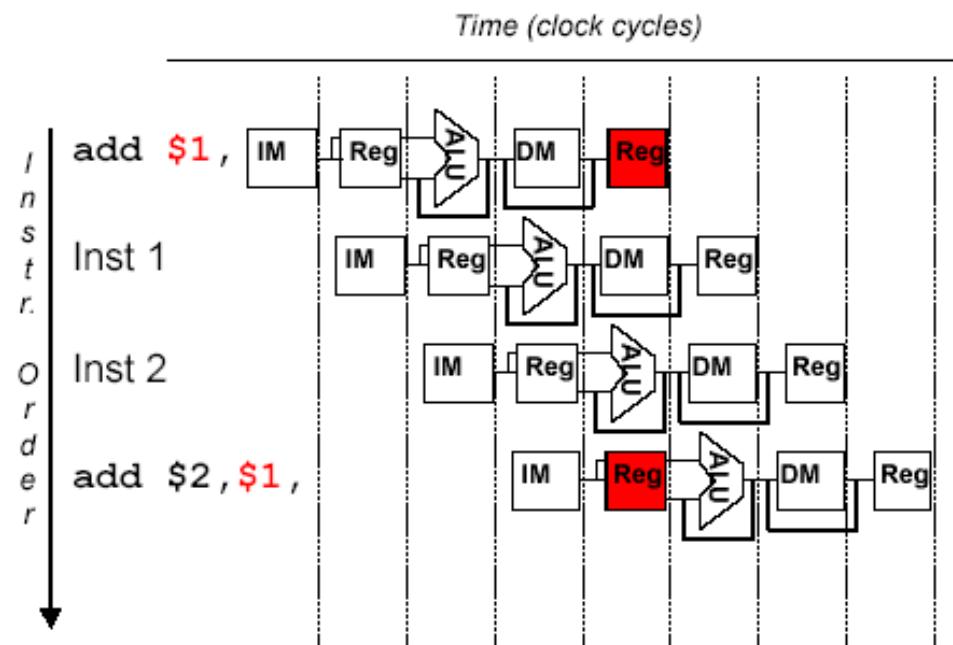


- Structural Hazard
 - Single Memory
 - Wait solution – slow (reduces the speed of execution)



The used solution – Independent Instruction and Data (cache in real processors)
memories; IM & DM are already “inherited” from the single-cycle processor

- Structural Hazard
 - Simultaneous access to the Register File → Structural hazard
 - Read operands in ID stage and write result in WB stage

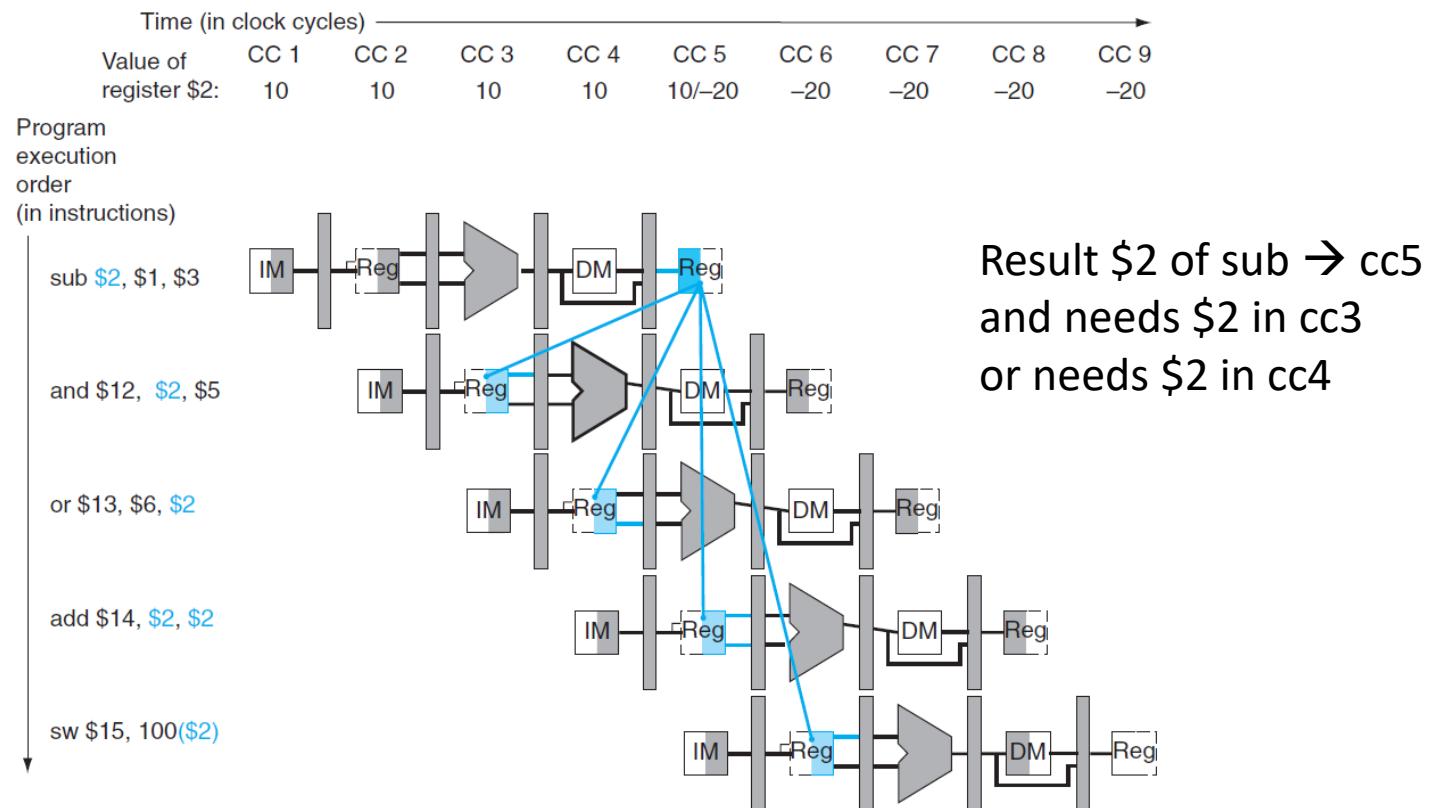


Simultaneous Register File access in WB (write) and ID (read) stages

- Solution:
 - Register file writes in the first half and reads in the second half of the cycle
 - Reads are asynchronous



- Data Hazards and Forwarding
 - The operands of the instructions are not available yet (they will be produced by previous instructions in the pipeline)



Dependencies that “go backward in time” cause data hazards



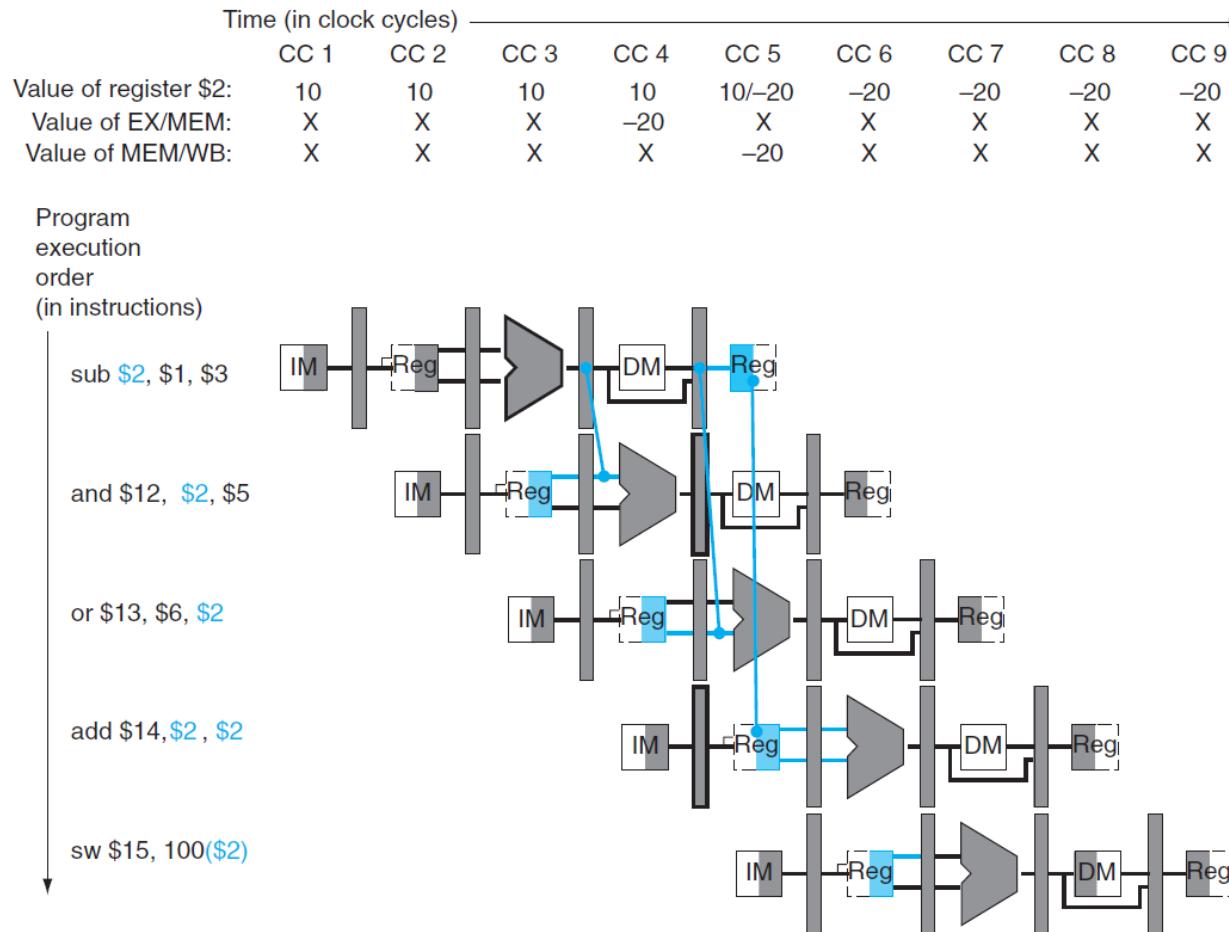
Pipeline CPU Design – Hazards



- Data Hazards and Forwarding
 - This hazard appears in ID stage, when an operand is in other pipeline stages
 - The usual name of this type of data hazard is **RAW (read after write)**
 - Software solution (programmer or compiler) – insert NoOps
 - Hardware solution – use **Forwarding**
- Forwarding – use temporary results, don't wait for them to be written in the Register File
 - “Forward” result from one stage to another to avoid the data hazard
 - The forwarded result can go to either ALU input; both ALU inputs can use forwarded inputs from the same or from different pipeline registers



- Data Hazards and Forwarding

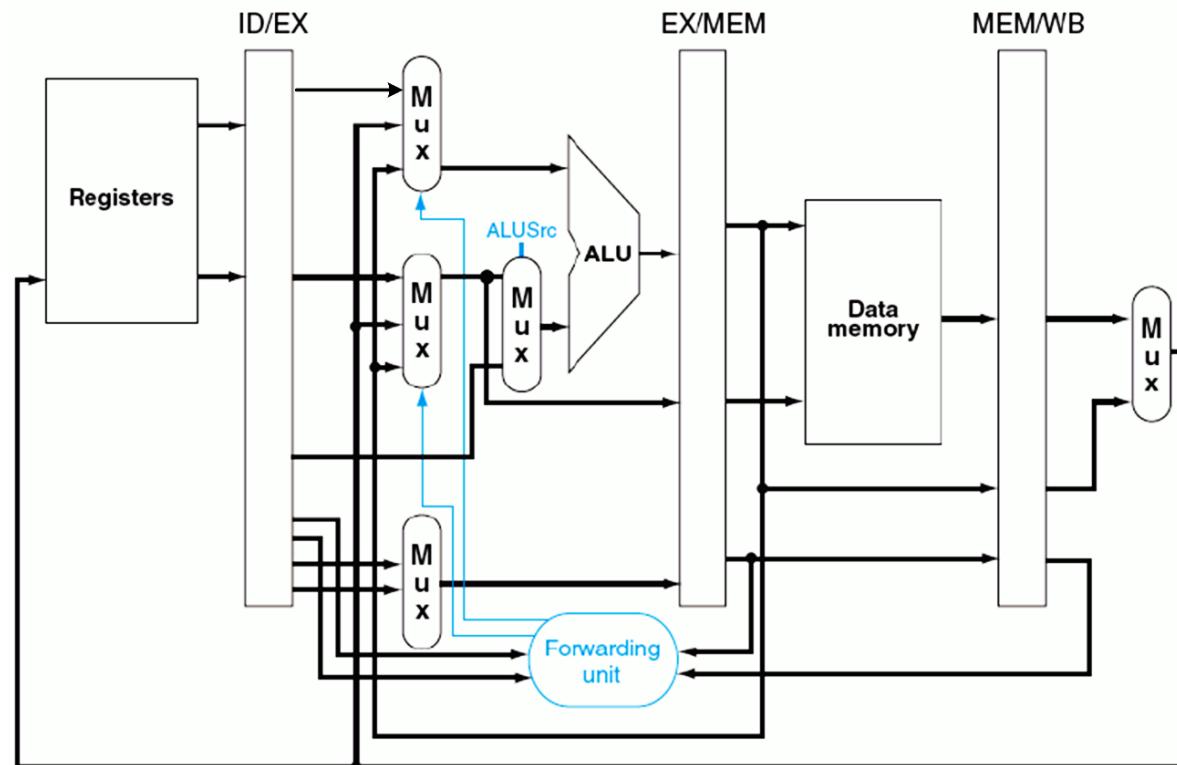


- Forwarding** – the use of the result from the EX and MEM stages, before they are written in the Register File

- Forwarding** of **sub** result from **EX/MEM**, **MEM/WB** to the input of the **ALU** for **and** & **or** instructions in **cc4, cc5**

- The **forwarding** for the **add** instruction through the Register File is **implicit**

- Hazard Detection for Forwarding
 - Hazard must be detected before execution
 - Detect if rs, rt at EX stage equals rd, rt in MEM or WB stage
 - In case of hazard, the data should be forwarded to the input of the ALU



Forwarding Unit and Multiplexers added to implement Forwarding



Pipeline CPU Design – Hazards



- Hazard Detection for Forwarding
 - Three sources for each MUX
 - ID/EX: no forwarding, just from the register file (00)
 - EX/MEM: forwarded data from the prior ALU results (10)
 - MEM/WB: from data memory or an earlier ALU result (01)
 - Notation: ID/EX.RegisterRs = The rs address field of the Register File in ID/EX pipeline register
 - Detect the Hazard condition
 - Do not forward when it is unnecessary, check the RegWrite signal
 - The control logic for Forwarding is implemented by the **Forwarding Unit**



Pipeline CPU Design – Hazards



- Data Forwarding Control Conditions
 - MEM Stage Hazard (consecutive instructions, distance = 1)

if (EX/MEM.RegWrite and
(EX/MEM.RegisterRd != 0) and (no forwarding for reg.nr = 0)
(EX/MEM.RegisterRd = ID/EX.RegisterRs))
ForwardA = 10

if (EX/MEM.RegWrite and
(EX/MEM.RegisterRd != 0) and
(EX/MEM.RegisterRd = ID/EX.RegisterRt))
ForwardB = 10

- Forwards the result from the previous instruction to either input of the ALU



Pipeline CPU Design – Hazards



- Data Forwarding Control Conditions
 - WB Stage Hazard (consecutive instructions, distance = 2)

| Tipical Problem, WB hazard | Atypical problem, MEM hazard |
|---|--|
| <p>add \$1, \$1, \$2</p> <p>add \$5, \$2, \$3</p> <p>add \$1, \$1, \$4</p> | <p>add \$1, \$1, \$2</p> <p>add \$1, \$1, \$3</p> <p>add \$1, \$1, \$4</p> |

- The most recent result must be forwarded
- The hazard in MEM Stage has priority, if it exists
- The hazard in WB Stage is resolved only if there is no MEM Stage hazard with the previous instruction



Pipeline CPU Design – Hazards



- Data Forwarding Control Conditions
 - WB Stage Hazard

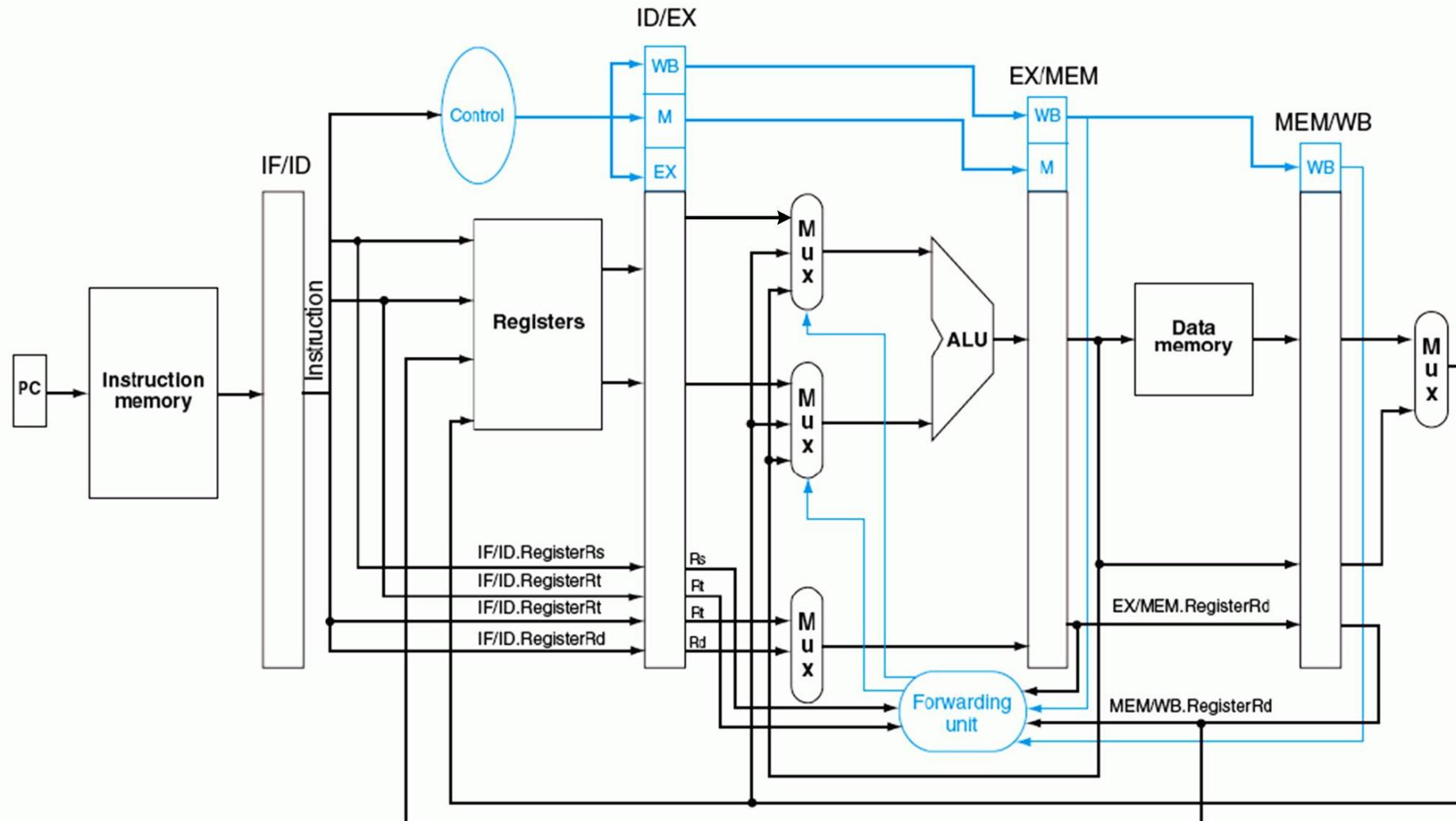
```
if (MEM/WB.RegWrite
    and (MEM/WB.RegisterRd != 0)
    and (EX/MEM.RegisterRd != ID/EX.RegisterRs)      (No MEM Hazard)
    and (MEM/WB.RegisterRd = ID/EX.RegisterRs))       (WB Hazard condition)
                                                       ForwardA = 01

if (MEM/WB.RegWrite
    and (MEM/WB.RegisterRd != 0)
    and (EX/MEM.RegisterRd != ID/EX.RegisterRt)        (No MEM Hazard)
    and (MEM/WB.RegisterRd = ID/EX.RegisterRt))         (WB Hazard condition)
                                                       ForwardB = 01
```

- Forwards the result from the previous instruction to either input of the ALU

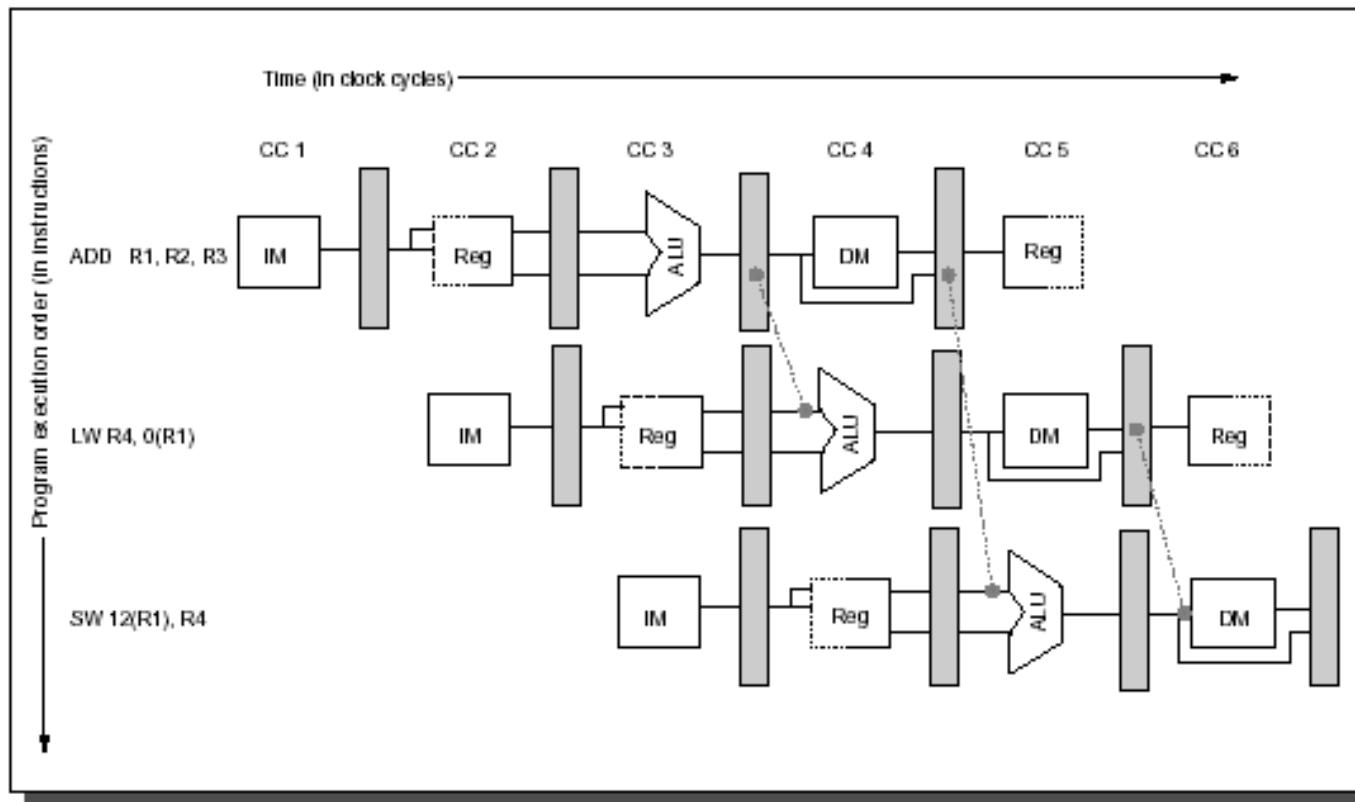


Pipeline CPU Design – Hazards



MIPS Pipeline Data-Path with forwarding (incomplete – some things are missing)

- Forwarding can also be applied to memory reference instructions



Example for LW-SW instructions:

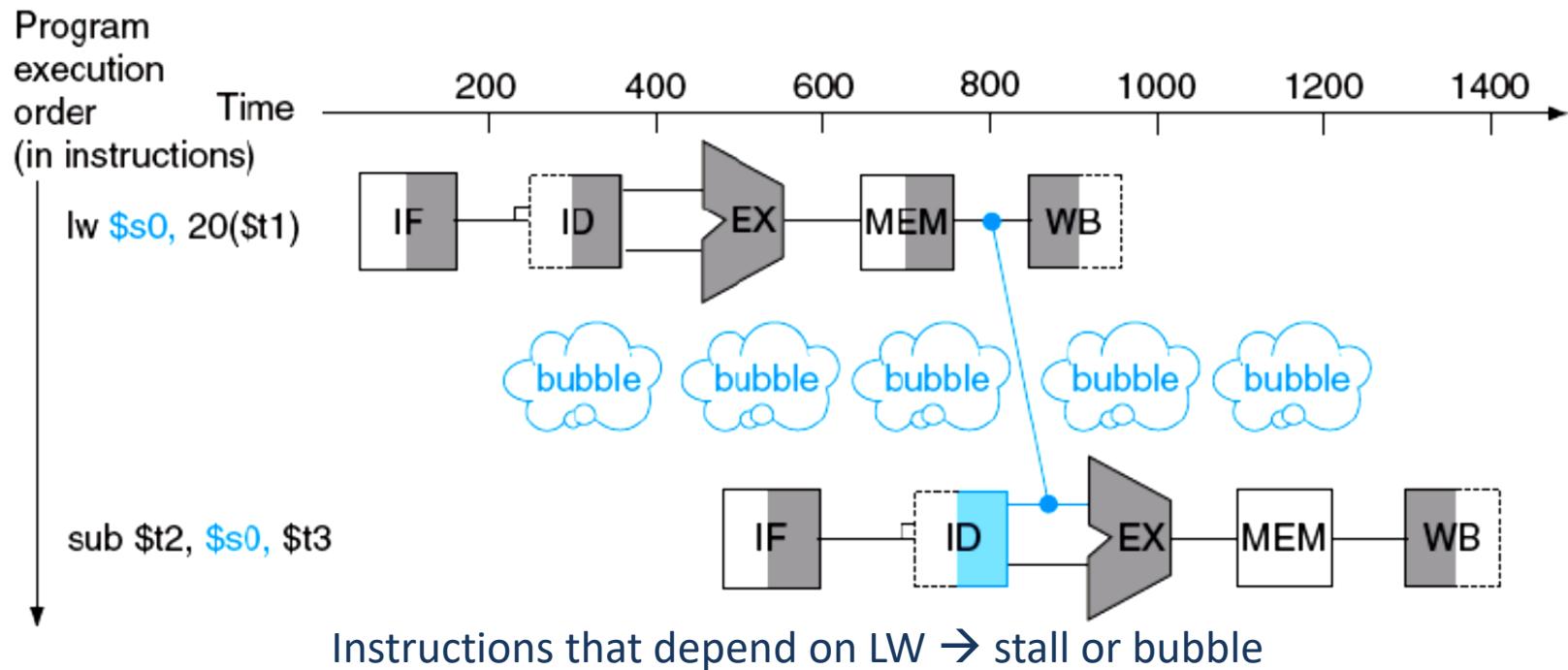
- Forwarding in CC4 and CC5 – the same as before
- Forwarding in CC6 – we need another multiplexer at the input of the memory in order to allow the forwarding of the output from the memory from the previous clock cycle



Pipeline CPU Design – Hazards



- Data Hazard and Stalls
 - Some hazards can only be resolved by **stalling** – forwarding does not help



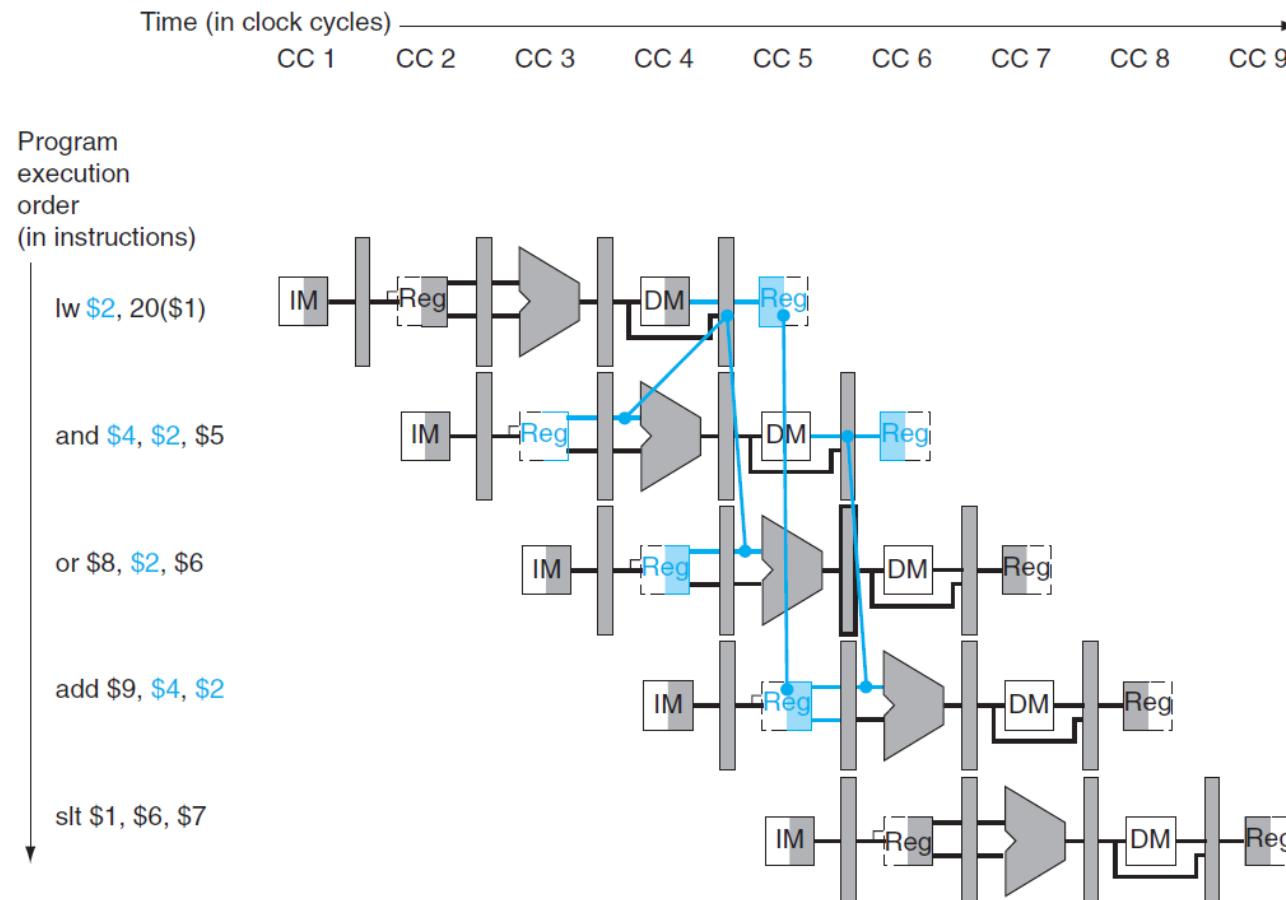
- Forwarding is not possible until the data is read from memory
- LW can cause hazards
 - The instruction that follows after a LW reads the register written by LW
- Named: Load Data Hazard



Pipeline CPU Design – Hazards



- Data Hazard and Stalls



The hazard between LW and AND cannot be solved by forwarding, AND has to wait

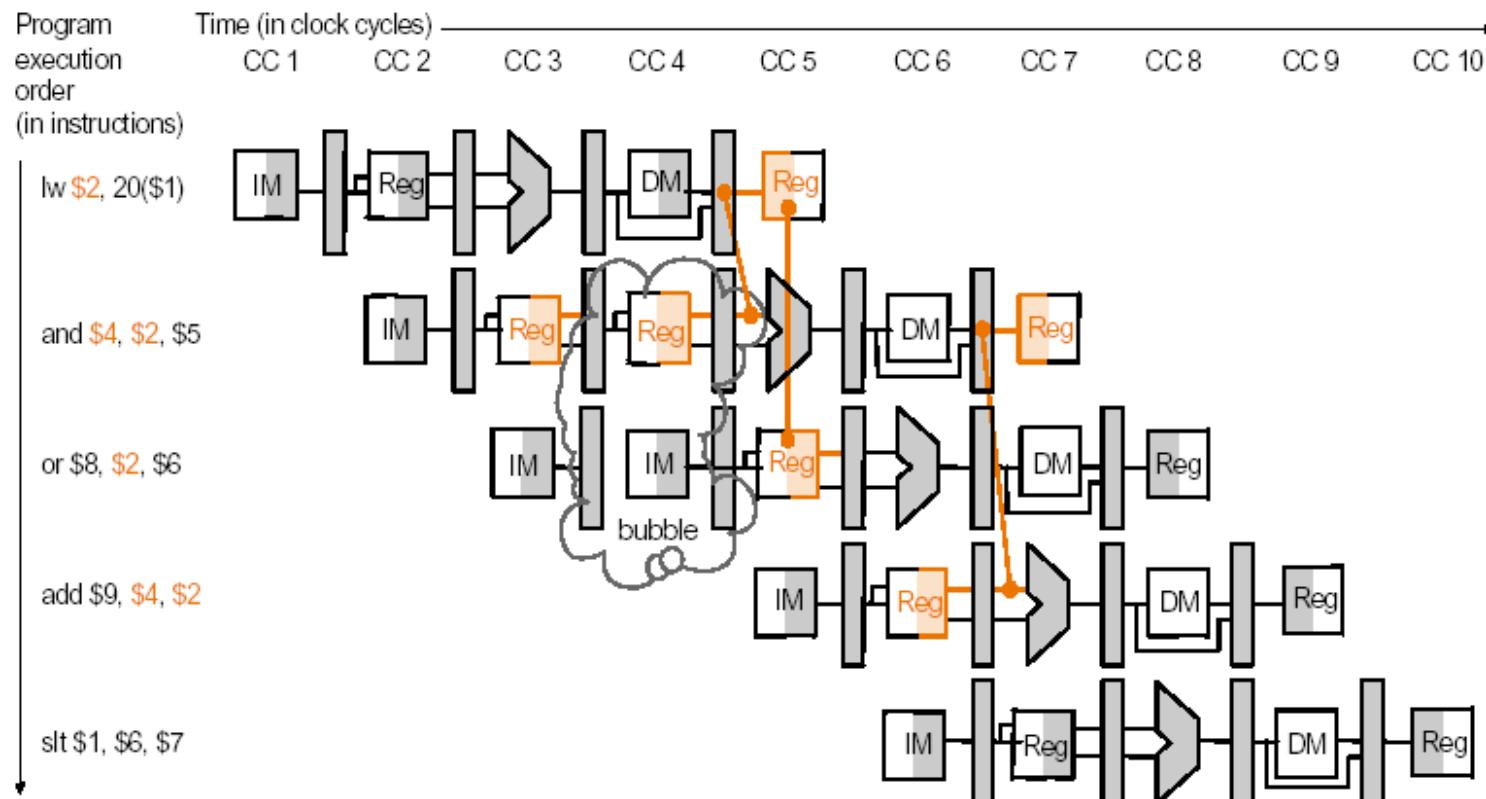
Solution: We introduce a wait cycle (bubble / stall / NoOp) in cc4 → NOP instead of AND



Pipeline CPU Design – Hazards



- Data Hazard and Stalls



For Load Data Hazard, even if forwarding is applied for LW, we need a stall (bubble)

- The instruction dependent on LW must be stalled
- A **Hazard Detection Unit** in ID stage to insert a “stall” between the LW & AND instructions
- The hardware equivalent to introducing a NoOp instruction after the LW



Pipeline CPU Design – Hazards

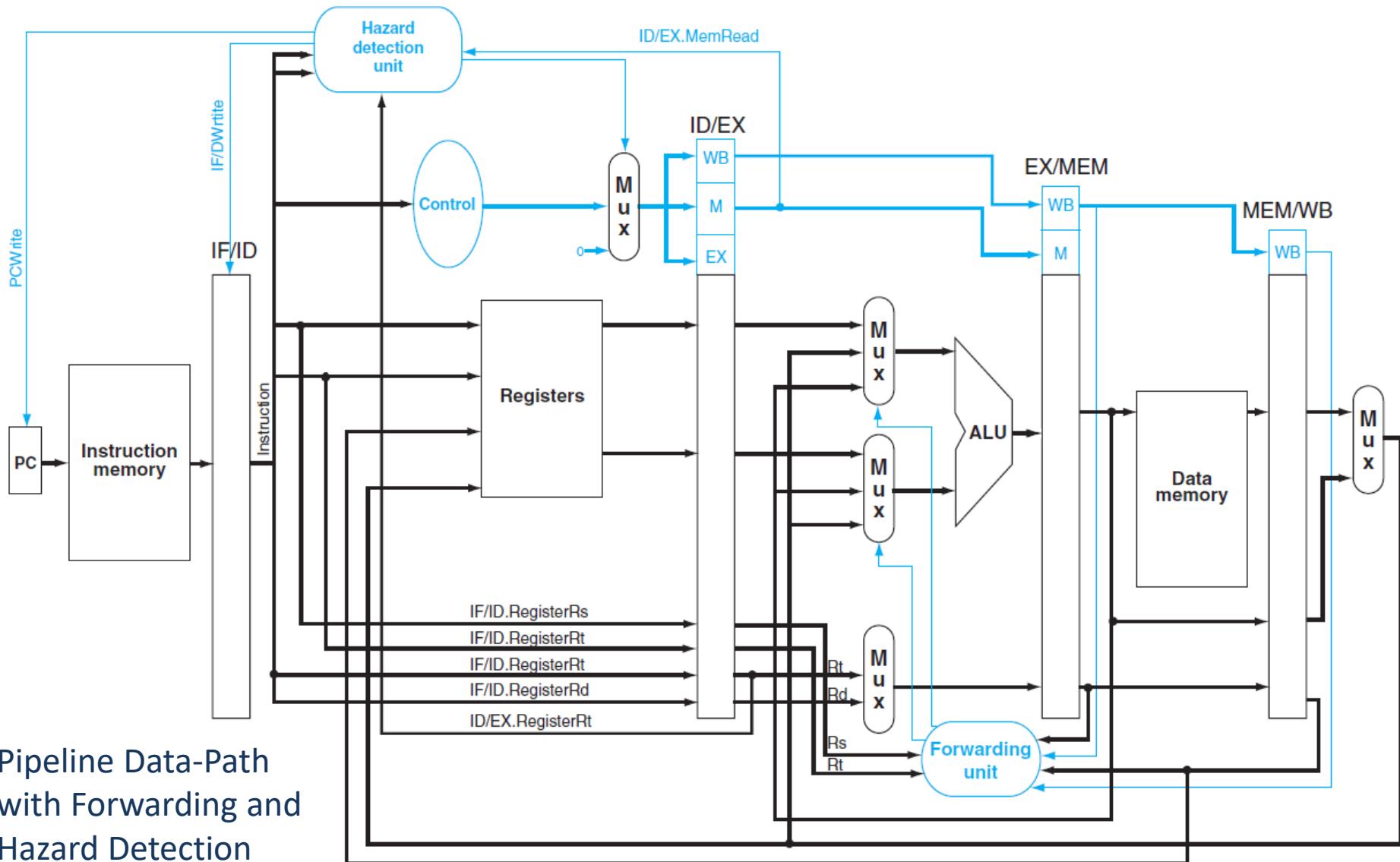


- Data Hazard and Stalls
 - Hazard Detection Unit for LW (in ID stage)
 - Verifies if the instruction in EX stage is LW
 - If yes and if the destination of the LW instruction is one of the sources for the instruction in the ID stage, then the pipeline will be blocked in the IF and ID stages (the instructions are “on hold”, delayed with one clock cycle)

If (ID/EX.MemRead
and ((ID/EX.RegisterRt = IF/ID.RegisterRs)
or (ID/EX.RegisterRt = IF/ID.RegisterRt)))
→ stall the pipeline

(LW INSTRUCTION?)
(destination in EX= source in ID?)

Pipeline CPU Design – Hazards





Pipeline CPU Design – Hazards



- Data Hazard and Stalls
 - A NoOp (stall/bubble) implemented in hardware implies:
 - to block the PC and the pipeline registers (that are located before the current instruction, i.e. ID stage) in order to hold their values for one more clock cycle;
 - Force the control signals from the ID/EX register to 0 (**FLUSH**) in order to further insert the NoOp in the pipeline
 - Concrete:
 - We stall the pipeline by keeping an instruction in the ID stage: PCWrite = 0 and IF/IDWrite = 0, and
 - We insert a bubble into the pipeline by writing ‘0’ to EX, MEM and WB control fields of the ID/EX register (NoOp)
 - The ‘0’ control values are transmitted forward in the pipeline at every clock cycle
→ no writes to registers or memory, no jump or branch



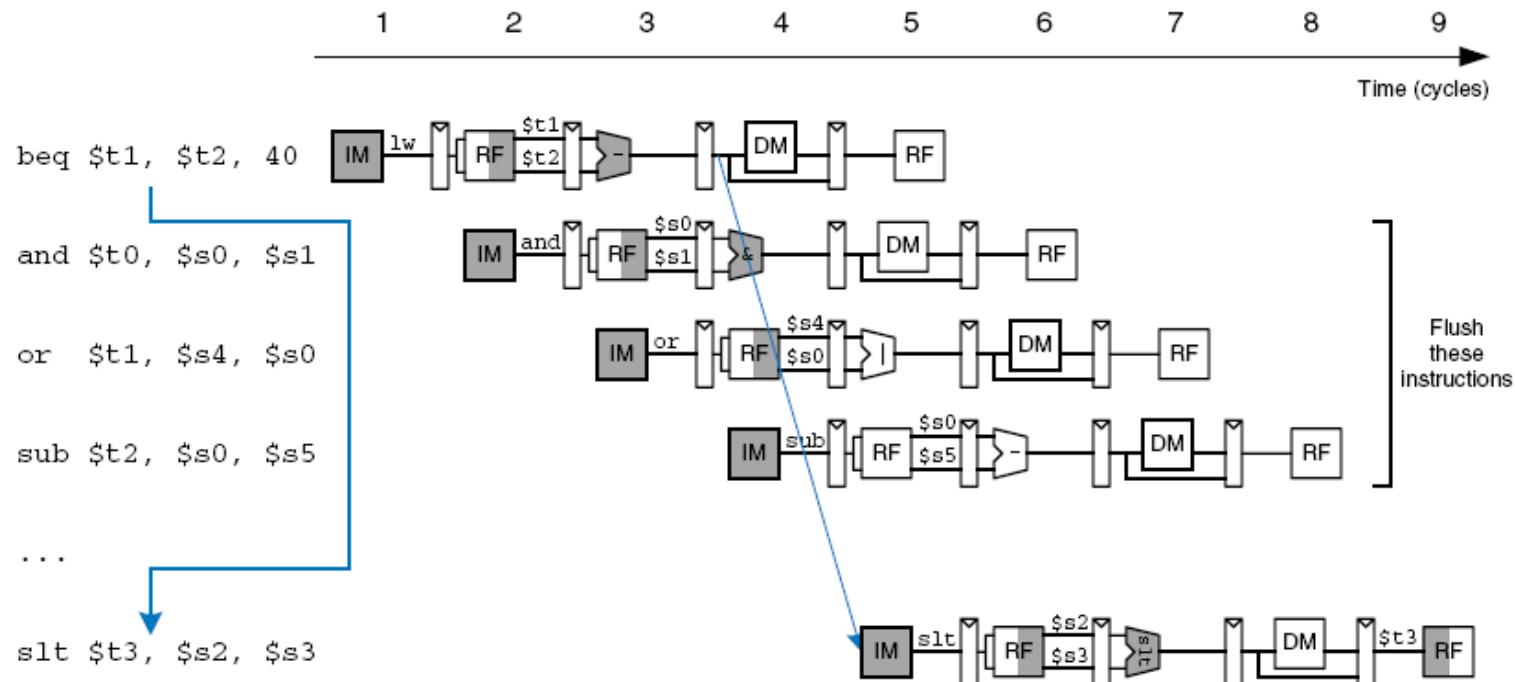
Load Data Hazard solutions



- HW Stalls to Resolve Data Hazard
 - “Interlock”: checks for hazard & stalls
- SW inserts independent instructions
 - Worst case – inserts NoOP instructions
 - MIPS I solution: No HW checking
- Compiler solutions to avoid Load Hazards
 - Compiler will detect data dependency and inserts NoOp instructions until data is available
 - Compiler will find independent instructions to fill in the *Load-delay slots*
 - Software Scheduling can help to Avoid Load Data Hazards
 - reordering of the instructions



- Control Hazards (Branch, Conditional Jumps)
 - Branch decision only in **MEM Stage**
 - The next 3 instructions after branch are already in IF, ID, EX stages
 - If branch is taken these instructions **must not write the results!**



BEQ condition is evaluated in cc4, if it is true:

- The new PC value will be written in cc5 → slt

What can be done to reduce the number of instructions?



Pipeline CPU Design – Hazards



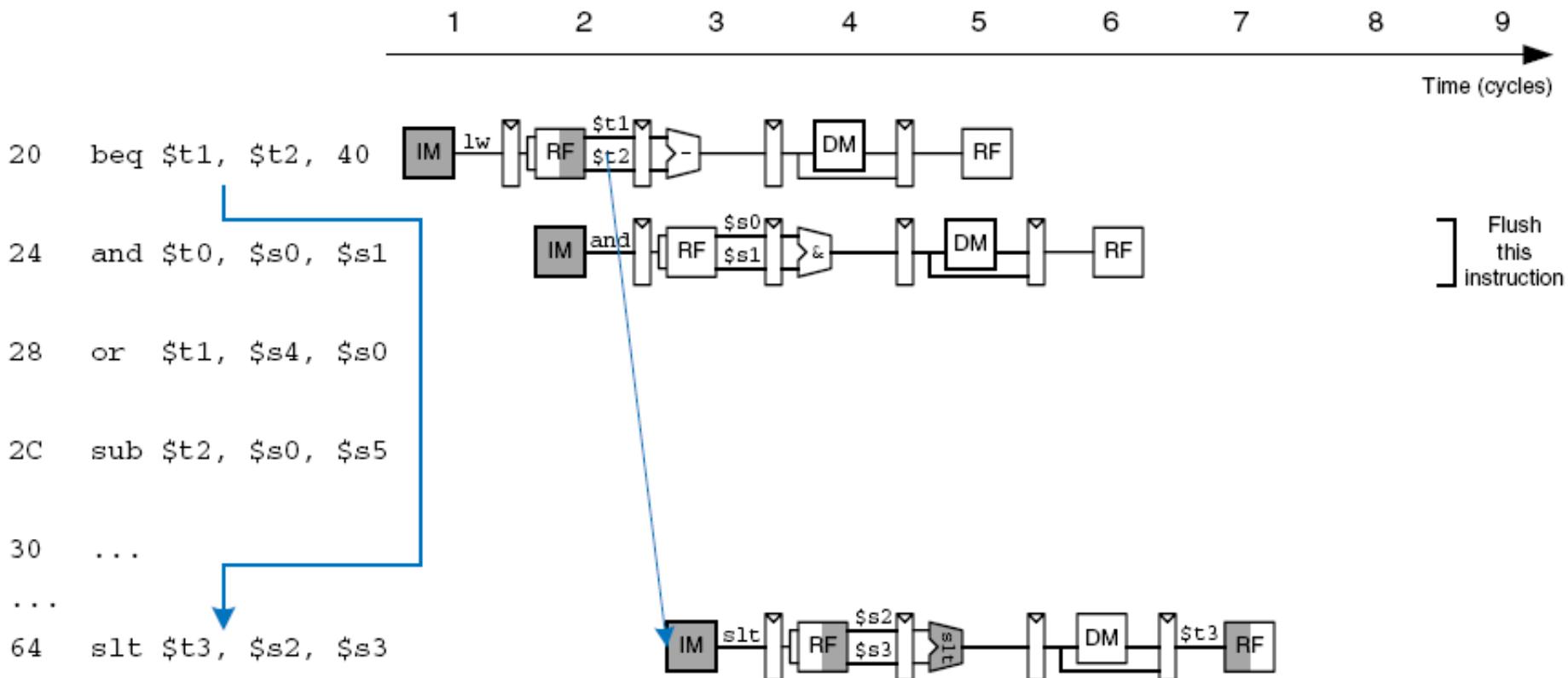
- Control Hazards (Branch, Conditional Jumps)
 - Branch instruction in pipeline
 - We are predicting “branch not taken” (execute the next instructions)
 - Need to add hardware for flushing instructions if we are wrong
 - Clever design techniques can reduce the delay to **ONE instruction**
 - Move the branch execution hardware earlier in the pipeline, fewer instructions need to be flushed
 - In the simple MIPS pipeline we have selected the next PC for a branch in MEM stage
 - We can move the Branch address calculation and condition detection in the ID stage → Only one instruction in the pipeline after a Branch
 - Branch condition test in ID implies additional forwarding and hazard detection, the branch can be dependent on a result still in pipeline
 - Forward data from EX/MEM or MEM/WB pipeline registers
 - Stall if a data hazard that can not be resolved by forwarding occurs (LW before the branch)
 - **Branch Delay now 1 clock cycle!**



Pipeline CPU Design – Hazards



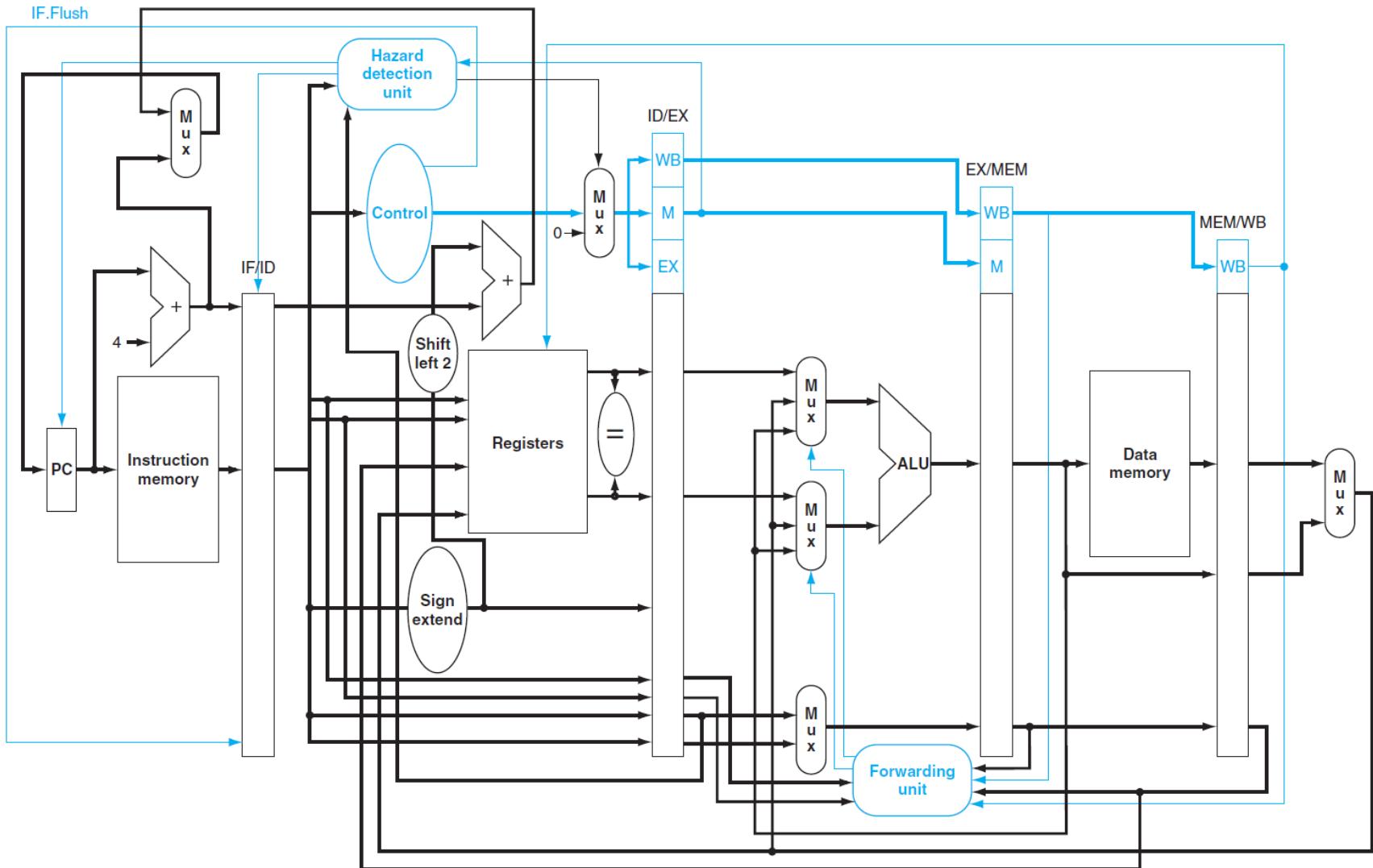
- Control Hazards (Branch, Conditional Jumps)



Branch resolved in ID stage, only one instruction enters the pipeline after a branch

We must add hardware to cancel (flush) the instruction that entered the pipeline in case the branch is taken!

Pipeline CPU Design – Hazards



MIPS Pipeline with Branch Logic resolved in ID Stage



Pipeline CPU Design – Hazards



- Handling Hazards (Summary)

- Flush

- Cancels an instruction by transforming it into a NoOp (at the hardware level) by annulling the next pipeline register
 - The canceling can be complete – reset the pipeline register, or minimal by setting to ‘0’ the Write control signals for the Register File and Data Memory
 - Flush for branch: canceling the next instruction after the branch – Clear the IF/ID register (NoOp)

- Stall – Bubble

- keeps the content of the pipeline registers for “younger” instructions (located before the current instruction) and wait for the “older” instructions (located after the current instruction) to finish
 - Block the writing in the previous pipeline registers
 - Transmit ‘0’ (NoOp) to the next pipeline stage where the depending instruction is located (flush)
 - Software example NoOp: sll \$0, \$0, 0 (bubble)



Pipeline CPU Design – Hazards



- Four Branch Hazard Alternatives
 - Stall until branch decision is clear
 - Predict Branch Not Taken
 - Execute successor instructions in sequence
 - FLUSH instructions in pipeline if branch actually taken
 - PC+4 already calculated, so use it to get next instruction
 - Must be careful not to alter state of registers until actual branch target is known
 - Only slightly more complicated than pipeline freeze to implement
 - Compiler can modify loops to favor branches not taken
 - Predict Branch Taken
 - Treat every branch as taken
 - Begin fetch at target when the branch is decoded and target address known
 - No advantage: because target address and branch outcome are in ID
 - Only makes sense for machines that compute target address before determining branch outcome
 - Delayed Branch
 - Define branch to take place AFTER the following instruction



Pipeline CPU Design – Hazards



- Branch delay of length n
 - 1 slot delay ($n = 1$) allows proper decision and branch target address in 5 stage pipeline (MIPS)
 - Define a “branch delay slot”
 - The next instruction after a branch is always executed
 - Rely on compiler to “fill” the slot with something useful
 - Worst case, Software inserts NoOp into branch delay slot
 - Static Branch Prediction Using Compiler Technology – Delayed branch
 - The instruction in the delay slot (there is only one delay slot) is executed
 - If the branch is not taken, execution continues with the instruction after the branch-delay instruction
 - If the branch is taken, execution continues at the branch target.
 - When the instruction in the branch delay slot is also a branch, the meaning is unclear.
 - Because of this confusion, architectures with delay branches often disallow putting a branch in the delay slot



Pipeline CPU Design – Hazards



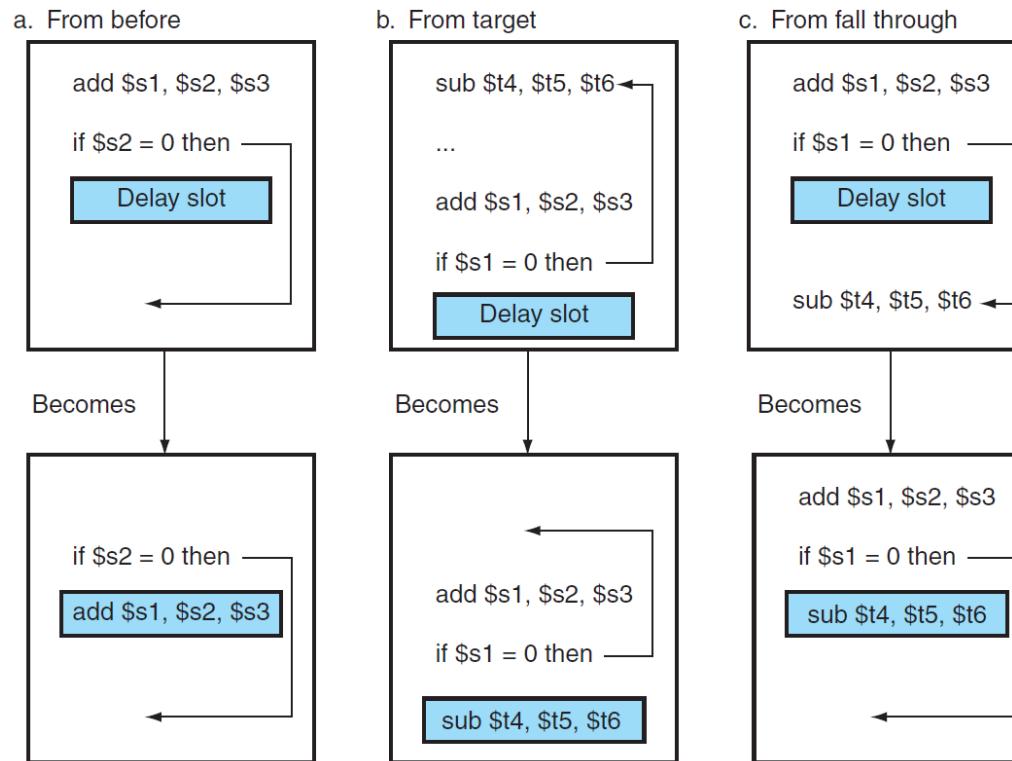
- **Delayed Branch implementation**
 - Fetch subsequent instruction independent on branch outcome
 - The **compiler** must fill the branch delay slot with a valid/useful instruction
 - Otherwise a NoOp is used
 - Where to find instructions to fill the branch delay slot?
 - Before branch instruction
 - From the target address: only valuable when branch taken
 - From fall through: only valuable when branch not taken

| Scheduling Strategy | Requirements | Improves Performance: When? |
|---------------------|--|---|
| From before | Branch must not depend on the rescheduled instructions | Always |
| From target | Must be ok to execute rescheduled instructions if branch is not taken. May need to duplicate instructions | When branch is taken. May enlarge programs if instructions are duplicated |
| From fall through | Must be ok to execute instructions if branch is taken | When branch is not taken. |

Delayed-branch scheduling schemes and their requirements



- Delayed Branch implementation



Static Prediction (Compiler) of conditional branches: Taken / Not Taken

- Improves strategy for placing instructions in delay slot
- Two strategies: Backward branch predict taken, forward branch not taken
- Profile-based prediction: record branch behavior, predict branch based on prior runs



Problems – Homework



- Implementing instructions in pipeline – same as previous lectures
- Draw the pipeline diagram with and without forwarding for the following code sequences:

| | | | |
|---|---|---|--|
| Iw \$6, 4000(\$7) add \$9, \$6, \$3 or \$2, \$9, \$6 lw \$2, 2000(\$2) add \$3, \$9, \$2 sw \$9, 2000(\$3) | loop: Iw \$6, 4000(\$7) add \$9, \$6, \$3 or \$5, \$9, \$6 lw \$2, 2000(\$5) add \$3, \$9, \$2 subi \$5, \$5, 12 sw \$9, 2000(\$3) bne \$9, \$0, loop | loop: lw R10, X(R20) lw R11, Y(R20) subu R10, R10, R11 sw Z(R20), R10 addiu R20, R20, 4 subu R5, R23, R20 bnez R5, LOOP nop; 1 delay slot | add \$r1, \$r5, \$r3 lw \$r2, 0(\$r1) add \$r1, \$r2, \$r3 sw \$r1, 0(\$r5) |
|---|---|---|--|

- The Branch can be resolved in ID or MEM stage
- Compute the number of clock cycles for the code sequence
- Consider 2 iterations for the loop examples



References



1. D. A. Patterson, J. L. Hennessy, “Computer Organization and Design: The Hardware/Software Interface”, 3rd edition, ed. Morgan–Kaufmann, 2005.
2. D. A. Patterson, J. L. Hennessy, “Computer Organization and Design: The Hardware/Software Interface”, 5th edition, ed. Morgan–Kaufmann, 2013.
3. MIPS32™ Architecture for Programmers, Volume I: “Introduction to the MIPS32™ Architecture”.
4. MIPS32™ Architecture for Programmers Volume II: “The MIPS32™ Instruction Set”.



Computer Architecture

Lecturer: Mihai Negru

2nd Year, Computer Science

Lecture 9: Advanced Pipelining

<http://users.utcluj.ro/~negrum/>



Data Hazard Classification



- Data hazards: depend on the order of **read** and **write** accesses to the registers from the Register File.
- Consider two instructions **i** and **j**, with **i** occurring before **j**.
- Possible data Hazards:
 - RAW – read after write
 - i: $R2 \leftarrow R1 + R3$
 - j: $R4 \leftarrow R2 + R3$
 - j tries to read R2 before i writes it, so j incorrectly gets the old value for R2 .
 - The most common type of hazard → forwarding to overcome it



Data Hazard Classification



- Data hazards:
 - WAW – write after write
 - i: $R2 \leftarrow R4 \times R7$
 - j: $R2 \leftarrow R1 + R3$
 - j tries to write $R2$ before it is written by i .
 - Writes in wrong order, leaving the value written by i rather than the value written by j in the destination register.
 - This hazard is present only in pipelines that write in more than one pipe stage (or allow an instruction to proceed even when a previous instruction is stalled).
 - Can't happen in MIPS 5 stage pipeline because:
 - All instructions take 5 stages, and
 - Writes are always in stage 5 (WB)

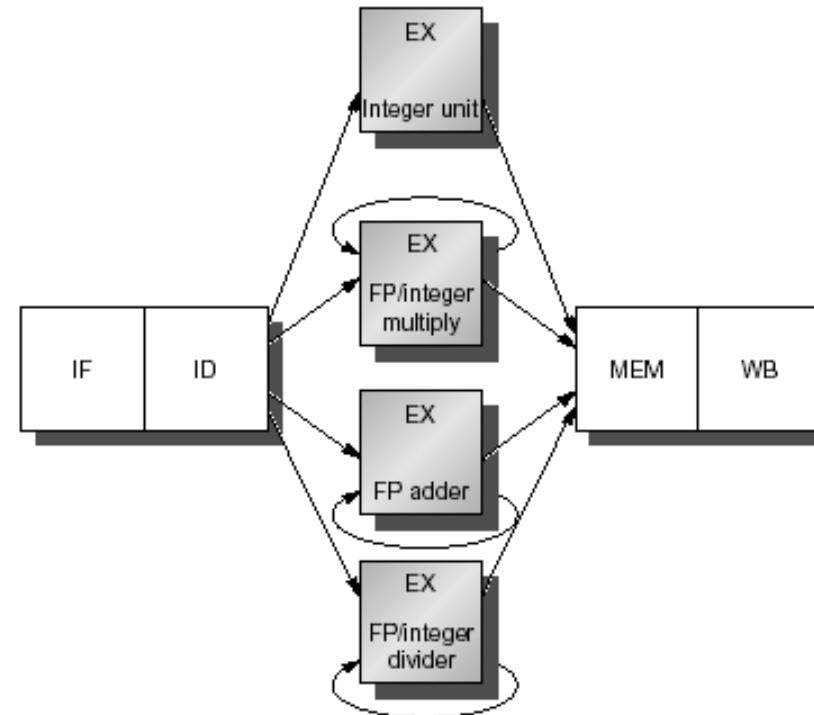


Data Hazard Classification



- Data hazards:
 - WAR – write after read
 - $i: R1 \leftarrow R2 + R3$
 - $j: R3 \leftarrow R4 + R5$
 - j tries to write a destination before it is read by i , so i incorrectly gets the new value – instruction i is stuck
 - Can't happen in MIPS 5 stage pipeline because:
 - All instructions take 5 stages
 - Reads are always in stage 2 (ID) and
 - Writes are always in stage 5 (WB)
 - This hazard occurs when there are some instructions that write results early in the instruction pipeline, and other instructions that read a source late in the pipeline.
 - RAR – read after read is not a hazard!

- Pipeline MIPS with variable length multi-cycle operations
 - Extended MIPS pipeline to handle Floating Point (FP) operations
 - The FP operations can have different latencies for variable length operations
 - Latency – number of clock cycles necessary to execute the operation



MIPS Pipeline with 4 Functional Units: 1 – INT, 3 – FP



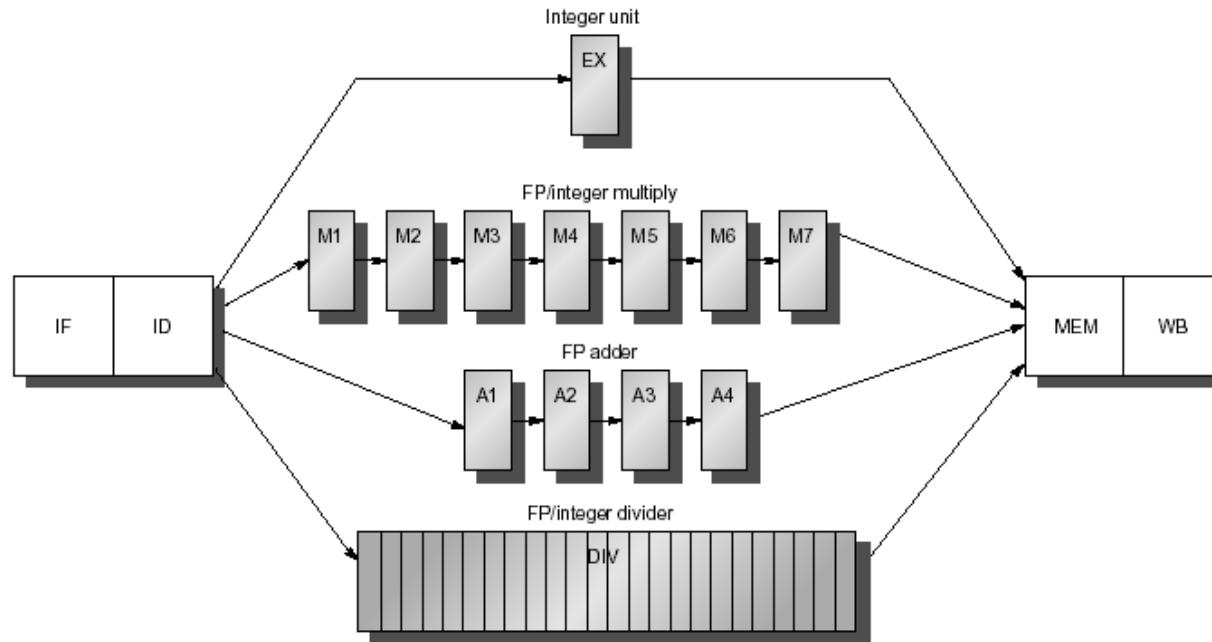
Advanced Pipelining



- EX Integer unit handles loads and stores, integer ALU operations, and branches;
- EX FP/Integer multiply, FP adder, FP/Integer divider.
- Only one instruction could be issued in a clock cycle,
- All instructions go through the standard pipeline stages:
 - IF, ID, EX, MEM, WB.
- The FP operations are executed in several clock cycles in EX.
- After EX stage → MEM and WB to complete execution.
- If the FP units are not pipelined
 - No new instruction may issue until the previous one leaves the EX stage.
- If an instruction cannot proceed to the EX stage, the entire pipeline behind that instruction will be stalled.



Advanced Pipelining



Pipeline in pipeline: A pipeline that supports multiple outstanding FP operations

- **Latency:** the number of intervening cycles between an instruction that produces a result and an instruction that uses the result.
- **Initiation or repeat interval:** the number of cycles that must elapse between issuing two operations of a given type

The latency of the FP operations increases the frequency of RAW hazards and stalls!



Advanced Pipelining



| Functional unit | Latency | Initiation interval |
|-------------------------------------|---------|---------------------|
| Integer ALU | 0 | 1 |
| Data memory (integer and FP loads) | 1 | 1 |
| FP add | 3 | 1 |
| FP multiply (also integer multiply) | 6 | 1 |
| FP divide (also integer divide) | 24 | 25 |

- The FP **multiplier** and **adder** are fully pipelined: **7** and **4** stages, respectively.
- The FP **divider** is not pipelined; it requires **24** clock cycles to complete.
- The latency in instructions between the issue of a FP operation and the use of the result of that operation without incurring a RAW stall is determined by the number of cycles spent in the execution stages.
- Example: the fourth instruction after a FP add can use the result of the FP add.
- For integer ALU operations, the depth of the execution pipeline is always one and the next instruction can use the results.
- Both FP loads and Integer loads complete during MEM, which means that the memory system must provide either 32 or 64 bits in a single clock cycle.



Advanced Pipelining

| | | | | | | | | | | | |
|------|----|----|-----------|-----------|-----------|------------|-----------|-----|-----------|-----|----|
| MULD | IF | ID | M1 | M2 | M3 | M4 | M5 | M6 | M7 | MEM | WB |
| ADDD | | IF | ID | A1 | A2 | A3 | A4 | MEM | WB | | |
| LD | | | IF | ID | EX | MEM | WB | | | | |
| SD | | | | IF | ID | EX | MEM | WB | | | |

The pipeline timing diagram for a set of independent FP operations.

- The stages where **data is needed**
- The stages where **a result is available**.
- FP loads and stores use a 64-bit path to memory, so the pipelining timing is just like an integer load or store.
- The structure of the pipeline requires the introduction of additional pipeline registers (e.g., A1/A2, A2/A3, A3/A4).
- The ID/EX register must be expanded to connect ID to EX, DIV, M1, and A1
- The forwarding is similar – check if the destination register in any of EX/MEM, A4/MEM, M7/MEM, D/MEM, or MEM/WB registers is one of the source registers of a FP instruction.
 - If so, the appropriate input multiplexer will have to be enabled so as to choose the forwarded data.



Advanced Pipelining – Problems



- The division unit is not fully pipelined, structural hazards can occur.
 - Should be detected and issuing instructions will be stalled.
- The instructions have varying running times, the number of register writes in a cycle can be larger than 1.
- WAW hazards are possible, since instructions no longer reach WB in order.
- WAR hazards are not possible, the register reads always occur in ID.
- Instructions can complete in a different order than they were issued, causing problems with exceptions.
- Because of longer latency of operations, stalls for RAW hazards will be more frequent.



Advanced Pipelining

| Instruction | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 |
|--------------------|----|----|----|-------|----|-------|-------|-------|-------|-------|-------|-----|----|-------|-------|-------|-----|
| LD F4, 0 (R2) | IF | ID | EX | MEM | WB | | | | | | | | | | | | |
| MULD F0, F4, F6 | | IF | ID | stall | M1 | M2 | M3 | M4 | M5 | M6 | M7 | MEM | WB | | | | |
| ADDD F2, F0, F8 | | | IF | stall | ID | stall | stall | stall | stall | stall | stall | A1 | A2 | A3 | A4 | MEM | WB |
| SD 0 (R2), F2 | | | | | IF | stall | stall | stall | stall | stall | stall | ID | EX | stall | stall | stall | MEM |

- Each instruction in this sequence is dependent on the previous and proceeds as soon as data is available
 - assumes the pipeline has full forwarding.
- The SD must be stalled an extra cycle so that its MEM does not conflict with the ADDD.
 - Extra hardware can handle this case (write buffer).



Dynamic Scheduling – Scoreboard Method



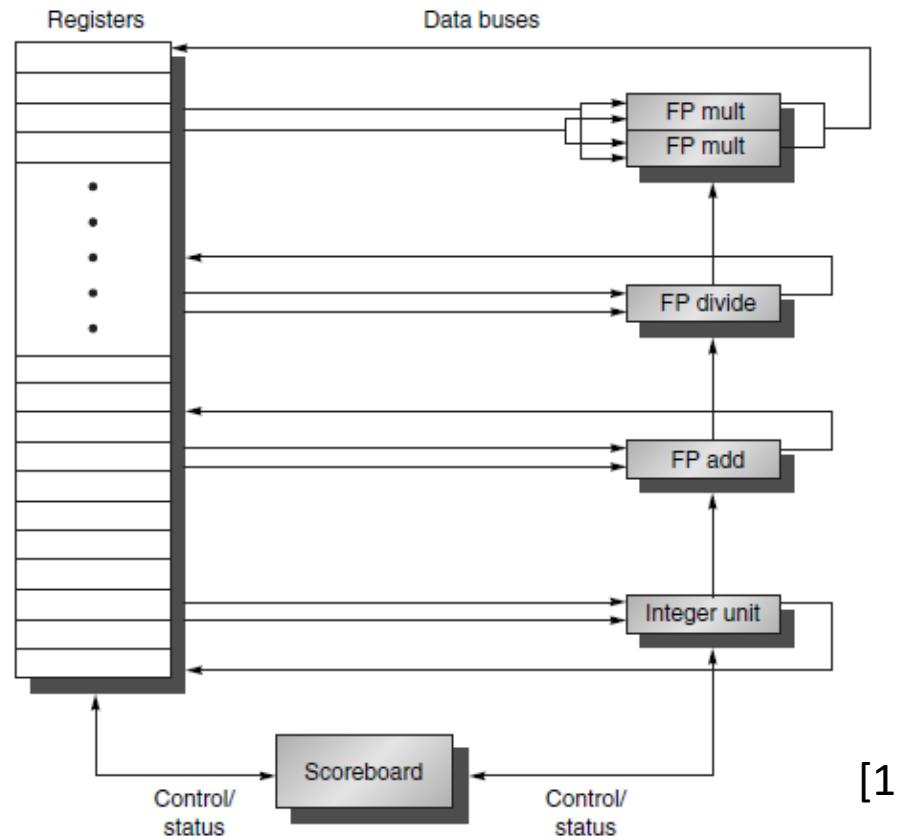
- Hardware scheme – instruction parallelism
 - Why in HW at run time?
 - Works when one does not know real data dependence at compile time
 - Code is not machine dependent
 - Simpler compiler
 - Key idea: Allow instructions behind stall to proceed.

DIVD F0, F2, F4

ADDD F10, F0, F8

SUBD F12, F8, F14

- Enables out-of-order execution → out-of-order completion
- Split ID stage to check both for structural & data dependencies
- Scoreboard realized in 1963 for CDC 6600 (First supercomputer)
- No forwarding!
- Instructions execute when they are not dependent on previous instructions and there are no hazards.



The basic structure of a MIPS processor with a Scoreboard

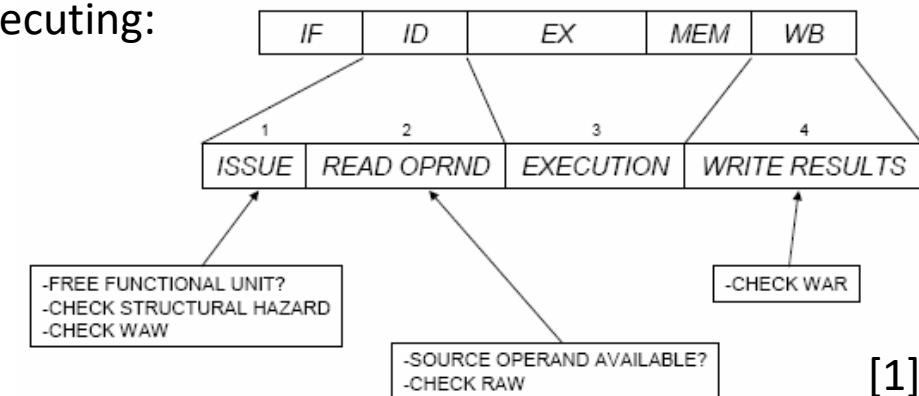
- The scoreboard's function is to control instruction execution (vertical control lines).
- All data flows between the RF and the FUs over the buses (the horizontal lines).
- 2 FP multipliers, 1 FP divider, 1 FP adder, and 1 integer unit.
- 2 inputs busses and 1 output bus serve a group of FUs.



Dynamic Scheduling – Scoreboard Method



- Every instruction goes through the scoreboard, where a record of the data dependences is constructed
 - this step corresponds to instruction issue and replaces part of the ID step in the MIPS pipeline.
- The scoreboard then determines when the instruction can read its operands and begin execution.
- If the scoreboard decides the instruction cannot execute immediately, it monitors every change in the hardware and decides when the instruction can execute.
- The scoreboard also controls when an instruction can write its result into the destination register.
- Thus, all hazard detection and resolution is centralized in the scoreboard.
- Each instruction undergoes four steps in executing:
 - Issue
 - Read operands
 - Execution
 - Write results



[1]



Dynamic Scheduling – Scoreboard Method



- Scoreboard stages:
 - Issue – decode instructions & check for structural & WAW hazards (ID1)
 - If a FU for the current instruction is **free** and no other active instruction has the same destination register (**WAW**) → The scoreboard **issues** the instruction to the FU and updates its internal data structure.
 - If a **structural or WAW hazard exists**, then the instruction issue **stalls**, and no further instructions will issue until these hazards are cleared
 - Read operands – wait until no **RAW data hazards**, then read operands (ID2)
 - A source operand is available if
 - No earlier issued active instruction is going to write it, or if
 - The register containing the operand is being written by a currently active FU.
 - Registers are only read when they are both available.
 - When the source operands are available, the scoreboard tells the FU to proceed to read the operands from the registers and begin execution.
 - Because the operands are read only when both are available in the Register File, **the scoreboard does not take advantage of forwarding!**
 - The scoreboard resolves RAW hazards dynamically in this step, and instructions may be sent into execution **out of order**



Dynamic Scheduling – Scoreboard Method



- Scoreboard stages:
 - Execute – operations with the operands (EX)
 - The FU begins execution upon receiving the operands.
 - When the result is ready, it notifies the scoreboard that it has completed execution.
 - Write result – finish execution (WB)
 - Once the scoreboard is aware that the FU has completed execution, the scoreboard checks for WAR hazards.
 - If none, it writes results.
 - If WAR, then it stalls the instruction
 - Example
 - DIVD F0, F2, F4
 - ADDD F10, F0, **F8**
 - SUBD **F8**, F8, F14
- CDC 6600 scoreboard would stall SUBD until ADDD reads operands
- **Scoreboard → In-order issue – out-of-order execute & commit**



Dynamic Scheduling – Scoreboard Method



- Applying the Scoreboard method
 - Timing: Latency (clock cycles) FP Addition = 2, Multiplication = 10, Division = 40, Read Op = 1, Execute Load = 1
 - In order Issue
 - Stage and Hazard test
 - Issue (Iss): Structural, WAW
 - Read operand (Rop): RAW
 - Write result (Wr): WAR
 - For the given instruction sequence:
 - Complete the following table

| # | Instruction | Str | WAW | Iss | RAW | Rop | stE | endE | WAR | Wr |
|---|-----------------------|-----|-----|-----|-----|-----|-----|------|-----|----|
| 1 | LD F6 34+R2 | | | | | | | | | |
| 2 | LD F2 45+R3 | | | | | | | | | |
| 3 | MULD F0 F2 F4 | | | | | | | | | |
| 4 | SUBD F8 F6 F2 | | | | | | | | | |
| 5 | DIVD F10 F0 F6 | | | | | | | | | |
| 6 | ADDD F6 F8 F2 | | | | | | | | | |



Dynamic Scheduling – Scoreboard Method



- Applying the Scoreboard method
 - stE: start Execution
 - endE: end Execution
 - Str (structural), WAW, RAW, WAR: hazard type
 - the number in hazard column shows the related instruction's number
 - The number in the Iss, Rop, stE, endE and Wr columns – clock cycle number

| # | Instruction | Str | WAW | Iss | RAW | Rop | stE | endE | WAR | Wr |
|---|-----------------------|-----|-----|-----|-----|-----|-----|------|-----|----|
| 1 | LD F6 34+R2 | | | 1 | | 2 | 3 | 3 | | 4 |
| 2 | LD F2 45+R3 | 1 | | 5 | | 6 | 7 | 7 | | 8 |
| 3 | MULD F0 F2 F4 | | | 6 | 2 | 9 | 10 | 19 | | 20 |
| 4 | SUBD F8 F6 F2 | | | 7 | 2 | 9 | 10 | 11 | | 12 |
| 5 | DIVD F10 F0 F6 | | | 8 | 3 | 21 | 22 | 61 | | 62 |
| 6 | ADDD F6 F8 F2 | 4 | | 13 | | 14 | 15 | 16 | 5 | 22 |



Dynamic Scheduling – Scoreboard Method



- **Scoreboard limits:**
 - A scoreboard uses the available ILP to minimize the number of stalls arising from the program's true data dependences.
 - In eliminating stalls, a scoreboard is limited by several factors:
 - The amount of parallelism available among the instructions
 - If each instruction depends on its predecessor, no dynamic scheduling scheme can reduce stalls
 - The number of scoreboard entries
 - Determines how far ahead the pipeline can look for independent instructions. The set of instructions examined as candidates for potential execution is called the **window**. The size of the scoreboard determines the size of the window. We assume a window does not extend beyond a branch
→ only straight-line code from a single basic block is inside the scoreboard
 - The number and types of functional units
 - Determines the importance of structural hazards, which can increase when dynamic scheduling is used
 - The presence of anti-dependences and output dependences
 - These lead to WAR and WAW stalls
 - No forwarding hardware

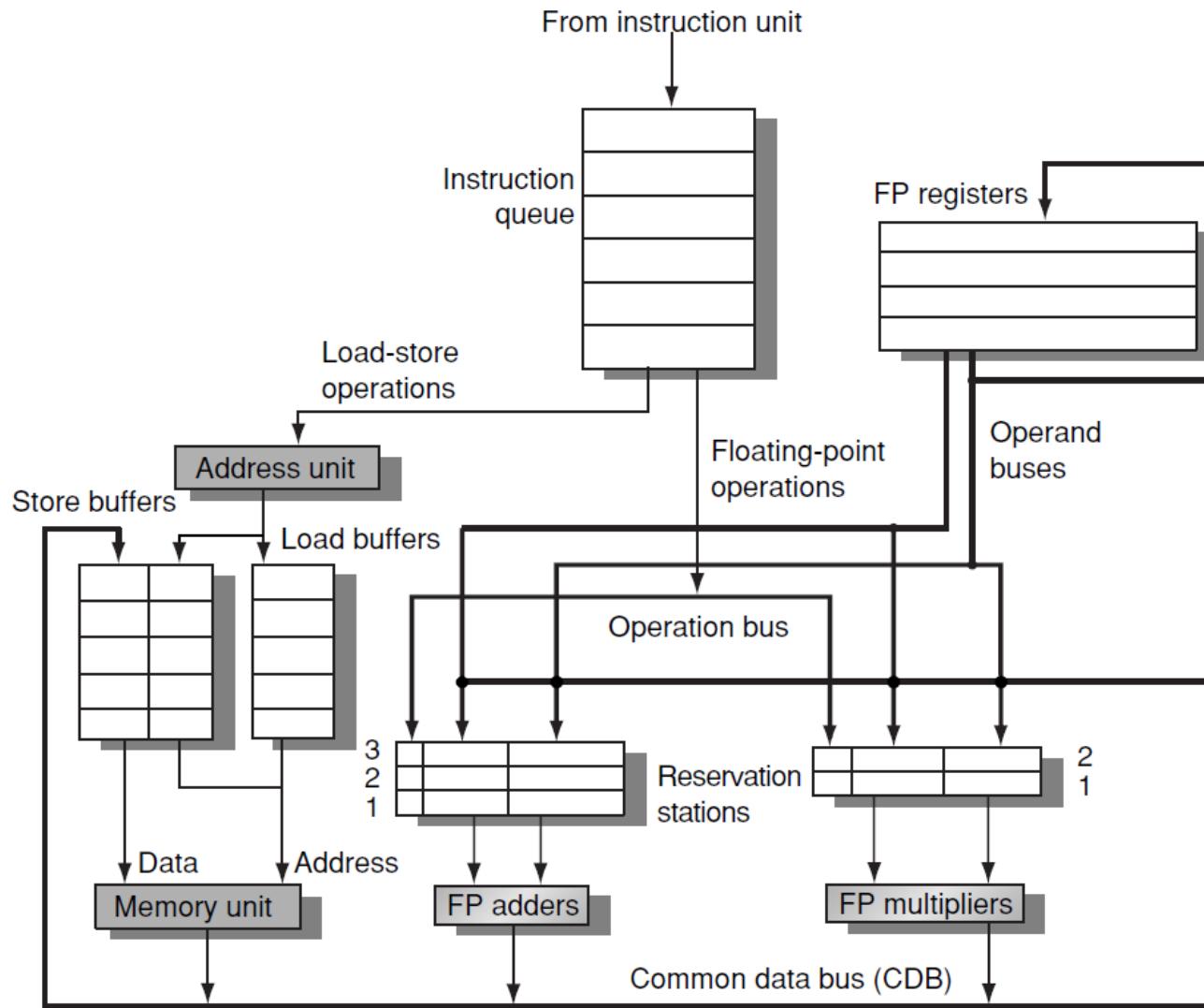


Dynamic Scheduling – Tomasulo Method



- Tomasulo algorithm
 - for IBM 360/91 about 3 years after CDC 6600 (1966)
 - Goal: High Performance dynamic scheduling of the execution without special compilers
 - Differences between IBM 360 & CDC 6600 ISA
 - IBM has only 2 register specifiers / instr. vs. 3 in CDC 6600
 - IBM has 4 FP registers vs. 8 in CDC 6600
 - Why Study? Implemented in
 - Alpha 21264
 - HP 8000
 - MIPS 10000
 - Pentium II
 - PowerPC 604
 - ...

Dynamic Scheduling – Tomasulo Method



[1]

The basic structure of MIPS FP Unit using Tomasulo's algorithm



Dynamic Scheduling – Tomasulo Method



- Tomasulo vs. Scoreboard
 - Control & buffers distributed with Function Units (FU) vs. centralized in scoreboard
 - FU buffers called “Reservation Stations” (RS); buffer the operands of instructions waiting to issue
 - RS fetches and buffers an operand as soon as it is available, eliminating the need to get the operand from the Register File
 - Pending instructions designate the RS that will provide their input
 - When successive writes to a register appear, only the last one is actually used to update the register
 - Registers in instructions replaced by values or pointers to RS; called **register renaming**
 - Avoids WAR, WAW hazards
 - More reservation stations than registers so can do optimizations that compilers cannot.
 - Results are broadcasted over **Common Data Bus (CDB)**
 - Load and Stores treated as FUs with RSs as well
 - Common data bus: data + source
 - 64 bits of data + 4 bits of Functional Unit source address
 - Write if matches → expected Functional Unit produces result
 - Does the broadcast



Dynamic Scheduling – Tomasulo Method



- Floating-point operations are sent from the instruction fetch unit into a queue.
- The reservation stations include the operation and the actual operands, as well as information used for detecting and resolving hazards.
- Load buffers to hold the addresses for memory reads that are waiting for the CDB.
- Similarly, store buffers are used to hold the destination memory addresses of stores waiting for their operands.
- All results from either the FP units or the load unit are put on the CDB, which goes to the FP register file as well as to the reservation stations and store buffers.
- The FP adders implement addition and subtraction, while the FP multipliers do multiplication and division



Dynamic Scheduling – Tomasulo Method



- Tomasulo stages:
 - Issue – get an instruction from the floating-point operation queue.
 - If the operation is a floating-point operation, issue if there is an empty reservation station, and send the operands to the reservation station if they are in the registers.
 - If the operation is a load or store, it can issue if there is an available buffer.
 - If there is no empty reservation station or an empty buffer, then there is a structural hazard and the instruction stalls until a station or buffer is freed.
 - This step also performs the process of renaming registers.
 - Execute – if one or more operands is not yet available, monitor the CDB
 - When an operand becomes available, it is placed into the corresponding reservation station.
 - When both operands are available, execution starts.
 - This step checks for RAW hazards
 - Write result – finish execution (WB)
 - When the result is available, write it on the CDB and from there into: registers, reservation stations waiting for the result, and to any waiting store buffers.
 - Mark the used reservation station as available



Dynamic Scheduling – Tomasulo Method



- The major advantages of the Tomasulo scheme are:
 - The distribution of the hazard detection logic
 - The elimination of stalls for WAW and WAR hazards.
 - The reduction in structural hazards – more reservation stations than functional units
 - CDB conflict may appear
 - The first advantage arises from the distributed reservation stations and the use of the CDB.
 - If multiple instructions are waiting on a single result, and each instruction already has its other operand, then the instructions can be released simultaneously by the broadcast on the CDB.
 - In Scoreboard the results are written in registers, and then read by the waiting instructions
 - WAW and WAR hazards are eliminated by renaming registers using the reservation stations, and by storing operands into the reservation station as soon as they are available.



Dynamic Scheduling – Tomasulo Method



- Applying the Tomasulo Method
 - Timing: Latency (clock cycles) FP Addition = 2, Multiplication = 10, Division = 40, Read Op = 1, Execute Load = 1
 - In order Issue
 - Stage and Hazard test
 - Issue (Iss): Structural
 - Execute: RAW
 - Write result (Wr): CDB conflict
 - For the given instruction sequence:
 - Complete the following table

| # | Instruction | Str | Iss | RAW | stE | endE | CDB | Wr |
|---|-----------------------|-----|-----|-----|-----|------|-----|----|
| 1 | LD F6 34+ R2 | | | | | | | |
| 2 | LD F2 45+ R3 | | | | | | | |
| 3 | MULD F0 F2 F4 | | | | | | | |
| 4 | SUBD F8 F6 F2 | | | | | | | |
| 5 | DIVD F10 F0 F6 | | | | | | | |
| 6 | ADDD F6 F8 F2 | | | | | | | |



Dynamic Scheduling – Tomasulo Method



- Applying the Tomasulo Method
 - stE: start Execution
 - endE: end Execution
 - Str (structural), RAW: hazard type
 - CDB – common data bus conflict
 - the number in hazard column shows the related instruction's number
 - The number in the Iss, stE, endE and Wr columns – clock cycle number

| # | Instruction | Str | Iss | RAW | stE | endE | CDB | Wr |
|---|----------------|-----|-----|-----|-----|------|-----|----|
| 1 | LD F6 34+ R2 | | 1 | | 2 | 3 | | 4 |
| 2 | LD F2 45+ R3 | | 2 | | 3 | 4 | | 5 |
| 3 | MULD F0 F2 F4 | | 3 | 2 | 6 | 15 | | 16 |
| 4 | SUBD F8 F6 F2 | | 4 | 2 | 6 | 7 | | 8 |
| 5 | DIVD F10 F0 F6 | | 5 | 3 | 17 | 56 | | 57 |
| 6 | ADDD F6 F8 F2 | | 6 | 4 | 9 | 10 | | 11 |



Dynamic Scheduling – Tomasulo Method



- Tomasulo Drawbacks
 - Complexity
 - Many associative stores (CDB) at high speed
 - Performance limited by Common Data Bus
 - Multiple CDBs → more FU logic for parallel stores
- Load/store disambiguation
 - The store buffers memorize the addresses and data such that the CPU does not wait for memory writes
 - The load and store can safely be done in a different order, provided the load and store access different addresses.
 - For every load the store addresses from the store buffers are examined
 - If a store address matches the load address, we must stop and wait until the store buffer gets a value; we can then access it or get the value from memory.



Dynamic Scheduling – Tomasulo Method



- **Tomasulo Summary**
 - Reservations stations: renaming to larger set of registers + buffering source operands
 - Prevents number of registers as bottleneck
 - Avoids WAR, WAW hazards of Scoreboard
 - Allows loop unrolling in HW
 - Contributions employed in modern processors
 - Dynamic scheduling
 - Register renaming
 - Load/store disambiguation
 - Tomasulo's scheme is appealing if one needs to pipeline an architecture for which it is difficult to schedule code or that has a shortage of registers
 - The adv. of the Tomasulo approach vs. compiler scheduling for an efficient single-issue pipeline are probably fewer than the costs of implementation.
 - But, due to the simultaneous issuing of more instructions (**superscalar**) and the necessity to improve the performance of difficult-to schedule code (static), **register renaming and dynamic scheduling are applied in modern processors.**



ILP: Loop Unrolling and Software Pipelining



- ILP: Overlap execution of unrelated instructions
 - One opportunity – loop level parallelism
 - The simplest and most common way to increase the amount of parallelism available among instructions is to exploit parallelism among iterations of a loop.
 - This type of parallelism is often called *loop-level parallelism*
- Basic Pipeline Scheduling and Loop Unrolling
 - Suppose standard MIPS pipeline with branch resolved in ID stage
 - No structural hazard, at each clock cycle a new instruction is issued

| Instruction producing result | Instruction using result | Latency in clock cycles |
|------------------------------|--------------------------|-------------------------|
| FP ALU op | Another FP ALU op | 3 |
| FP ALU op | Store double | 2 |
| Load double | FP ALU op | 1 |
| Load double | Store double | 0 |
| Integer op | Integer op | 0 |

Latency column shows the number of clock cycles needed to avoid a stall



ILP: Loop Unrolling and Software Pipelining



- Example – add the same value to the elements of an array in memory

Loop:

| | |
|-----------------|-----------------------------------|
| LD F0, 0(R1) | F0 = vector element |
| ADDD F4, F0, F2 | add scalar in F2 |
| SD 0(R1), F4 | store result |
| SUBI R1, R1, #8 | decrement pointer 8 bytes (DWord) |
| BNEZ R1, Loop | branch R1 != zero |
| NOP | branch delay slot |

FP Loop: Where are the Hazards?

- Without any scheduling the loop will execute as follows:

Loop:

| | | |
|---|-----------------|--------------------------------|
| 1 | LD F0, 0(R1) | F0 = vector element |
| 2 | stall | |
| 3 | ADDD F4, F0, F2 | add scalar in F2 |
| 4 | stall | |
| 5 | stall | |
| 6 | SD 0(R1), F4 | store result |
| 7 | SUBI R1, R1, #8 | decrement pointer 8 bytes (DW) |
| 8 | BNEZ R1, Loop | branch R1 != zero |
| 9 | stall | branch delay slot |

FP loop with stalls

- 9 clock cycles
- Rewrite the code to minimize the stalls



ILP: Loop Unrolling and Software Pipelining



Loop:

```
1 LD   F0, 0(R1)
2 stall
3 ADDD F4, F0, F2
4 SUBI R1, R1, #8
5 BNEZ R1, Loop    delayed branch
6 SD   8(R1), F4  why 8(R1): to compensate the effect of SUBI
                           when accessing the memory location
```

- Replace BNEZ stall with SD and change address of SD
- 6 clocks: Unroll loop 4 times code to make faster?



ILP: Loop Unrolling and Software Pipelining



Unroll loop 4 times

Loop:

| | | | |
|----|------|--------------|---------------------------|
| 1 | LD | F0, O(R1) | after LD 1 cycle stall |
| 2 | ADDD | F4, F0, F2 | after ADDD 2 cycles stall |
| 3 | SD | O(R1), F4 | drop SUBI & BNEZ |
| 4 | LD | F6, -8(R1) | |
| 5 | ADDD | F8, F6, F2 | |
| 6 | SD | -8(R1), F8 | drop SUBI & BNEZ |
| 7 | LD | F10, -16(R1) | |
| 8 | ADDD | F12, F10, F2 | |
| 9 | SD | -16(R1), F12 | drop SUBI & BNEZ |
| 10 | LD | F14, -24(R1) | |
| 11 | ADDD | F16, F14, F2 | |
| 12 | SD | -24(R1), F16 | |
| 13 | SUBI | R1, R1, #32 | alter to 4*8 |
| 14 | BNEZ | R1, LOOP | |
| 15 | NOP | | |

- Stalls:
 - LD – 1 stall
 - ADDD – 2 stalls
- $15 + 4 \times (1+2) = 27$ clock cycles, or 6.8 per iteration
- Assumes R1 is multiple of 4
- How many registers do we need? 3 vs. 9
- This unrolled version is slower than the scheduled version of the original loop.



ILP: Loop Unrolling and Software Pipelining



Unrolled loop that minimizes stalls

Loop:

```
1 LD    F0, 0(R1)
2 LD    F6, -8(R1)
3 LD    F10, -16(R1)
4 LD    F14, -24(R1)
5 ADDD F4, F0, F2
6 ADDD F8, F6, F2
7 ADDD F12, F10, F2
8 ADDD F16, F14, F2
9 SD    0(R1), F4
10 SD   -8(R1), F8
11 SD   -16(R1), F12
12 SUBI R1, R1, #32
13 BNEZ R1, LOOP
14 SD    8(R1), F16      8-32 = -24 the SUBI effect
```

- What assumptions made when code moved?
- OK to move store past SUBI even though changes register
- OK to move loads before stores: get right data?
- 14 clock cycles, or 3.5 per iteration.
- Using new registers is equivalent to register renaming (done by programmer or compiler)



Problems – Homework



- A MIPS machine has the following resources:

| Functional Unit | Nr. | Clock Cycles in EX Stage |
|-----------------|-----|--------------------------|
| Integer Unit | 1 | 1 |
| FADD Unit | 2 | 3 |
| FMUL Unit | 1 | 10 |
| FDIV Unit | 1 | 40 |

- Show the execution of the following instructions with
 - Scoreboard Method
 - Tomasulo Method
 - Draw the table for each method
 - Highlight the possible hazards and how they are resolved

| # | Instruction |
|---|----------------|
| 1 | ADDD F6,F8,F2 |
| 2 | SD F6, 100(R3) |
| 3 | MULD F6,F6,F2 |
| 4 | ADDD F6,F6,F4 |
| 5 | DIVD F8,F1,F6 |
| 6 | ADDD F6,F8,F2 |
| 7 | ADDD F1,F3,F2 |



References

1. D. A. Patterson and J. L. Hennessy, “Computer Organization and Design: A Quantitative Approach”, 5th edition, ed. Morgan-Kaufmann, 2011
2. D. A. Patterson, J. L. Hennessy, “Computer Organization and Design: The Hardware/Software Interface”, 5th edition, ed. Morgan–Kaufmann, 2013.
3. MIPS32™ Architecture for Programmers, Volume I: “Introduction to the MIPS32™ Architecture”.
4. MIPS32™ Architecture for Programmers Volume II: “The MIPS32™ Instruction Set”.



Computer Architecture

Lecturer: Mihai Negru

2nd Year, Computer Science

Lecture 10: Advanced Pipelining 2

<http://users.utcluj.ro/~negrum/>



More ILP → Speculation

- Speculation vs. Prediction
 - Prediction
 - Refers to Instruction Fetch → Branch prediction → issue rate
 - Must be de-coupled from the decision to execute the fetched instructions
 - Prediction can increase the issue rate but not the completion rate of the instructions
 - The completion rate has to be increased to keep up with the issue rate
 - Speculation
 - Refers to the execution of predicted instructions before it is known if it is safe to do so
 - Helps enhance the executable ILP
 - Relies on branch prediction
 - Key idea: separate instruction **execution** from instruction **commitment**
 - Compute on a temporary basis until speculation (prediction) outcome is determined



Hardware Based Speculation



- 3 components:
 - Dynamic scheduling
 - out-of-order data flow execution model
 - operations execute as soon as their operands are available
 - Dynamic branch prediction
 - fetch instructions to be speculatively executed
 - Speculation
 - execution of instructions before control dependences are resolved and
 - the ability to undo the effects of incorrectly speculated sequence

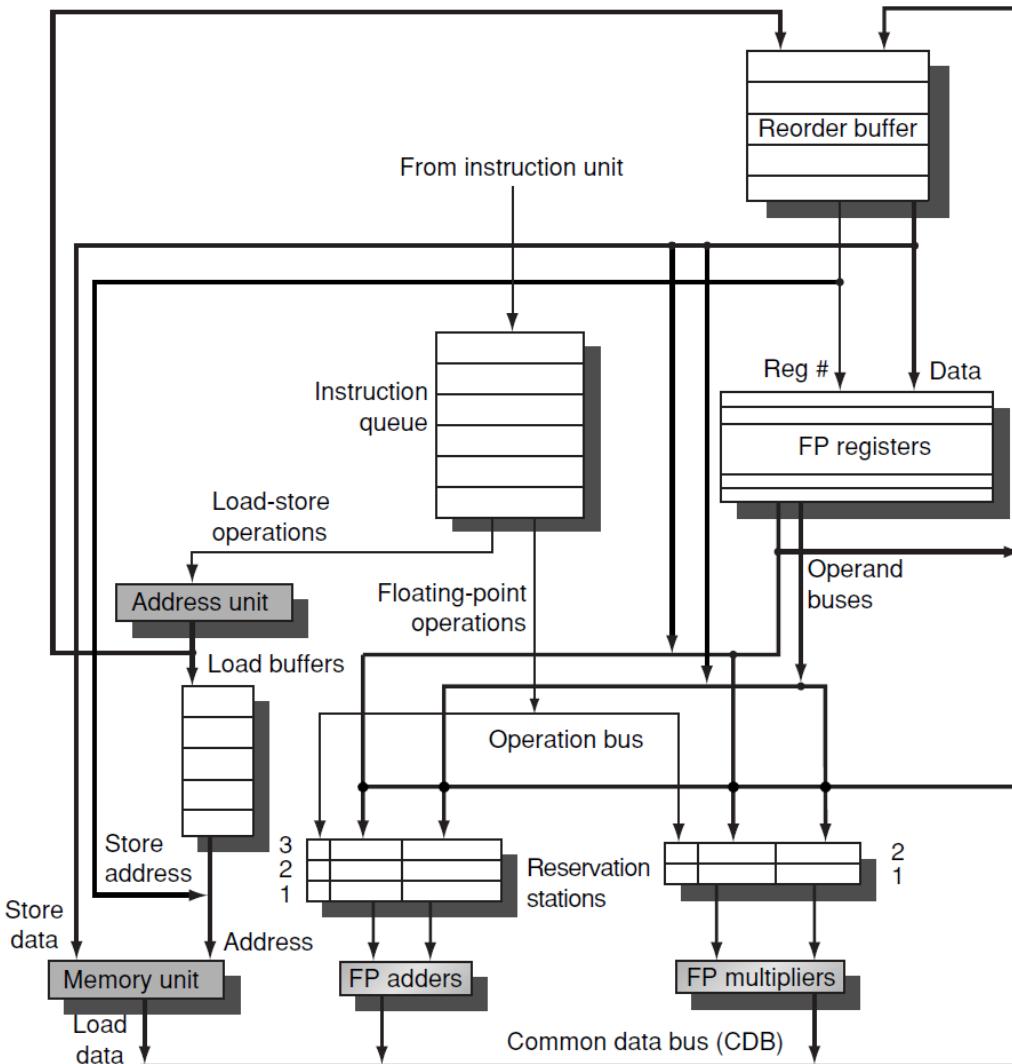


Speculative Tomasulo



- Some processors that implement speculative execution based on Tomasulo's algorithm:
 - PowerPC 603/604/G3/G4, MIPS R10000/R12000, Intel Pentium II / III / IV, Alpha 21264, and AMD K5/K6/Athlon
- Additional step – instruction commit
 - When an instruction is no longer speculative, allow it to update the Register File or Memory **in the order of the program**
- Requires an additional set of buffers:
 - To hold the results of instructions that have finished execution but have not committed
 - To reorder the instructions that complete out-of-order
 - This **reorder buffer (ROB)** is also used to pass results among instructions that may be speculated

Speculative Tomasulo



Changes in the data-path:

- ROB (Reorder Buffer)
- Elimination of the store buffers
 - Their function is integrated into the ROB.
- This mechanism can be extended to multiple issues (superscalar) by making the common data bus (CDB) wider to allow for multiple completions per clock.

[1]

MIPS FP Unit using Tomasulo's algorithm extended to handle speculation



Speculative Tomasulo



- Load / Store execution:
 - Loads and stores require a two-step execution as in the non-speculative Tomasulo
 - First step computes the effective address, when the base register is available, and the effective address is then placed in the load or store buffer
 - Loads in the load buffer execute as soon as the memory unit is available
 - Stores in the store buffer wait for the value to be stored before being sent to the memory unit
 - Stores still execute in two steps, but the second step is performed by instruction commit
- ROB completely replaces the store buffers



Reorder Buffer (ROB)



- Reorder Buffer – Possible Entries
 - Instruction type
 - A branch (has no destination result),
 - A store (has a memory address destination)
 - Register operation (ALU operation or load, which has register destinations)
 - Destination
 - Register number (for loads and ALU operations)
 - Memory address (for stores) where the instruction result should be written
 - Value
 - Value of instruction result until the instruction commits
 - Busy
 - Indicates that instruction has completed execution, and the value is ready
 - Supplementary fields to handle speculation and exceptions
 - The speculative field has 3 values: speculative, confirm and not confirmed
 - Should a branch become confirmed it will turn the “speculative” bits of the corresponding speculative instructions to “confirm”.
 - If it is not confirmed, status is set to “not confirmed”



Reorder Buffer (ROB)



- In non-speculative Tomasulo's algorithm, the instructions write their results in the RF.
- With speculation, the RF is not updated until the instruction commits (we know that the instruction should execute):
- The ROB supplies operands in the interval between completion of instruction execution and instruction commit
 - ROB extends architecture registers like Reservation Stations
- Concept of Reorder Buffer (ROB):
 - Holds instructions in FIFO order, exactly as they were issued
 - When instructions complete, results placed into ROB
 - Supplies operands to other instructions between execution complete and commit → more registers like RS
 - Tag results with ROB buffer number instead of reservation station
 - Instructions commit → values at head of ROB are placed in RF
 - As a result, speculated instructions on miss-predicted branches or on exceptions are easily canceled.



Speculative Tomasulo Steps



- Issue
 - Get an instruction from the instruction queue
 - Issue the instruction if there is an empty RS (reservation station at required FU) and an empty slot in the ROB
 - If either all RS are full or the ROB is full, then instruction issue is stalled until both have available entries
 - Allocate a RS and ROB entry
 - Rename the source and destination registers
 - Dispatch the decoded instruction and renamed registers to the RS and ROB
 - Send the operands to the RS if they are available in either the Registers or the ROB
 - Update the control entries to indicate the buffers are in use.
 - The number of the ROB allocated for the result is also sent to the RS, so that the number can be used to tag the result when it is placed on the CDB.



Speculative Tomasulo Steps



- Execute
 - If both operands are available at a RS, execute the operation
 - If one or more of the operands is not yet available, monitor the CDB for them
 - This step checks for **RAW hazards**
 - Instructions may take multiple clock cycles in this stage, and loads still require two steps in this stage.
 - Stores need only have the base register available at this step, since execution for a store at this point is only effective address calculation.



Speculative Tomasulo Steps

- Write Result (finish execution)
 - Condition: At a given FU, some instruction finishes execution
 - When the result is available, write it on the CDB (with the ROB tag sent when the instruction issued) and from the CDB into the ROB, as well as to any RS waiting for this result.
 - Mark the RS as available (de-allocate)
 - Special actions are required for store instructions.
 - If the value to be stored is available, it is written into the Value field of the ROB entry for the store.
 - If the value to be stored is not available yet, the CDB must be monitored until that value is broadcast, at which time the Value field of the ROB entry of the store is updated.



Speculative Tomasulo Steps



- Commit
 - IF: ROB is not empty and ROB head instruction has finished execution
 - Commit valid instructions at the head of the ROB
 - **Commit instructions in-order!**
 - There types of actions depending on the committing instruction:
 - A branch with incorrect prediction / store / any other instruction (normal commit).
 - The normal commit case occurs when an instruction reaches the head of the ROB and its result is present in the buffer; at this point, the processor updates the register with the result and removes the instruction from the ROB.
 - Committing a store is similar: the Memory is updated, not the Register File.
 - When a branch with incorrect prediction reaches the head of the ROB, it indicates that the speculation was wrong.
 - The ROB is flushed and execution is restarted at the correct successor of the branch.
 - If the branch was correctly predicted, the branch is finished.
 - Once an instruction commits, its entry in the ROB is reclaimed and the register or memory destination is updated, eliminating the need for the ROB entry.
 - If the ROB fills, stop issuing instructions until an entry is made free.



Speculative Tomasulo



- **Memory Disambiguation**
 - Difference for store between speculative and classical Tomasulo's algorithm
 - In Tomasulo's algorithm, a store can update memory when it reaches Write Results and the data value to be stored is available
 - In a speculative processor, a store updates memory only when it reaches the head of the ROB
 - This difference ensures that memory is not updated until an instruction is no longer speculative
 - We must avoid hazards through memory
 - **WAW and WAR hazards through memory are eliminated with speculation**
 - Memory updating occurs in order, when a store is at the head of the ROB
 - Hence, no earlier loads or stores can still be pending
 - **RAW hazards through memory are handled by two restrictions**
 - Not allowing a load to initiate the second step of its execution if any active ROB entry occupied by a store has destination address of the load
 - Maintaining the program order for the computation of an effective address of a load with respect to all earlier stores.
 - Together, these two restrictions ensure that any load that accesses a memory location written to by an earlier store cannot perform the memory access until the store has written the data.



Speculative Tomasulo



- How much to speculate?
 - One of the significant advantages of speculation is its ability to cover events that would otherwise stall the pipeline early, such as **cache misses**.
 - A potential disadvantage – the processor may speculate that some costly exceptional event occurs and begins processing the event, when in fact, the speculation was incorrect.
 - To maintain some of the advantage, while minimizing the disadvantages, most pipelines with speculation will allow **only low-cost exceptional events** (such as a **first-level cache miss**) to be handled in speculative mode.
 - If an expensive exceptional event occurs, such as a **second-level cache miss** or a **TLB miss**, the processor will wait until the instruction causing the event is no longer speculative before handling the event.
 - This may slightly degrade the performance of some programs, but avoids significant performance losses in others (high frequency of such events coupled with less than excellent branch prediction)

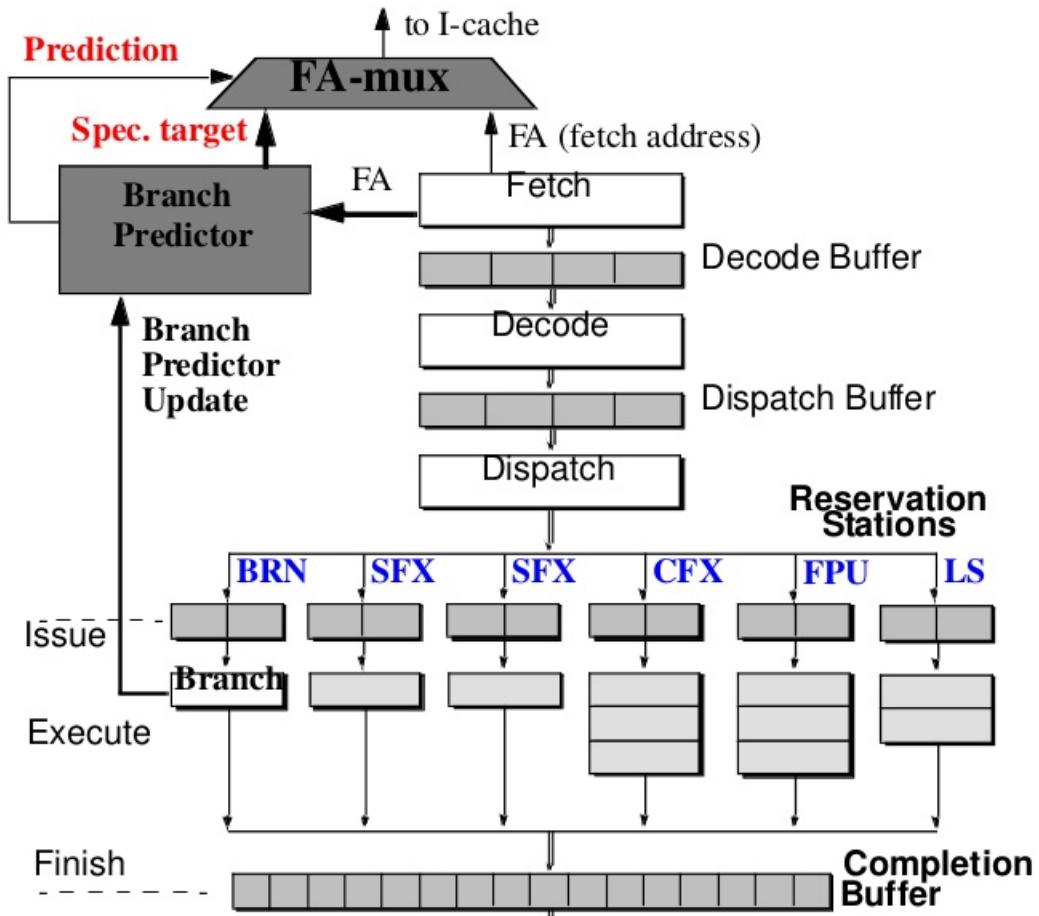


Speculative Tomasulo



- Speculation through multiple branches
 - 3 situations in which speculating on multiple branches is advantageous:
 - A very high branch frequency
 - Significant clustering of branches
 - Long delays in functional units
 - In the first two cases, achieving high performance may mean that multiple branches are speculated
 - Likewise, long delays in functional units can raise the importance of speculating on multiple branches as a way to avoid stalls from the longer pipeline delays
 - Speculating on multiple branches slightly complicates the process of speculation recovery
 - How much parallelism is available depends on the running program
 - The implementation techniques cannot take advantage of more parallelism than the one provided by the application

- Branch predictor – determines if a conditional branch is likely to be taken or not!



General Pipeline CPU Architecture with Branch Predictor



Branch Prediction



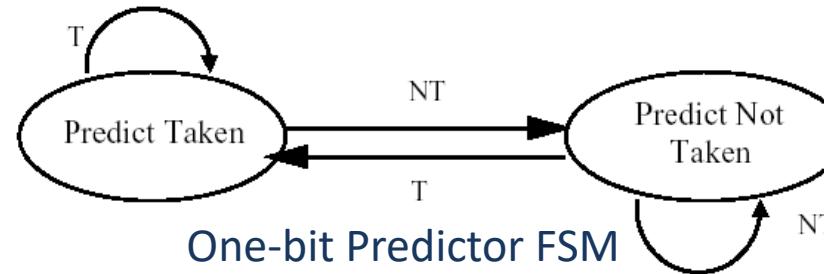
- **Static branch prediction**
 - Compilers decide the direction
 - Direction of the branch – sign bit of the offset; predict:
 - backward branches taken
 - forward branches not taken
 - Algorithm: Backward Taken Forward Not taken (BTFN) ~ 65% correct (SPECint98)
 - Other:
 - Special branch instructions that encode the compiler's prediction – 1-bit
 - Profiling or feedback-directed compilation
- **Dynamic branch prediction**
 - Hardware decides the direction using history information
 - Dynamic algorithms take into account run-time information
 - The processor learns from its mistakes and changes its predictions to match the behavior of each particular branch
 - A dynamic algorithm keeps a record of previous branch behavior, allowing it to improve its predictions over time



Dynamic Branch Prediction



- **1-bit predictor**
 - A simple scheme, maintains a single history bit for each branch
 - When a branch is encountered, it is predicted to go the same way it did the previous time
 - This technique can push accuracy to 80%.

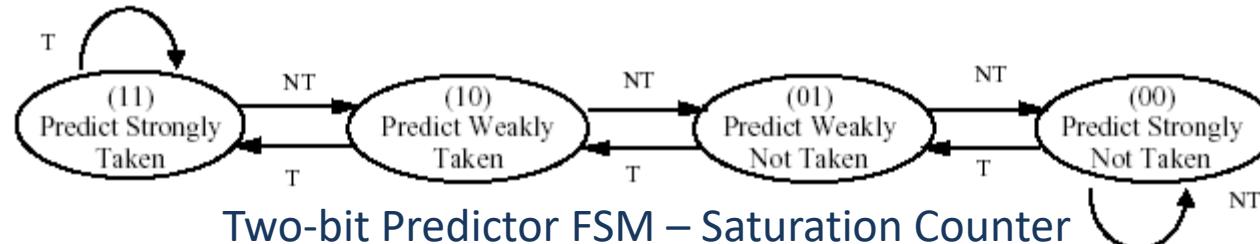


- A one-bit predictor correctly predicts a branch at the end of loop iteration, as long as the loop does not exit.
- In nested loops, a one-bit prediction scheme will cause two miss-predictions for the inner loop:
 - One at the end of the loop, when the iteration exits the loop and
 - One at the first loop iteration, when it predicts exit instead of looping
 - **2-bit predictor – avoids the double miss-prediction in nested loops**



Dynamic Branch Prediction

- 2-bit predictor



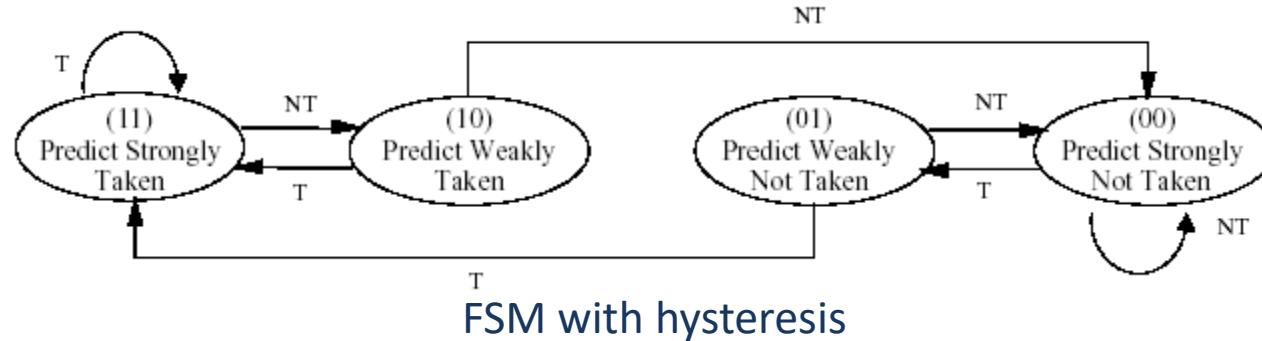
- States 01, 10 – one wrong prediction changes direction
- States 00, 11 – two wrong predictions change direction
- The counter is inc./dec. when the branch is taken/not-taken
- The most-significant bit is used to predict future occurrences
- Saturated counter
- By using a 2-bit counter, the predictor can tolerate a branch going in an unusual direction one time and keep predicting the usual branch direction.
- This hysteresis effect can boost prediction accuracy to 85% on SPECint92, depending on the size and type of history table that is used.
- Studies showed that a two-bit prediction scheme does almost as well as an $n > 2$ -bit scheme



Dynamic Branch Prediction

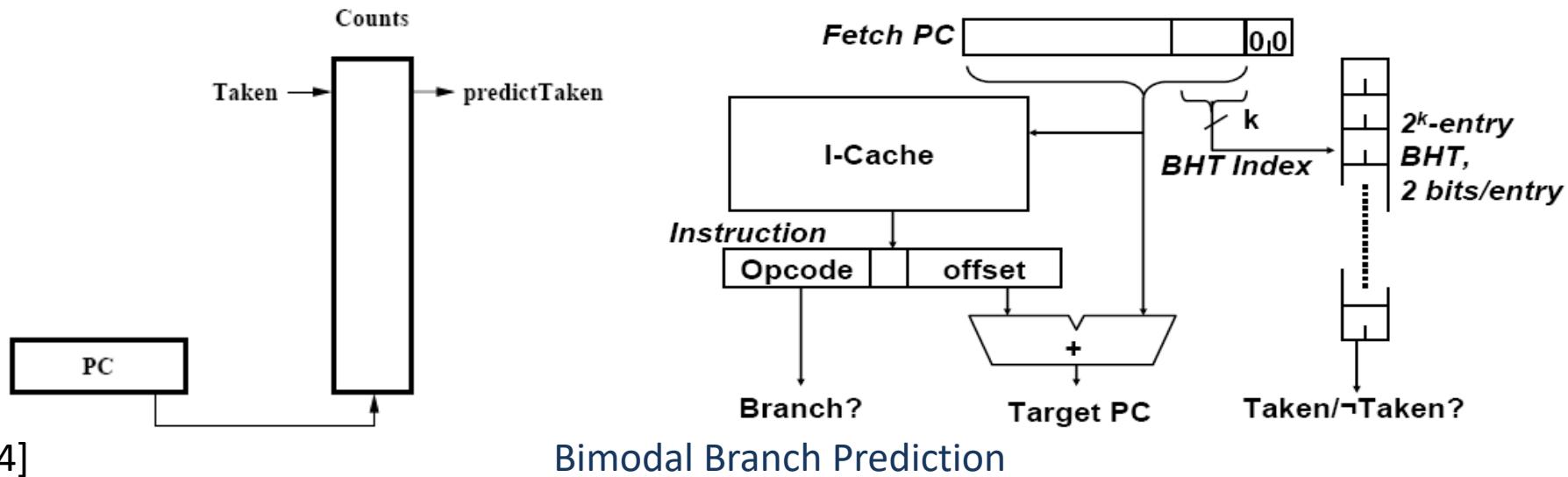


- 2-bit predictor – UltraSPARC



- More hysteresis, two wrong predictions needed to change the direction
- No direct connections between Weakly Taken and Weakly Not Taken states

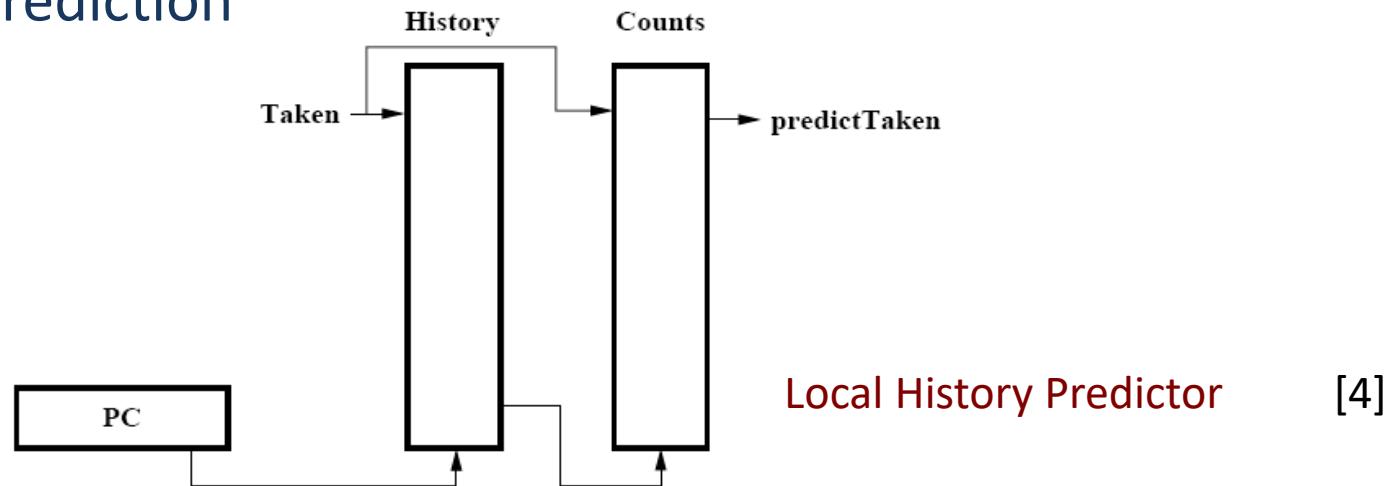
- Bimodal Branch Prediction
 - Branches are either taken or not taken.
 - Bimodal branch prediction takes advantage of behavior



- Counts: Table of 2-bit counters indexed by the low order address bits of PC
 - **BHT – Branch History Table**
- For each taken/not-taken branch, the appropriate counter is inc. / dec.
- Prediction is based on the current state (not updated) of the selected counter



- Local Branch Prediction



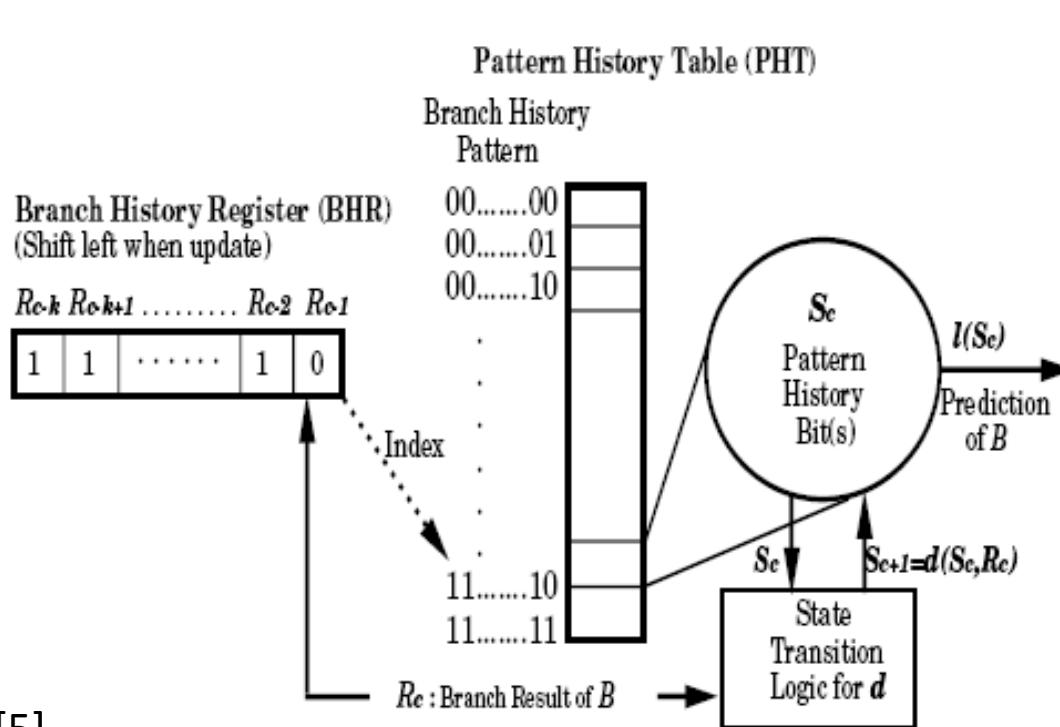
- The first table records the **history** of recent branches: an array indexed by the low-order bits of the branch address.
- Performance decreases when multiple branches point to the same entry
- The history table records the direction of the last n branches whose addresses map to the same entry; Ex: $n=6$ **100100** 1=taken, 0=not taken
- The second table (**Counts**) is an array of 2-bit counters identical to those used for bimodal branch prediction (indexed with history – reduces aliases)
- Referred to as local branch prediction (Scott McFarling)



Dynamic Branch Prediction

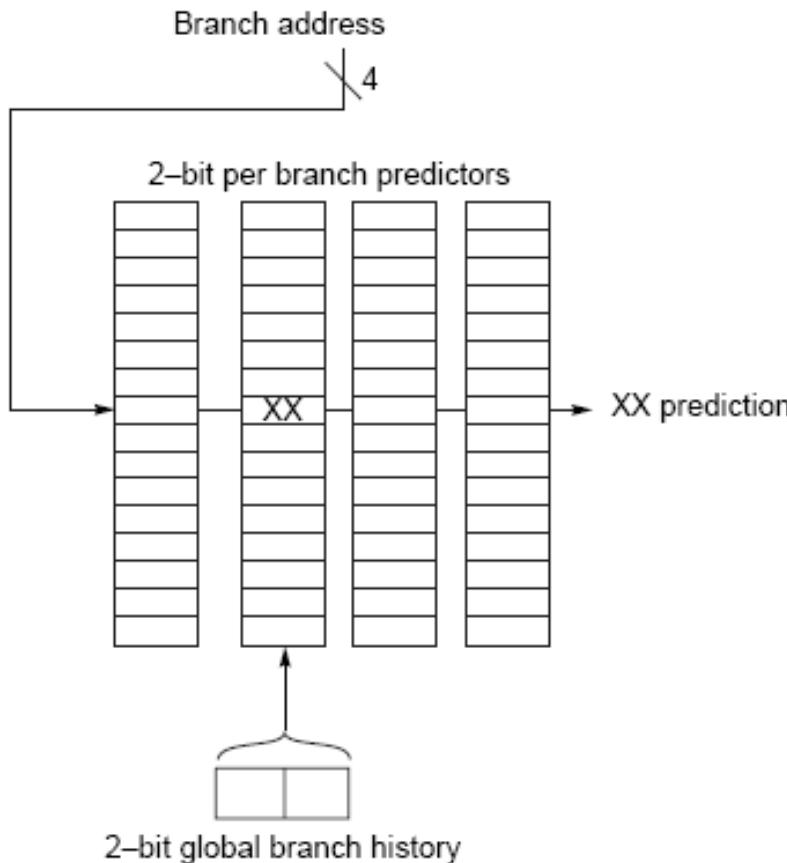


- Global Branch Prediction
 - 2-level Adaptive Prediction [Yeh & Patt]
 - Branch predictors that use the behavior of other branches to make a prediction are called **correlating predictors** or **two-level predictors**



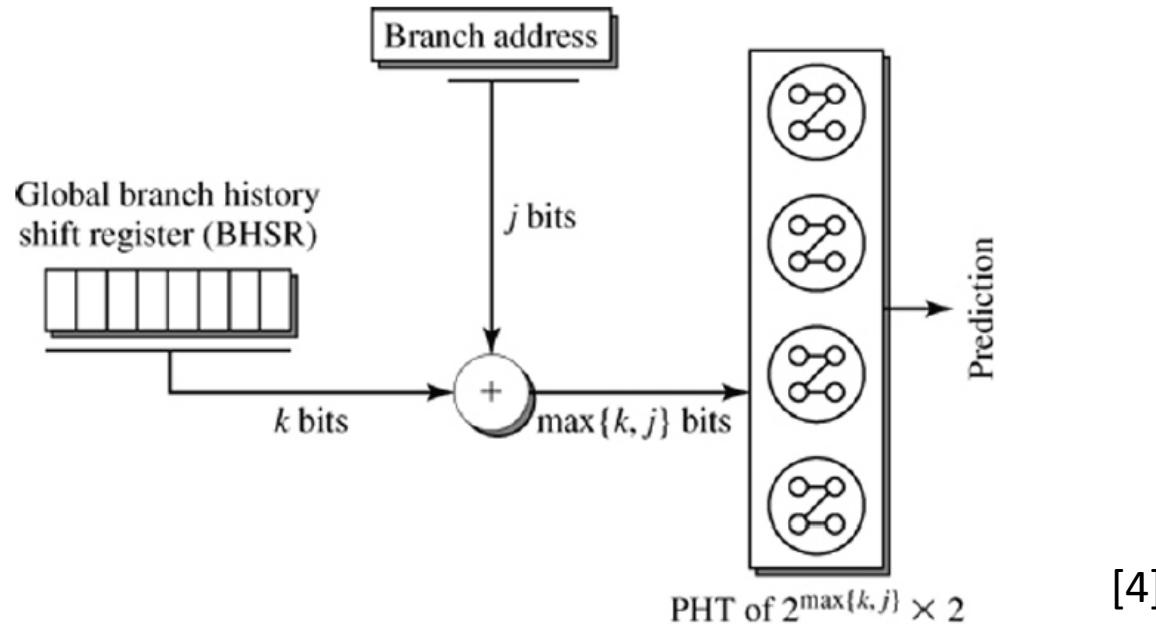
Two-level adaptive branch prediction (Global BHR and global PHT)

- 2-level adaptive branch prediction



- 2-bit global history selects 1 of 4 predictors for each branch address
- Each predictor is in turn a two-bit predictor for that particular branch
- $4 \times 16 = 64$ entries; the branch address is used to choose four of these entries and the global history is used to choose one of the four
- Global history is a shift register that shifts in the behavior of the branch
- ARM Cortex A57

- Gshare Branch Prediction [McFarling]



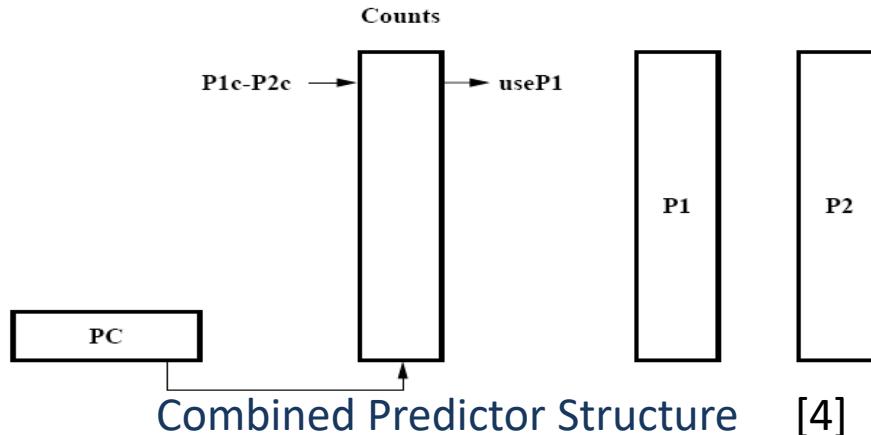
- An address maps to many locations
- Combine branch address with history to reduce aliasing and capture context
- Ex: AMD Athlon, MIPS R12000, Intel Atom (Silvermont MicroArchitecture), ARM Cortex A53



Dynamic Branch Prediction



- Combining Branch Predictors
 - Two predictors P1 and P2 that could be one of the predictors discussed
 - An additional **counter array** serves to select the best predictor to use
 - 2-bit up/down saturating counters are used
 - Each counter keeps track of which predictor is more accurate for the branches that share that counter
 - Notation: P1c and P2c denote correct predictions for P1 and P2
 - The counter is incremented or decremented by P1c-P2c as shown
 - One combination of branch predictors that is useful is bimodal/Gshare



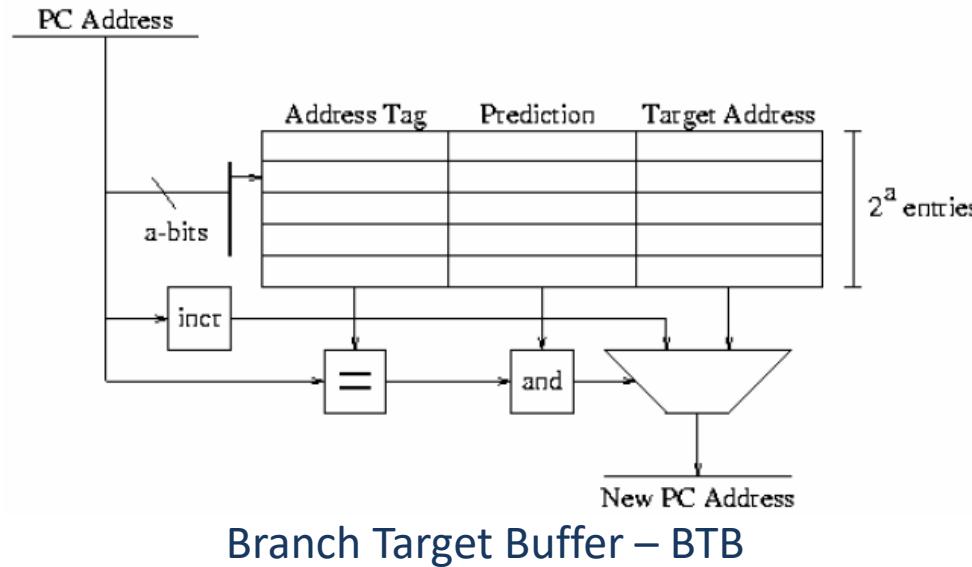
| P1c | P2c | P1c-P2c | Next counter value |
|-----|-----|---------|--------------------|
| 0 | 0 | 0 | No change |
| 0 | 1 | -1 | Decrement counter |
| 1 | 0 | 1 | Increment counter |
| 1 | 1 | 0 | No change |



Dynamic Branch Prediction



- Branch Target Buffer – BTB
 - A branch-prediction **cache** that stores the Target Addresses for the taken branches is called a **branch-target buffer** or **branch-target cache**



Branch Target Buffer – BTB

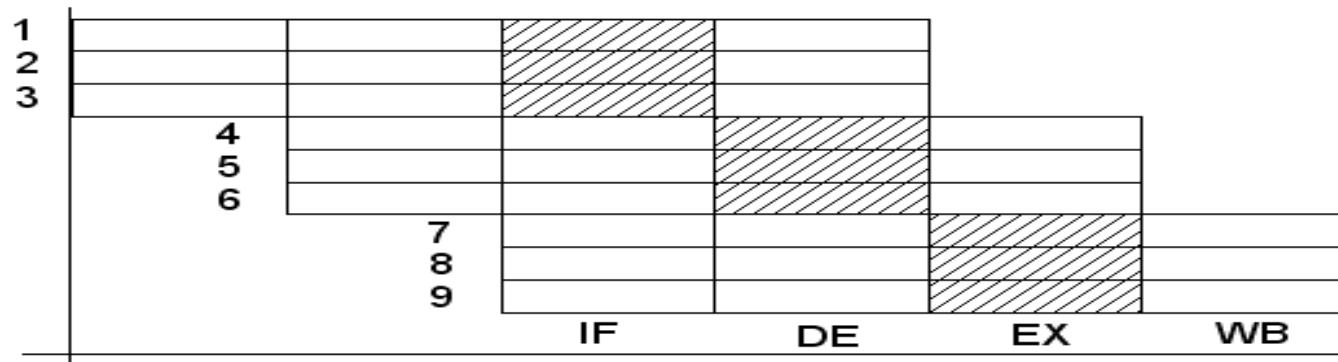
- BTB is used in the instruction fetch stage to determine the new PC.
- If the PC matches one of the BTB entries, the new PC Address is selected according to Prediction; Fetching begins immediately at that address.
- One variation on the branch-target buffer is to store one or more *target instructions* in addition to the predicted *target address*
- Intel Pentium, Intel Pentium MMX



More Advanced Pipelining Concepts



- Superscalar architecture – more ILP
 - Execute more than one instruction in one clock cycle by simultaneously dispatching multiple instructions
 - Exploit ILP
 - A superscalar **is dynamically scheduled in hardware**



- Branch prediction – more critical as pipelines get longer and wider
 - Predict both the branch condition and the target
- Use more registers (reservation stations), load/store buffers, re-order buffer
- In order issue – out of order execution – in order commit



More Advanced Pipelining Concepts



- Multi-Core CPUs – single-chip multiprocessors
- Thread Level Parallelism (TLP) – Hardware Multithreading
 - Fine-grained Multithreading
 - Switching between threads after every instruction
 - Coarse-grained Multithreading
 - Switching between threads only after significant events (Ex: cache miss)
 - Simultaneous Multithreading (SMT)
 - Combines hardware multithreading with superscalar processors to allow multiple threads to issue instructions in each clock cycle
 - Speculation is necessary for SMT performance
 - Intel Hyper-threading = SMT
 - With Hyper-Threading the computer can have one physical processor installed in the motherboard, but the OS will see two logical processors, and treat the system as if there were actually two processors.



References



1. D. A. Patterson, J. L. Hennessy, “Computer Organization and Design: The Hardware/Software Interface”, 5th edition, ed. Morgan–Kaufmann, 2013.
2. R. Iris Bahar, Advanced Computer Architecture, Lecture 9: Branch Prediction, EN292-S10 October 3, 2006
3. Shen & Lipasti, Modern Processor Design: Fundamentals of Superscalar processors, *McGraw Hill*, 2005
4. Combining Branch Predictors; *Scott McFarling* WRL Technical Note TN-36, 1993
5. A Comparison of Dynamic Branch Predictors that use Two Levels of Branch History, Tse-Yu Yeh and Yale N. Patt, Department of Electrical Engineering and Computer Science, University of Michigan.



Computer Architecture

Lecturer: Mihai Negru

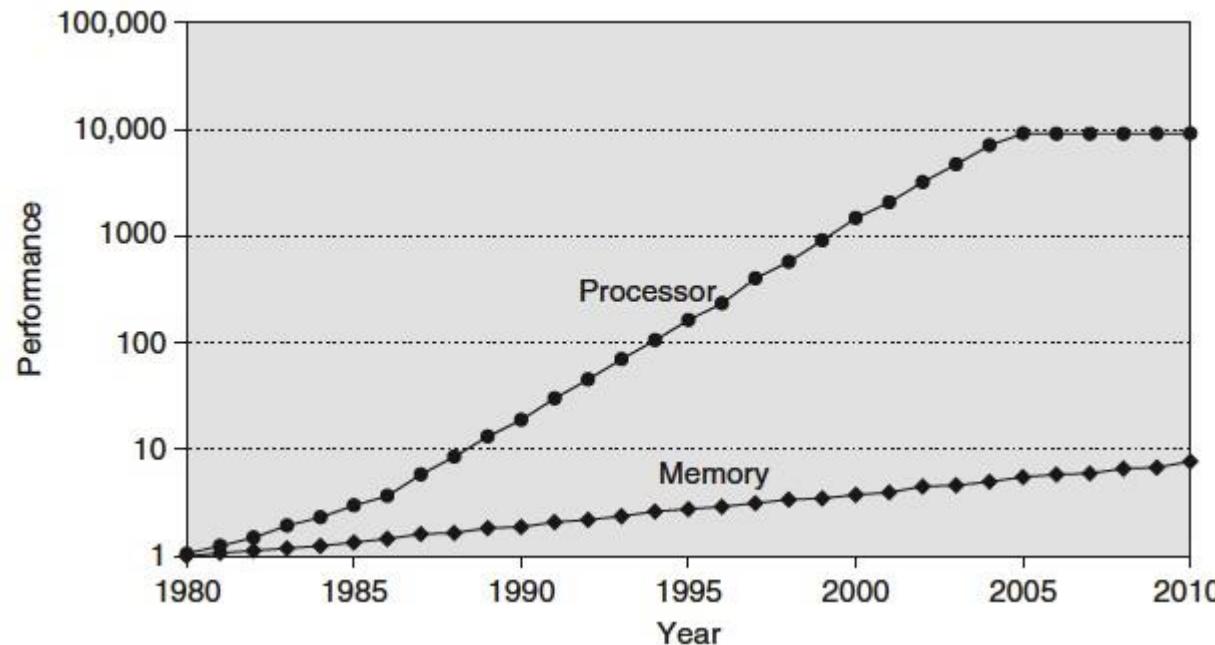
2nd Year, Computer Science

Lecture 11: Memory

<http://users.utcluj.ro/~negrum/>



Processor – Memory Performance Gap

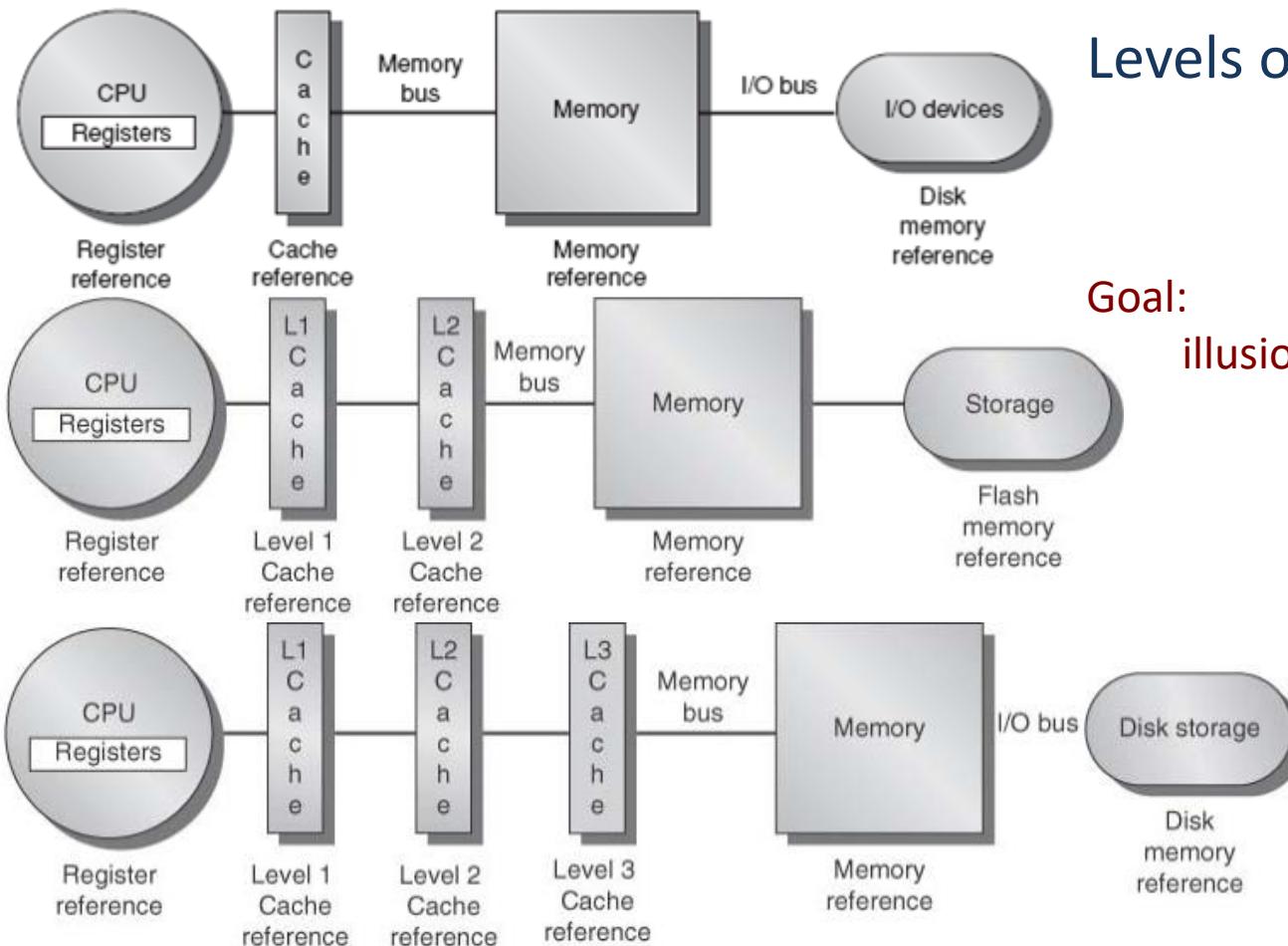


Processor (Single Core) vs. Memory (DRAM) Performance Gap [1]

| Memory Technology | Typical Access Time | Cost / GB in 2012 |
|----------------------------|---------------------------|-------------------|
| SRAM semiconductor memory | 0.5 – 2.5 ns | \$500 – \$1000 |
| DRAM semiconductor memory | 50 – 70 ns | \$10 – \$20 |
| Flash semiconductor memory | 5,000 – 50,000 ns | \$0.75 – \$1.00 |
| Magnetic disk | 5,000,000 – 20,000,000 ns | \$0.05 – \$0.10 |



Memory Hierarchy



Levels of memory hierarchy [1]

Goal:

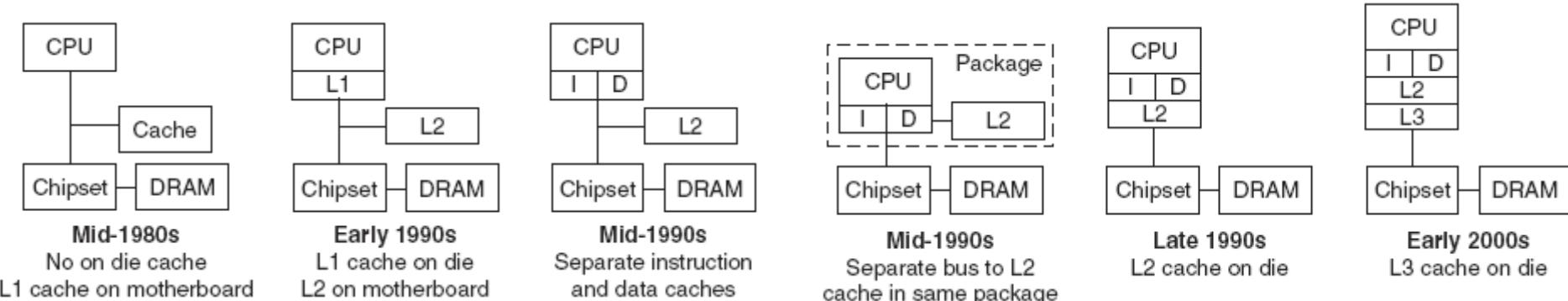
illusion of large, fast, cheap memory

Distance from Processor → lower speed but greater size

Cache – a safe place for hiding or storing things!



Memory Hierarchy Evolution



The Evolution of the Memory Hierarchy. Separate Instruction and Data Caches

Why hierarchy works? The Principle of Locality:

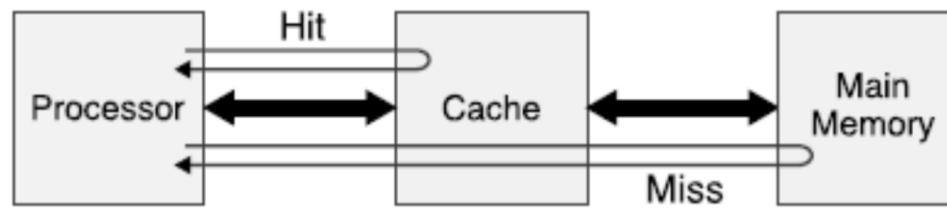
- **Temporal Locality – Locality in Time**
 - If a data location is referenced then it will tend to be referenced again soon
 - Keep most recently accessed data items closer to the processor.
- **Spatial Locality – Locality in Space**
 - If a data location is referenced, nearby addresses will tend to be referenced soon
 - Move blocks consisting of contiguous words to the upper levels.



Cache Memory Terminology



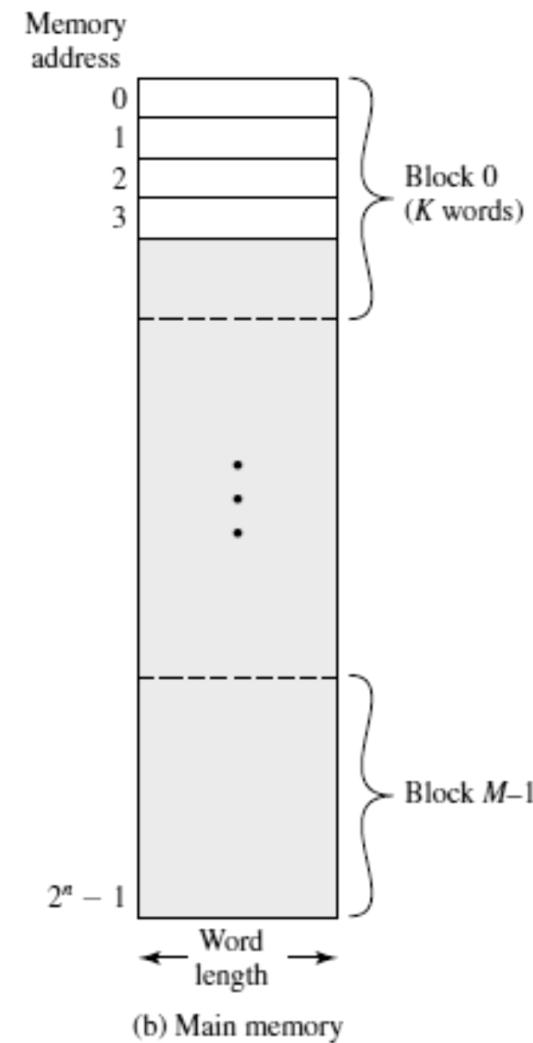
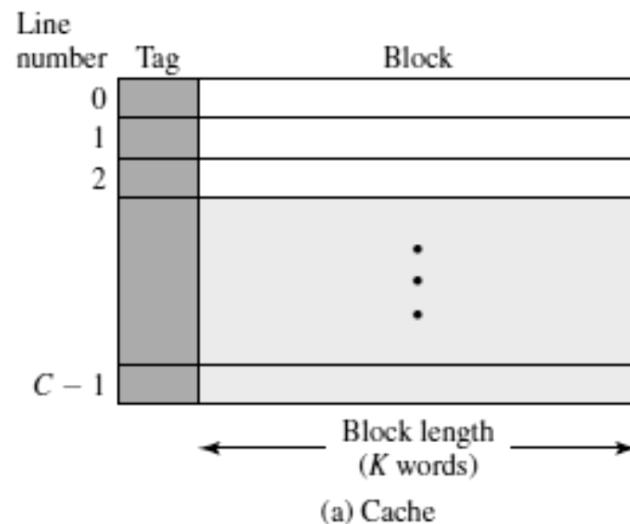
- Hit: data requested is in upper level.
- Miss: data requested is not in upper level.
- Hit rate: fraction of memory accesses that are hits (i.e., found at upper level).
- Miss rate: fraction of memory accesses that are not hits: $\text{miss rate} = 1 - \text{hit rate}$
- Hit time: time to determine if the access is a hit + time to access and deliver the data from the upper level to the CPU.
- Miss penalty: time to determine if the access is a miss + time to replace block at upper level with corresponding block at lower level + time to deliver the block to the CPU
- Average memory access time (AMAT) = Hit time + Miss rate x Miss penalty



[1]

- To improve performance → reduce AMAT
 - Reduce the miss rate, miss penalty, or the hit time

Cache Memories



Block in Cache and Main Memory [1]

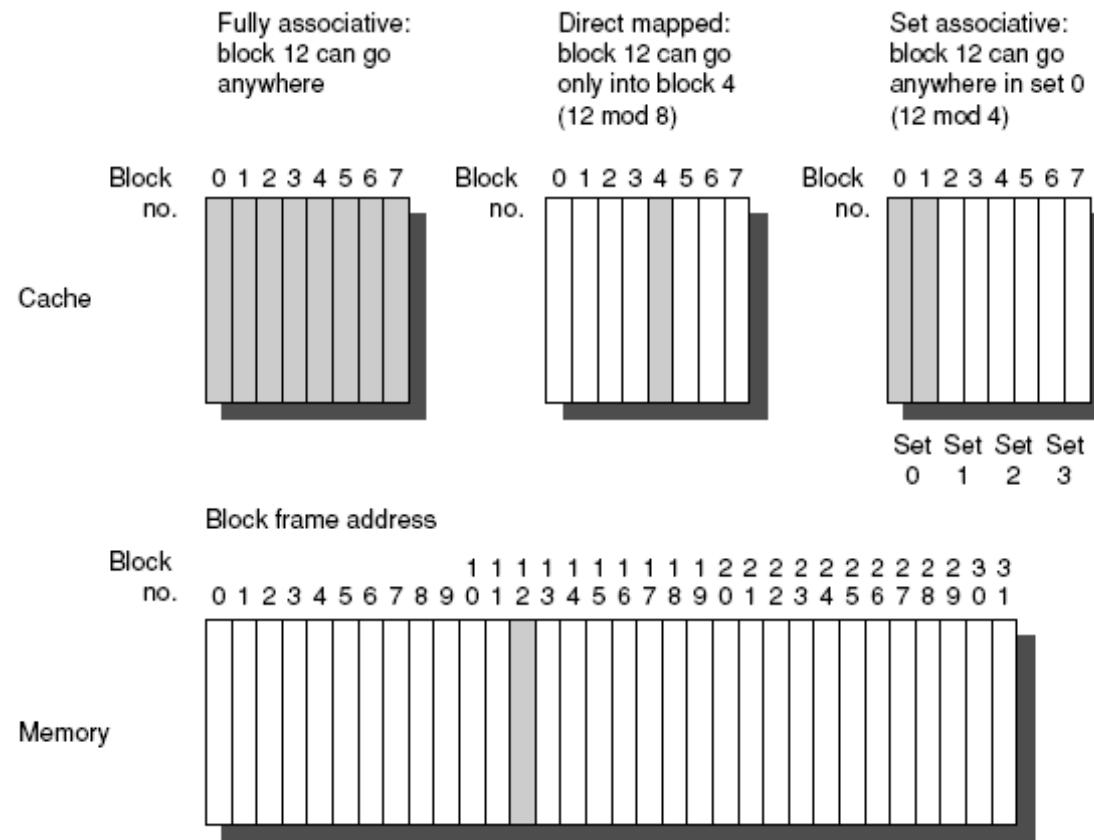


Cache Memories



- 4 Problems for Cache Memory Specification
 - Q1: Where can a block be placed in the Cache? (Block placement)
 - Associativity: Fully Associative, Set Associative, Direct Mapped
 - Q2: How is a block found in the Cache? (Block identification)
 - Tag / Index / Block
 - Q3: Which block should be replaced on a Cache miss? (Block replacement)
 - Random, LRU, FIFO, NLRU, FIFO with exception for most recently used
 - Q4: How to write in the Cache? (Write strategy)
 - Write Back or Write Through, Write Buffer

Q1: Block Placement in the Cache Memory



Example: Cache memory with 8 blocks [1]

Address mapping for set assoc. cache:

(Block address) MOD (Nb. sets in the cache)

“The miss rate of a Direct Mapped Cache of size X is about the same as
for a 2- to 4-way Set Associative cache of size X/2”

Cache Memory Organization

- **Direct Mapped Cache** – each block has only one corresponding place in the cache
- **Fully Associative Cache** – a block can be placed anywhere in the cache
- **Set Associative Cache** – a block can be placed in a restricted set of places in the cache. N sets in a cache → N-way Set Associative



Q1: Block Placement in the Cache Memory



| Block | Tag | Data |
|-------|-----|------|
| 0 | | |
| 1 | | |
| 2 | | |
| 3 | | |
| 4 | | |
| 5 | | |
| 6 | | |
| 7 | | |

Direct Mapped Cache
1-way Set Associative Cache

Set = a group of blocks,
Index in the cache memory

| Set | Tag | Data | Tag | Data | Tag | Data | Tag | Data |
|-----|-----|------|-----|------|-----|------|-----|------|
| 0 | | | | | | | | |
| 1 | | | | | | | | |
| 2 | | | | | | | | |
| 3 | | | | | | | | |

2-way Set Associative Cache

| Set | Tag | Data | Tag | Data | Tag | Data | Tag | Data |
|-----|-----|------|-----|------|-----|------|-----|------|
| 0 | | | | | | | | |
| 1 | | | | | | | | |

4-way Set Associative Cache

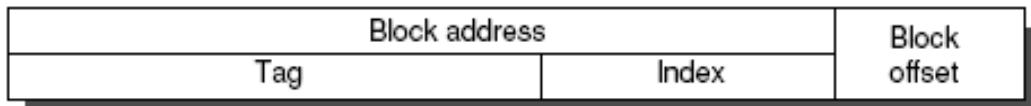
| Tag | Data |
|-----|------|-----|------|-----|------|-----|------|-----|------|-----|------|-----|------|
| | | | | | | | | | | | | | |

8-way Set Associative (Fully Associative)

Configurations of an 8-block cache with different degrees of associativity.

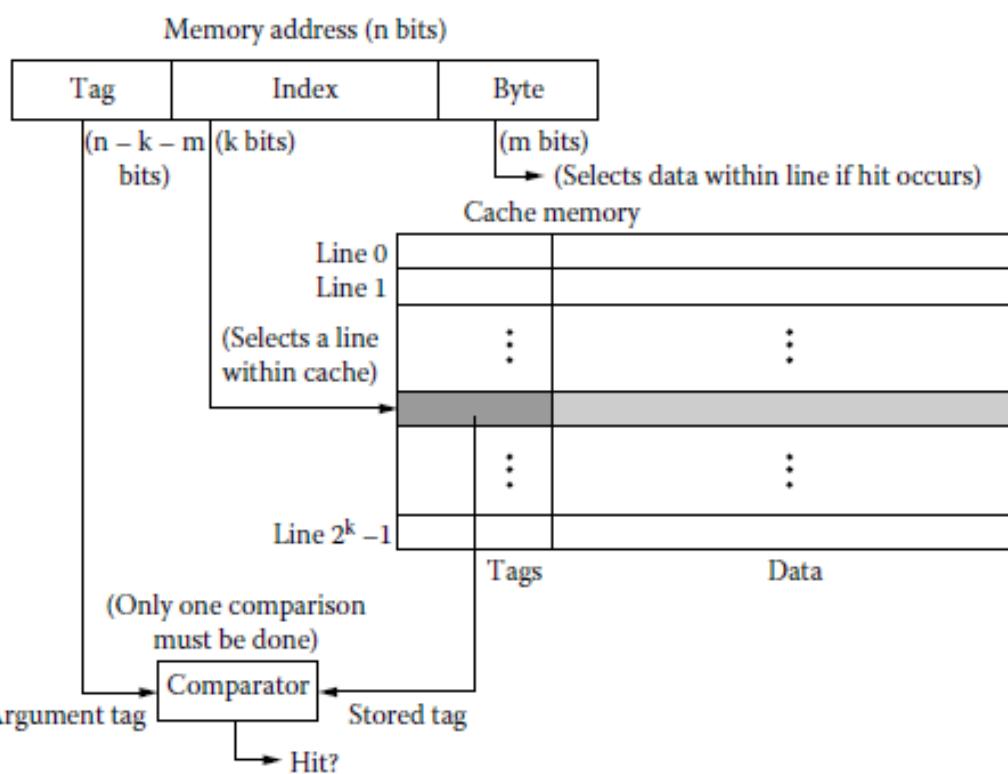


Q2: Block Identification in the Cache Memory

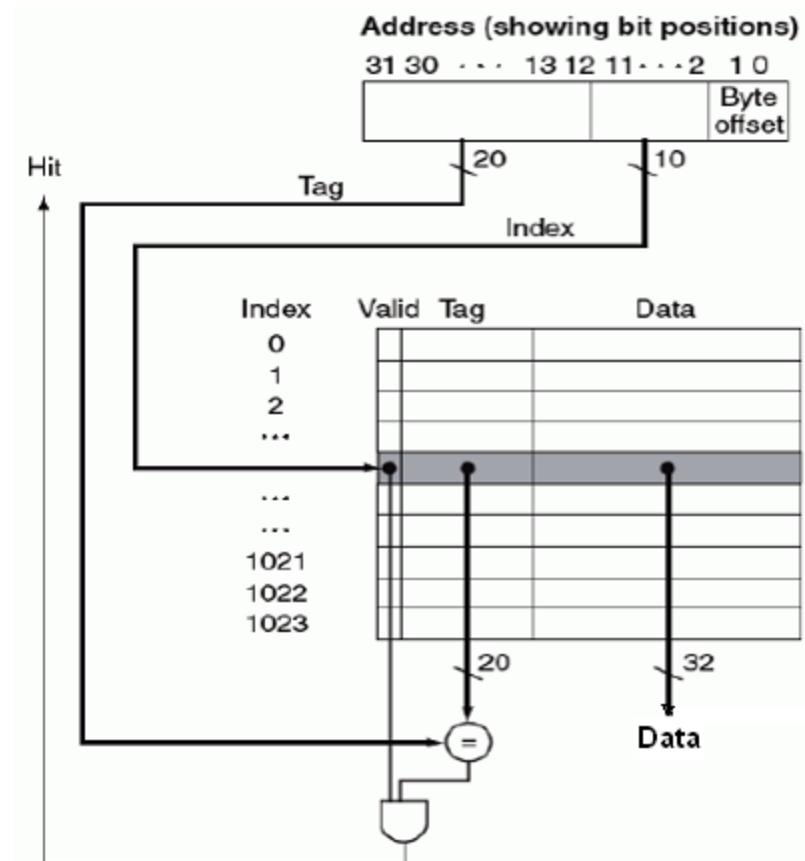


Address for a Set Associative or Direct Mapped Cache

A fully Associative caches has no index field

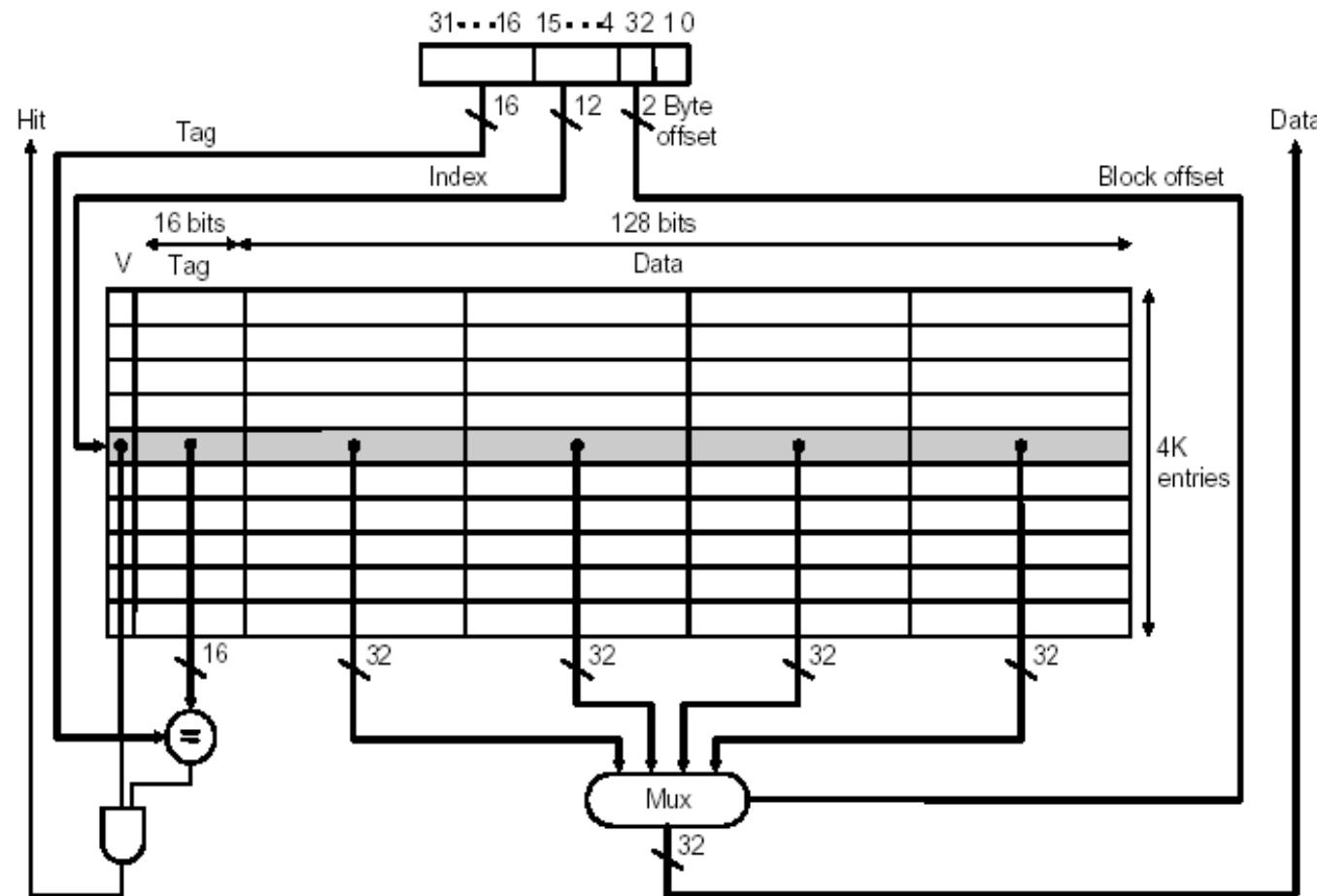


Direct Mapped Cache – general [1]



Direct Mapped Cache, 1024, 1-word blocks [1]

Q2: Block Identification in the Cache Memory



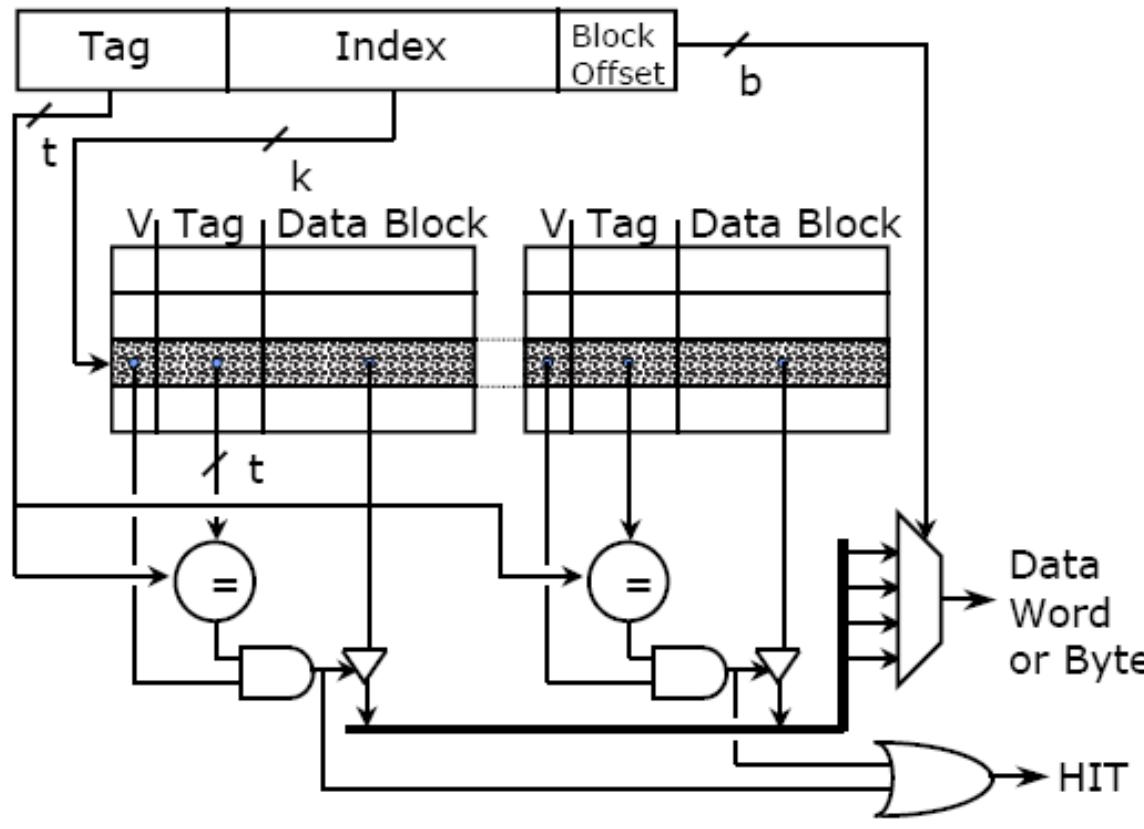
Direct Mapped Cache with multiple data/tag [1]. Taking advantage of spatial locality

- 64 KB cache, 4K blocks, 4 words per block; byte offset ignored – we read words (32 bits) from cache; block offset – which word to read.

Q2: Block Identification in the Cache Memory

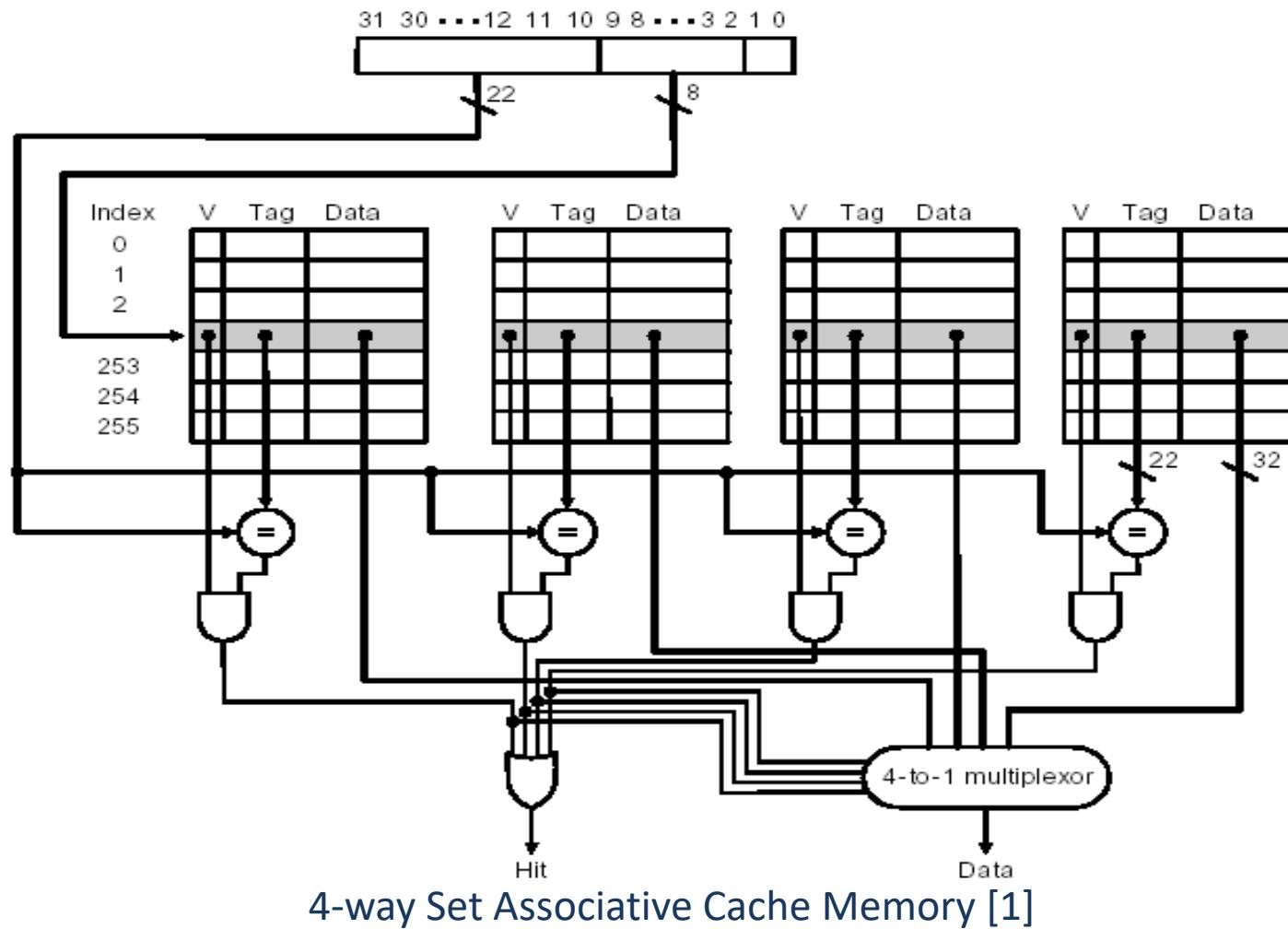


- **N-way Set Associative Cache:** N entries for each cache index
 - N Direct Mapped Caches that operate in parallel



2-way Set Associative Cache

Q2: Block Identification in the Cache Memory



4-way Set Associative Cache Memory [1]

- 4-way set-associative cache with 4 comparators and one 4-to-1 multiplexor
- Size of cache: 4 KB cache, 1K blocks = 256 sets * 4-block/set (4x256 blocks, 1 word per block)

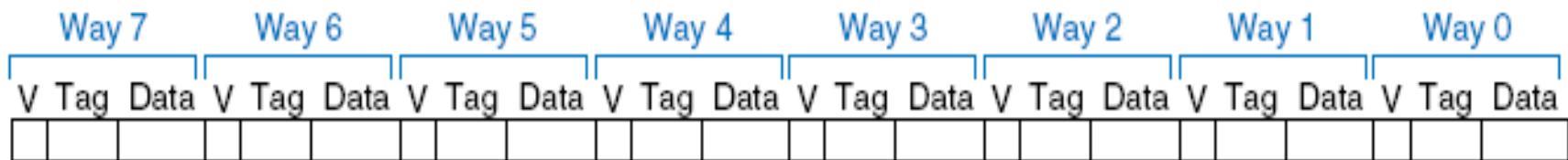


Q2: Block Identification in the Cache Memory

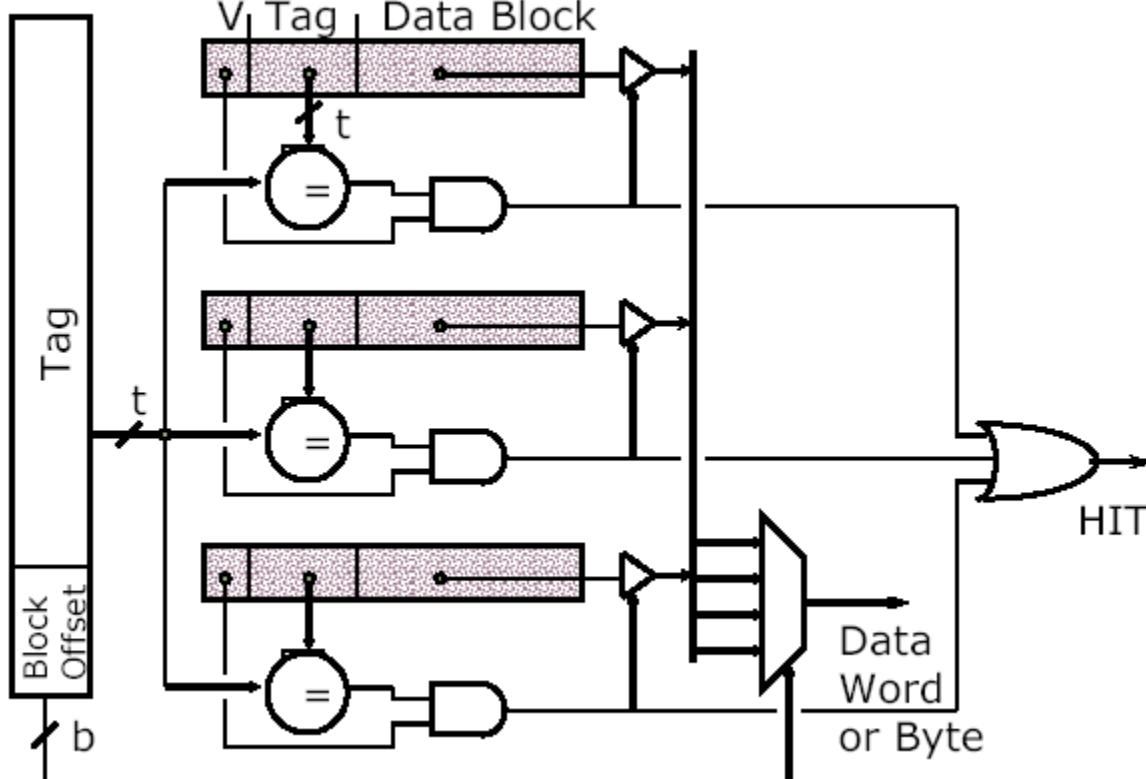


- Advantages of Set Associative Cache
 - Higher Hit rate for the same cache size.
 - Fewer Conflict Misses.
- Disadvantages of Set Associative Cache
 - N-way Set Associative Cache versus Direct Mapped Cache
 - N comparators vs. 1
 - Extra MUX delay for the data
 - Data comes **AFTER** Hit/Miss decision and set selection
 - In a Direct Mapped Cache, Cache Block is available **BEFORE** Hit/Miss
 - Possible to assume a hit and continue. Recover later if miss.

Q2: Block Identification in the Cache Memory



Fully Associative cache, 8 blocks [1]



Fully Associative Cache Memory

Fully Associative Cache

- No Cache Index.
- Compare the Cache Tags of all cache entries in parallel.
- Needs a lot of comparators.
- Implemented using content addressable memory (CAM).
- Conflict Miss = 0 for a fully associative cache.



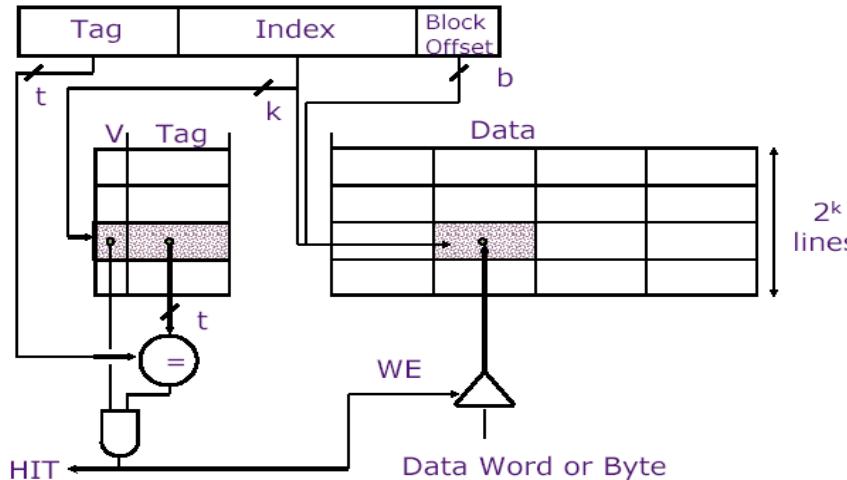
Q3: Block Replacement on a Cache Miss



- Direct Mapped Cache – easy: only one possible block to replace
- Associative cache – need a block replacement algorithm
 - Least Recently Used (LRU)
 - Expensive – keeps track when an element in the set was used
 - For 2-way set assoc. – use 1 bit (**USE bit**)
 - On a block reference Use bit $\leftarrow 1$; Use Bit of the other block $\leftarrow 0$
 - For fully associative – keep track of all references. Use a list: the most recently used block is the front of the list. The last block in the list is replaced
 - First-In-First-Out (FIFO)
 - Replace the block that has been in the cache longest
 - Easy to implement as a round-robin or circular buffer reference
 - Least Frequently Used (LFU)
 - Counter per block that increments on reference
 - Block with lowest count is replaced
 - Most Recently Used (MRU)
 - Random
 - Victim blocks are randomly selected
 - Simulations indicate almost as good as LRU



Q4: Write Strategy for Cache Memories



- Cache write
 - Modifying a block cannot begin until the tag is checked to see if the address is a hit.
 - Tag checking is done before the write → writes normally take longer than reads.
 - Write size: 1 – 8 bytes specified; only that portion of a block can be changed.
 - In contrast, reads can access more bytes than necessary
 - **Pipelined writes:** hold write data for store in single buffer ahead of cache, write cache data during next store's tag check



Q4: Write Strategy for Cache Memories



- Write Policy Choices
 - Write-Through (WT)
 - Replaces a block in the cache and low-level memory to avoid inconsistency
 - Write-through is slow because it always requires a write in main memory
 - Performance is improved with a **write buffer** where blocks are stored while waiting to be written to memory – processor can continue execution until write buffer is full
 - Advantages: read misses do not result in writes and assures **data coherency**
 - Write-Back (WB)
 - Write the data block only into the cache and write-back the block to main memory only when it is replaced in cache
 - More efficient than write-through, more complex to implement
 - A **dirty bit** per block can further reduce the traffic
 - Write Once – first write as write-through, the followings as write back



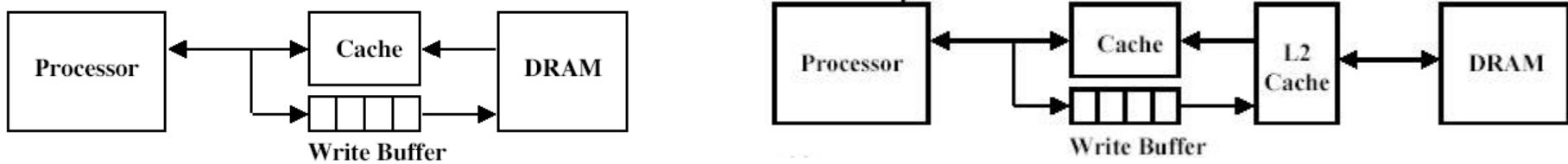
Q4: Write Strategy for Cache Memories



- Write miss actions: **allocate** block if it's a miss?
 - **Write allocate** – the block is allocated on a write miss, followed by the write hit.
 - **No write allocate** – only write to main memory.
 - **Common combinations**
 - Write through and no write allocate, even if there are subsequent writes to that block, the writes must still go to the lower level memory.
 - WT combined with write buffers so that it doesn't wait for memory.
 - Write back with write allocate, hoping that subsequent writes to that block will be captured by the cache

- **Write Buffer**

- Contains evicted dirty lines for WB cache or all writes in WT cache
- It reduces **Read Miss Penalty**
 - Processor is not stalled on writes, read misses can go ahead of writes to main memory



- Implemented as a **FIFO** (queue) – holds data to be written to memory
- Memory controller writes contents of the buffer to memory
 - **Frees** the write buffer data entry after completing memory write
 - Stall the CPU if write buffer is full
- **Problem: Write Buffer may hold a value needed by a read miss!**
 - Simple: on a read miss, wait for the write buffer to go empty
 - Faster: check write buffer addresses against read miss addresses
 - if no match, allow read miss to go ahead of writes, else
 - return the value from the write buffer



Cache Memory Performance



- AMAT (average memory access time)
- CPU Time

AMAT = Hit time + Miss rate × Miss penalty

CPU time = (CPU Execution clock cycles + **Memory stall clock cycles**) × Clock cycle time

$$\text{CPU time} = \text{IC} \times \left(\text{CPI}_{\text{execution}} + \frac{\text{Memory stall clock cycles}}{\text{Instruction}} \right) \times \text{Clock cycle time}$$

$$\text{CPU time} = \text{IC} \times \left(\text{Miss rate} \times \frac{\text{Memory access}}{\text{Instruction}} \times \text{Miss penalty} \right) \times \text{Clock cycle time}$$

CPU Execution clock cycles – includes the execution clock cycles and the memory access for a cache hit

Memory stall clock cycles – includes the auxiliary penalties for working with memory

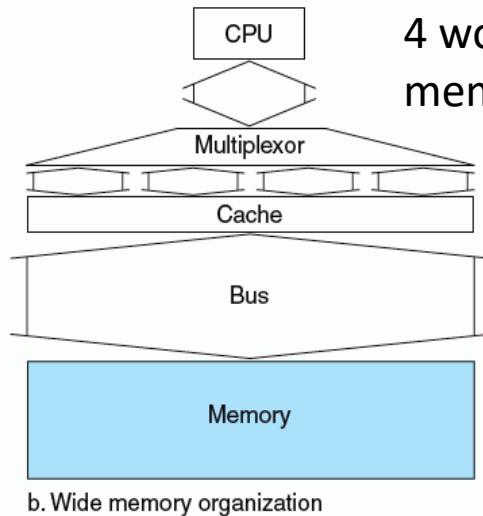
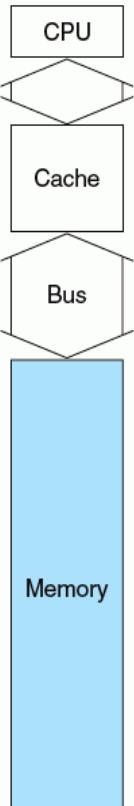


Causes for Cache Misses

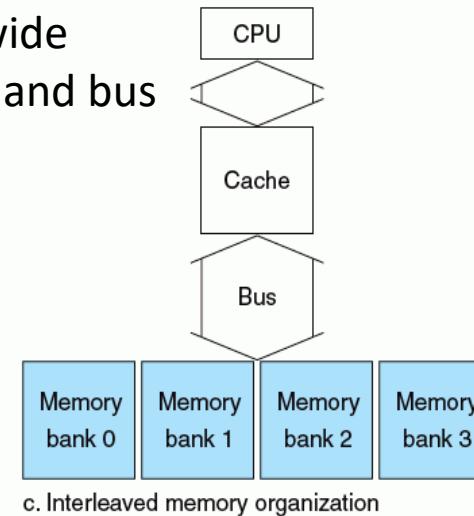
- 3 C's model
 - **Compulsory**: first-reference to a block (cold start misses)
 - Misses that would occur even with infinite cache
 - **Capacity**: cache is too small to hold all data needed by the program
 - Misses that would occur even under perfect replacement policy
 - **Conflict**: misses that occur because of collisions due to block-placement strategy
 - Misses that would not occur with full associativity
- 4th C: **Coherence**
 - Misses caused by cache coherence (Multiprocessors)

| Design change | Effect on miss rate | Possible negative performance effect |
|------------------------|---|--|
| Increase cache size | Decreases capacity misses | May increase access time |
| Increase associativity | Decreases miss rate due to conflict misses | May increase access time |
| Increase block size | Decreases miss rate due to spatial locality | Increases miss penalty. Very large block size can increase miss rate |

Cache Memory Connections



4 word wide
memory and bus



4 word wide memory
Interleaved memory units
compete for bus

Example: we assume

- cache block of 4 words
- 1 clock cycle to send address to memory address buffer (1 bus cycle)
- 15 clock cycles for each memory data access
- 1 clock cycle to send data to memory data buffer (1 bus cycle)

Miss penalties

- a: $1 + 4*15 + 4*1 = 65$ cycles
- b: $1 + 1*15 + 1*1 = 17$ cycles
- c: $1 + 1*15 + 4*1 = 20$ cycles

Improving Cache Performance by Increasing Bandwidth [1]



Cache Memory Evolution

| Processor | Year | Frequency (MHz) | Level 1 Data Cache | Level 1 Instruction Cache | Level 2 Cache |
|-------------------|------|-----------------|--------------------|---------------------------|------------------------------|
| 80386 | 1985 | 12 – 40 | none | none | None |
| 80486 | 1989 | 16 – 150 | 8 KB unified | | None on chip |
| Pentium | 1993 | 60 – 100 | 8 KB | 8 KB | None on chip |
| Pentium Pro | 1995 | 150 – 200 | 8 KB | 8 KB | 256 KB – 1 MB |
| Pentium II | 1997 | 233 – 450 | 16 KB | 16 KB | 256 KB – 512 KB |
| Pentium III | 1999 | 450 – 1400 | 16 KB | 16 KB | 256 KB – 512 KB |
| Pentium 4 | 2001 | 1400 – 3730 | 8-16 KB | 12 KB | 256 KB – 2 MB |
| Pentium M | 2003 | 900 – 2130 | 32 KB | 32 KB | 1 – 2 MB on chip |
| Core Duo | 2005 | 1500 – 2160 | 32 KB / core | 32 KB / core | 2 MB shared on chip |
| Skylake (Core i7) | 2015 | Up to 4200 | 32 KB / core | 32 KB / core | 256 KB / Core (8 M L3 cache) |

Evolution of intel IA-32 Microprocessor Cache Memory Systems



Virtual Memory



- Virtual address space, i.e., space addressable by a program is determined by ISA
- Main memory size \leq disk size \leq virtual address space size
- Virtual memory is organized in fixed-size (power of 2, typically at least 4 KB) blocks, called **pages**
- Physical memory is considered a collection of pages of the same size
- The unit of data transfer between disk and physical memory is a page
- Advantages of Virtual Memory:
 - Illusion of having more physical memory
 - Program reallocation
 - Protection

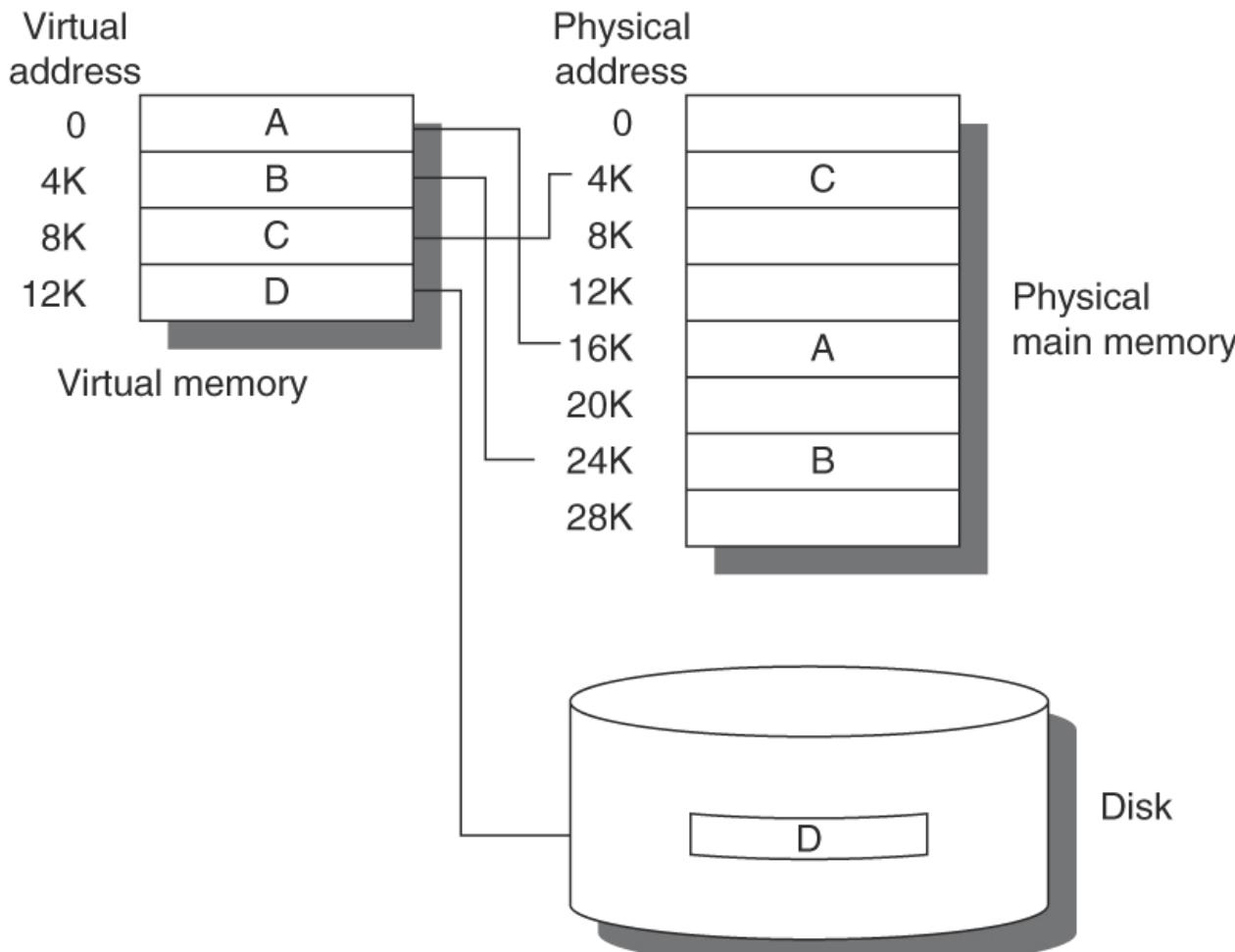


Virtual Memory



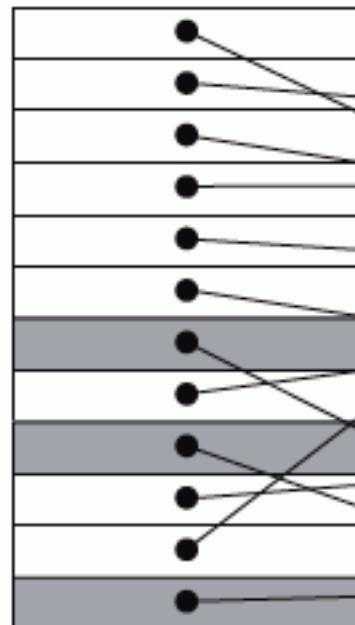
- Main Memory acts like a cache for the secondary memory (disk)
- Pages: virtual memory blocks
- Page faults
 - The data is not in main memory → retrieve it from disk
 - Huge miss penalty, thus pages should be fairly large (e.g., 4 KB)
 - Reducing page faults is important (LRU is worth the price)
 - Can handle the faults in software instead of hardware
 - Overhead is small compared to the disk access time
 - Using write-through is too expensive so write back is used

Virtual Memory



The logical program – contiguous virtual address space: four pages A, B, C, and D. [1]

Virtual addresses



Address translation

Physical addresses

Physical page can be shared by 2 virtual addresses to share data or code.

Example: OS code shared by more programs

Disk addresses

The actual location of the blocks is in physical main memory and on the disk [1]

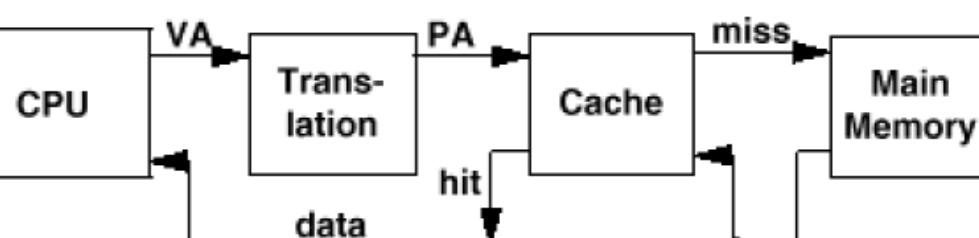
Mapping of pages from a virtual address to a physical address or disk address:

- Main memory acts as cache for secondary storage (disk)

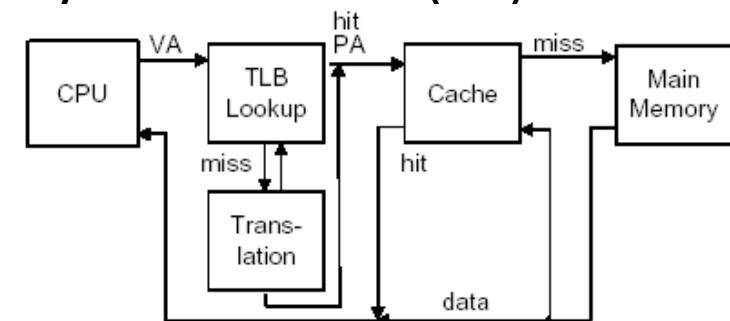
Virtual Memory – Address Translation



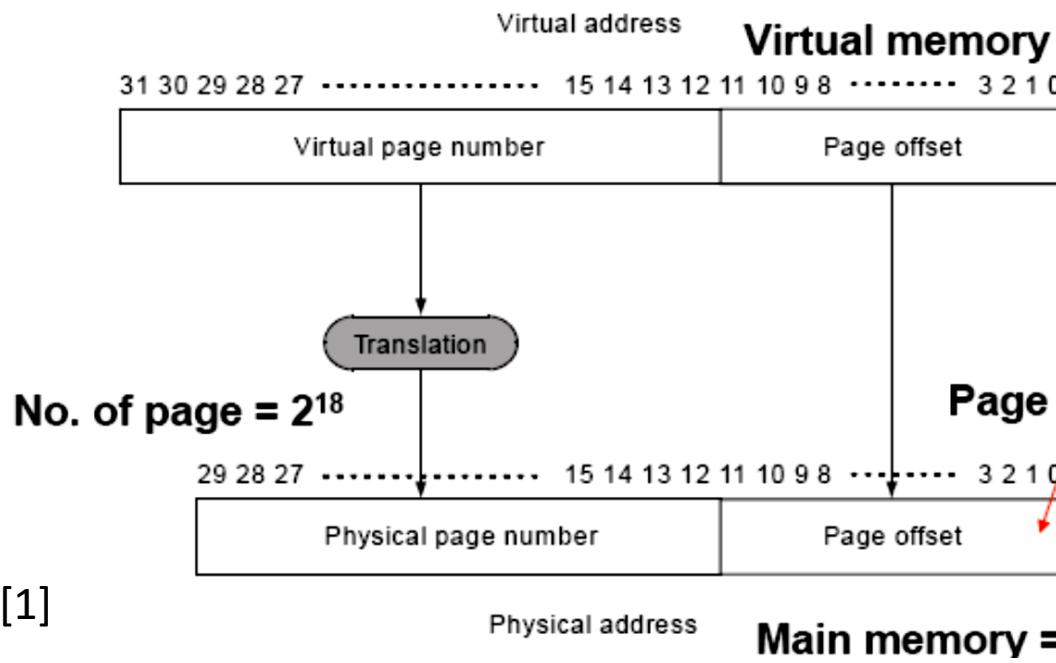
- Translation from virtual address (VA) to physical address (PA)



VA – PA translation with Page Tables



VA – PA translation with Page Tables + TLB

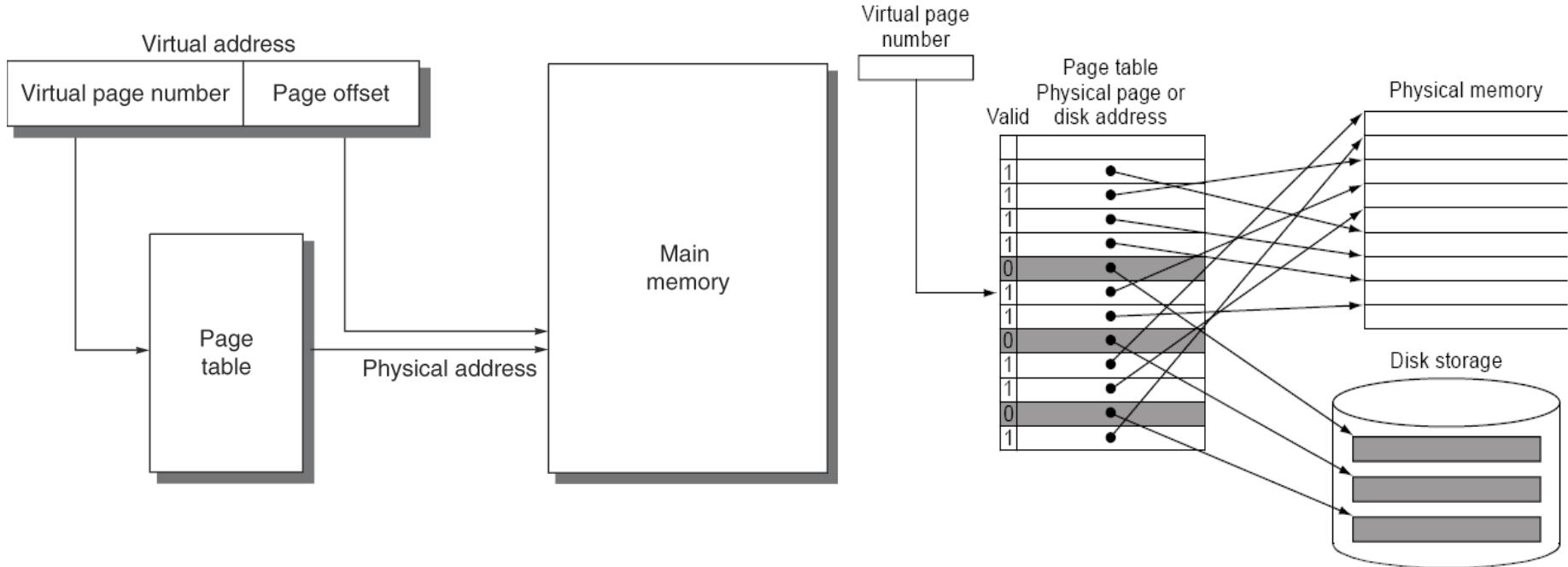


The number of bits in the page offset field determine the page size (4 KB)

Usually, number of virtual pages > number of physical pages



Virtual Memory – Page Table

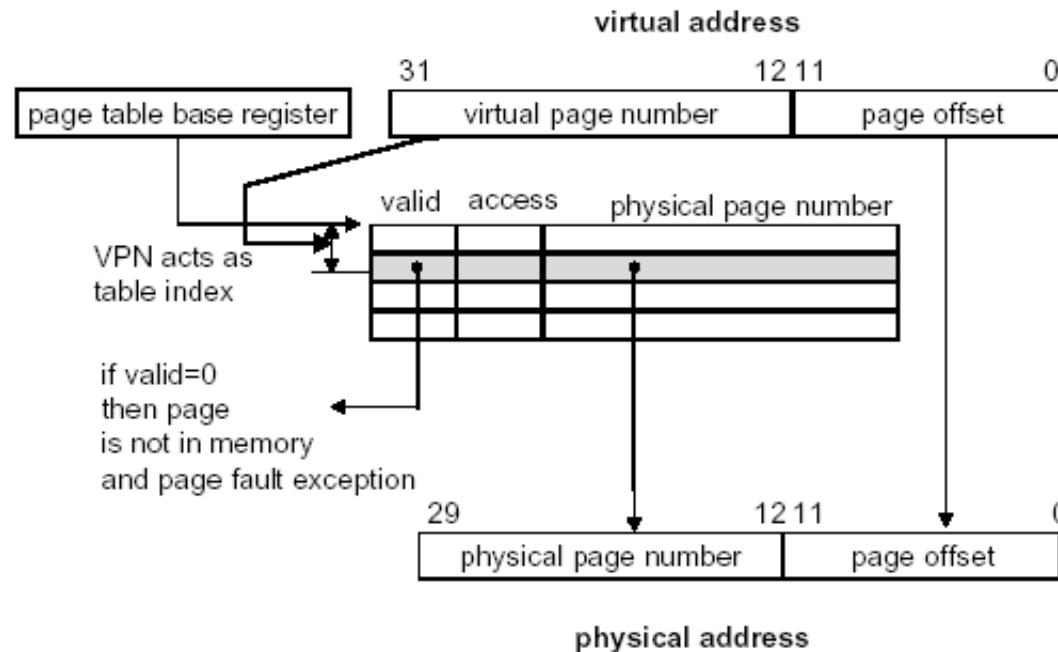


The mapping of a virtual address to a physical address via a page table [1]
Page table maps virtual page to either physical page or disk page

How Do You Place the Page and Find it Again?

- Locate pages by an index table: **page table**
- Each program has its own page table.
- A register (page table register) points to the start of the page table.
- The Page Table Implements Virtual to Physical Address Translation

Virtual Memory – Page Table



Address translation using the Page Table [1]

page size 4 KB, virtual address space 4 GB, physical memory 1 GB, VPN = 20 bits, PPN= 18 bits

To avoid large page table sizes:

- Each program has its own page table.
- Page table register points to start of program's page table.
- Other techniques – multiple-level page tables, hashing virtual address, etc.

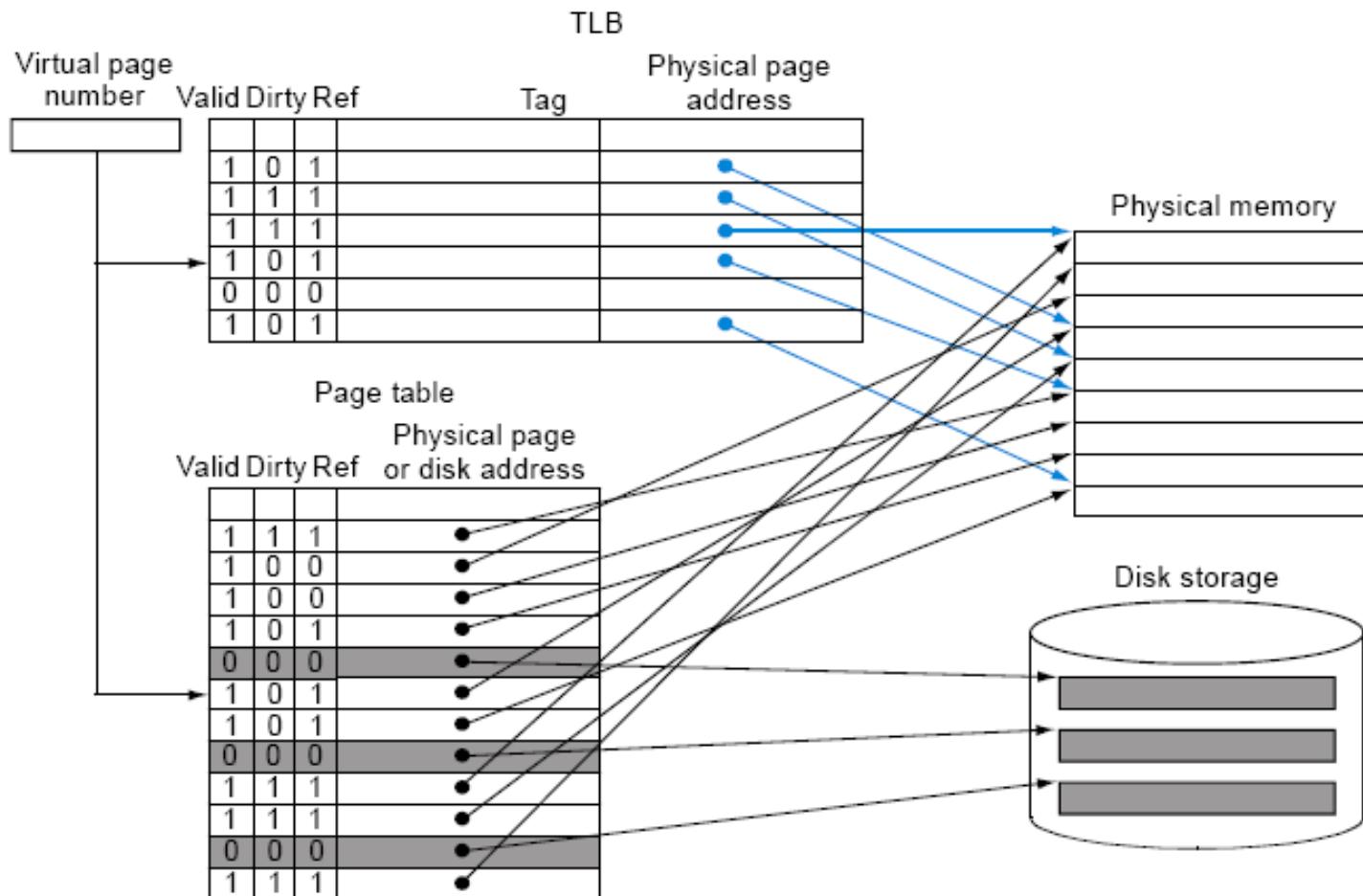


Virtual Memory



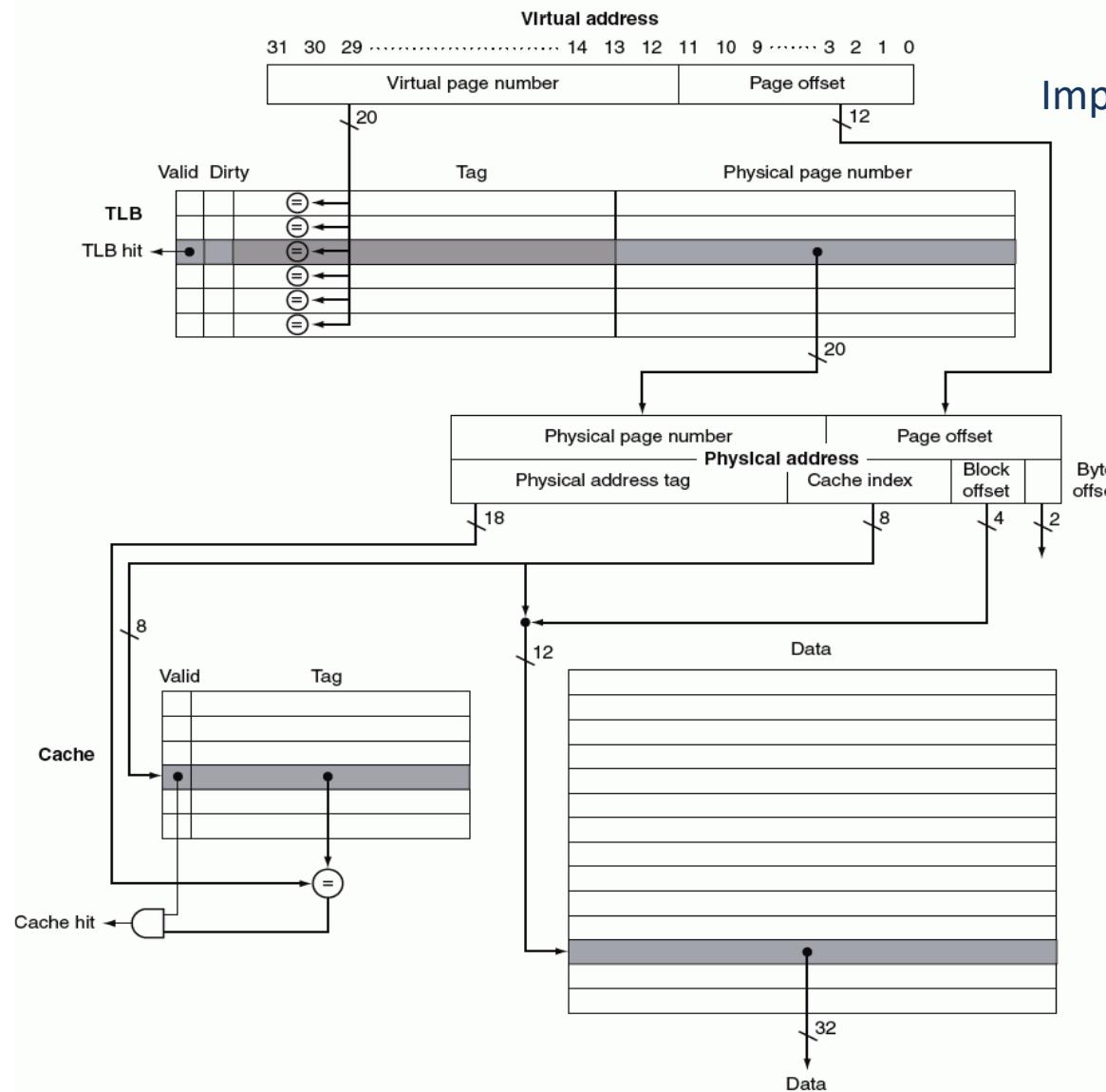
- Where to Place the Requested Pages? (in main memory)
 - If some pages are empty, use them
 - If all pages in main memory are in use, choose a page to replace it
 - LRU replacement (least recently used)
 - Replaced pages are written to swap space on the disk
- Making Address Translation Fast: TLB (translation look aside buffer)
 - Address translation mechanism – Slow
 - Two cycle memory access, the page tables are stored in main memory.
 - One memory access to obtain the physical address,
 - Second access to get the data
 - **TLB – cache for recently used Page Table Entries**

Virtual Memory – TLB



Translation with TLB: TLB – fully associative cache [1]

Virtual Memory – TLB



Fully associative TLB [1]

Implemented as a direct mapped cache
Data read – 16 words in a block

On a page reference, look up the virtual page number in the TLB

If TLB hit:

- Get the physical page number
- Turn on the reference bit
- Turn on the dirty bit if write

If TLB miss:

- Look up the page table
- If miss again then true page fault

TLB, Typical values:

- 16 – 512 entries
- Miss-rate: 0.01% – 1%.
- Miss-penalty: 10 – 100 cycles



Problems – Homework



- A CPU generates 32-bit addresses for a byte addressable memory. Design an 8 KB cache memory for this CPU (8 KB is the cache size only for the data; it does not include the tag). The block size is 32 bytes. Show the block diagram, and the address decoding for
 - direct mapped cache memory
 - 4-way set associative cache memory



References



1. D. A. Patterson and J. L. Hennessy, “Computer Organization and Design: A Quantitative Approach”, 5th edition, ed. Morgan-Kaufmann, 2011.
2. D. A. Patterson, J. L. Hennessy, “Computer Organization and Design: The Hardware/Software Interface”, 5th edition, ed. Morgan–Kaufmann, 2013.



Computer Architecture

Lecturer: Mihai Negru

2nd Year, Computer Science

Lecture 12: Modern CPU Architectures

<http://users.utcluj.ro/~negrum/>



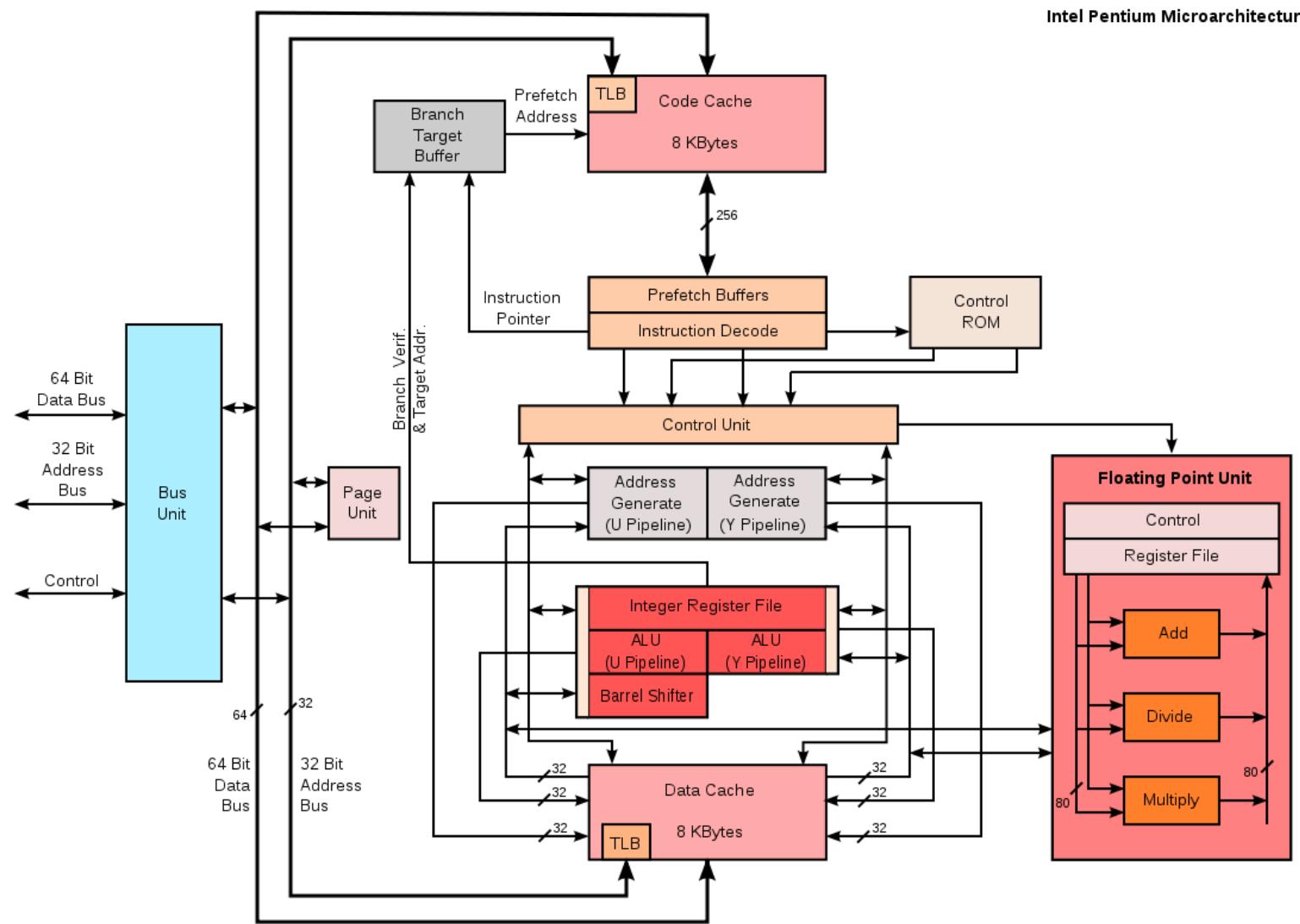
Intel Processor History – Single Cores



| Intel Processor | Year | Data | Technology | Instruction Set | CPU Clock |
|-----------------------|------|--------|----------------|----------------------------|-------------------|
| 8008 | 1972 | 8-bit | 10 µm | 8008 | 0.2 – 0.8 MHz |
| 8080 | 1974 | 8-bit | 6 µm | 8080 | 2 MHz |
| 8086 | 1976 | 16-bit | 3.2 µm | x86-16 | 5 – 10 MHz |
| 80186 | 1982 | 16-bit | 3.2 µm | x86-16 | 6 – 25 MHz |
| 80286 | 1982 | 16-bit | 1.5 µm | x86-16 | 6 – 25 MHz |
| 80386 | 1985 | 32-bit | 1 – 1.5 µm | x86 (IA-32) | 12 – 40 MHz |
| 80486 | 1989 | 32-bit | 0.6 – 1 µm | x86 (x87 – FP) | 16 – 150 MHz |
| Pentium (P5) | 1993 | 32-bit | 0.8 µm | IA-32 | 60 – 100 MHz |
| Pentium Pro (P6) | 1995 | 32-bit | 0.35 µm | IA-32 | 150 – 200 MHz |
| Pentium MMX | 1996 | 32-bit | 0.18 – 0.35 µm | IA-32, MMX | 120 – 233 MHz |
| Pentium II (P6) | 1997 | 32-bit | 0.35 µm | IA-32, MMX | 233 – 450 MHz |
| Pentium III (P6) | 1999 | 32-bit | 0.13 – 0.25 µm | IA-32, MMX, SSE | 450 MHz – 1.4 GHz |
| Pentium IV (Netburst) | 2000 | 32-bit | 0.18 µm | IA-32, MMX, SSE, SSE2, ... | 1.3 – 3.8 GHz |

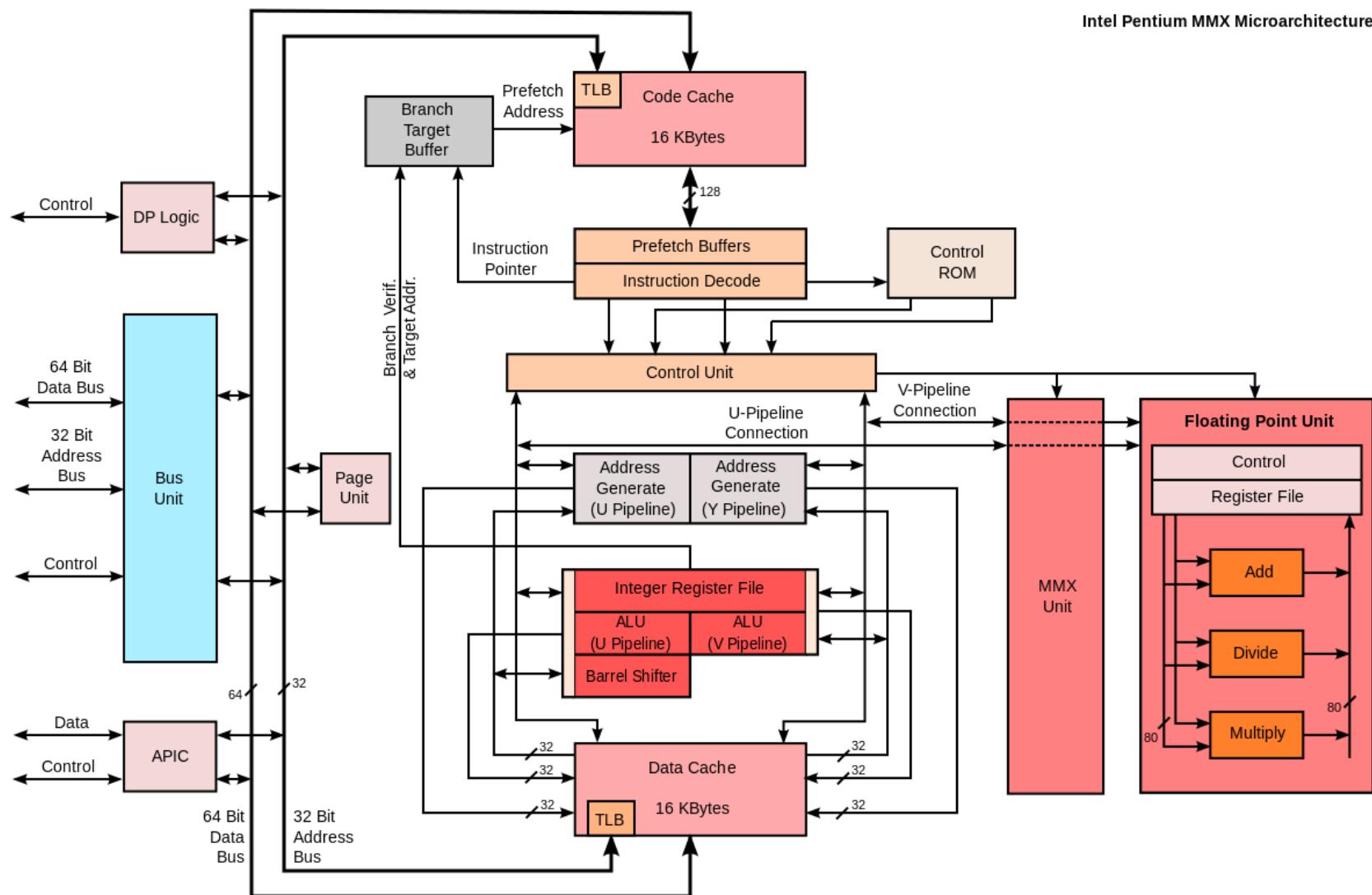


Intel Pentium MicroArchitecture





Intel Pentium MMX MicroArchitecture





NetBurst MicroArchitecture



| Pentium IV | Year | Data | Technology | Instruction Set | CPU Clock | Cores |
|--|------|--------|------------|------------------------|----------------|-------|
| Willamette | 2000 | 32-bit | 180 nm | IA-32, MMX, SSE, SSE2 | 1.3 – 2 GHz | 1 |
| Northwood | 2002 | 32-bit | 130 nm | IA-32, MMX, SSE, SSE2 | 1.6 – 3.06 GHz | 1 |
| Prescott | 2004 | 64-bit | 130 nm | MMX, ..., SSE3, x86-64 | 2.66 – 3.8 GHz | 1 |
| Hyper-Threading Technology – Virtual Cores | | | | | | |
| Northwood | 2003 | 32-bit | 130 nm | IA-32, MMX, SSE, SSE2 | 3.06 GHz | 1 |
| Prescott | 2005 | 64-bit | 90 nm | MMX, ..., SSE3, x86-64 | 2.66 – 3.8 GHz | 1 |
| Cedar Mill | 2006 | 64-bit | 65 nm | MMX, ..., SSE3, x86-64 | 3 – 3.6 GHz | 1 |

Major Issues:

- Large number of pipeline stages: 20 – 31
- power consumption
- heat dissipation



The End of the Single Core Era

| Pentium D | Year | Data | Technology | Instruction Set | CPU Clock | Cores |
|------------|------|--------|------------|------------------------|-----------------|-------|
| Smithfield | 2005 | 64-bit | 90 nm | MMX, ..., SSE3, x86-64 | 2.66 – 3.73 GHz | 2 |
| Pressler | 2006 | 64-bit | 65 nm | MMX, ..., SSE3, x86-64 | 2.66 – 3.73 GHz | 2 |



Intel Tick-Tock Era



- **Tick – Shrink**
 - Shrinking in the process technology of the previous Intel MicroArchitecture
 - Can introduce new instructions
 - Can make the processor more efficient, more powerful
- **Tock – Innovate**
 - a new MicroArchitecture
- Every 12 – 18 months → a new tick / tock
- **MicroArchitectures:**
 - Netburst, P6
 - Core
 - Nehalem
 - Sandy bridge
 - Haswell
 - Skylake



Intel Tick-Tock Era



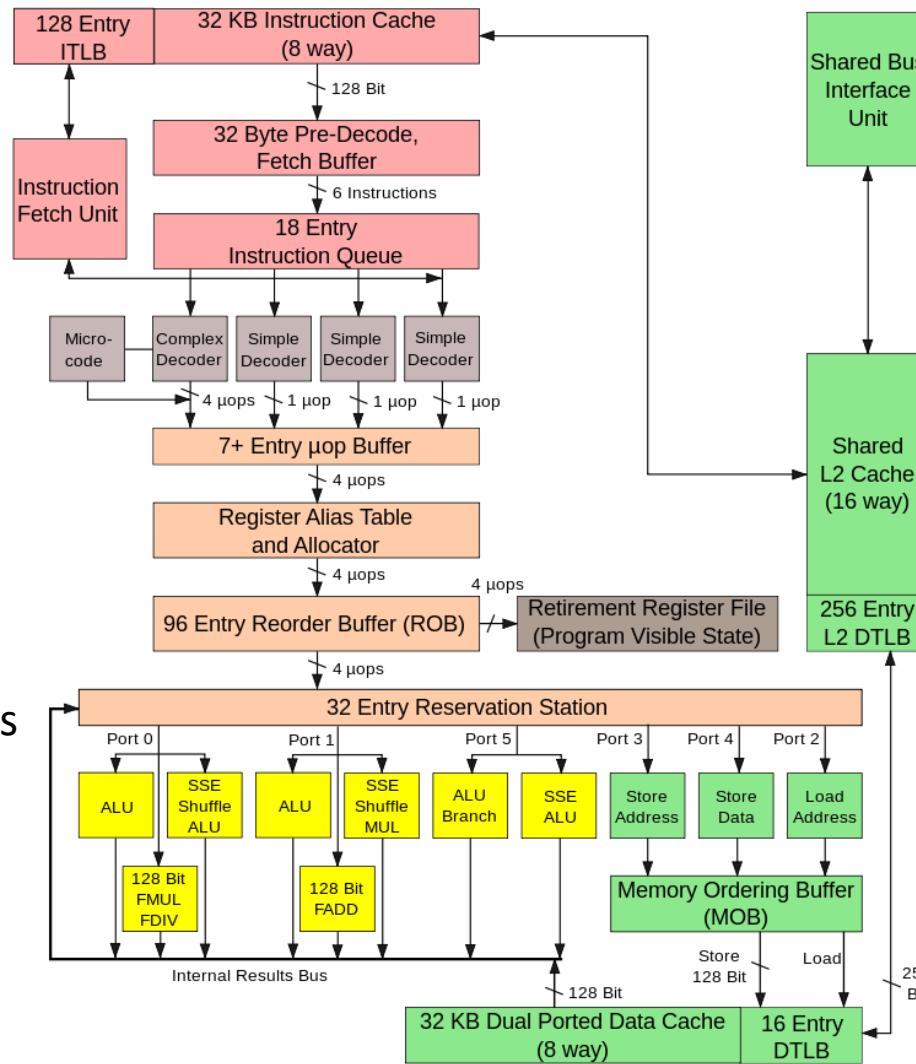
| Change | Technology | MicroArchitecture | Code Names | Year | Processors |
|--------|------------|-------------------|----------------------|------|--|
| Tick | 65 nm | P6, Netburst | Pressler, Cedar Mill | 2006 | Core, Pentium 4, Pentium D, Pentium M, Pentium Dual-Core |
| Tock | 65 nm | Core | Merom | 2006 | Core 2, Pentium Dual-Core |
| Tick | 45 nm | | Penryn | 2007 | |
| Tock | 45 nm | Nehalem | Nehalem | 2008 | Core i3, i5, i7, ... |
| Tick | 32 nm | | Westmere | 2010 | |
| Tock | 32 nm | Sandy Bridge | Sandy Bridge | 2011 | Core i3, i5, i7 – 2 nd , ... |
| Tick | 22 nm | | Ivy Bridge | 2012 | Core i3, i5, i7 – 3 rd , ... |
| Tock | 22 nm | Haswell | Haswell | 2013 | Core i3, i5, i7 – 4 th , ... |
| Tick | 14 nm | | Broadwell | 2014 | Core i3, i5, i7 – 5 th , ... |
| Tock | 14 nm | Skylake | Skylake | 2015 | Core i3, i5, i7 – 6 th , ... |
| Tock | 14 nm | | Kaby Lake | 2017 | Core i3, i5, i7 – 7 th , ... |
| Tock | 14 nm | | Coffee Lake | 2018 | Core i3, i5, i7 – 8 th , ... |
| Tick | 10 nm | | Cannonlake | 2019 | ?? |
| Tock | 10 nm | Ice Lake | Ice Lake | 2019 | ?? |



Intel Core 2 MicroArchitecture



- Number of Cores
 - 1, 2, 4
- No Hyper-threading
- 14-stage pipeline
- 4 decoders – 7 µOps
- Cache memory
 - L1: 32 KB data
 - L1: 32 KB instr.
 - L2: 256 KB
- Scheduler – 32 entries
6 µOperations / cycle



retirement bandwidth
4 µOperations / cycle

Intel Core 2 Architecture



Nehalem MicroArchitecture

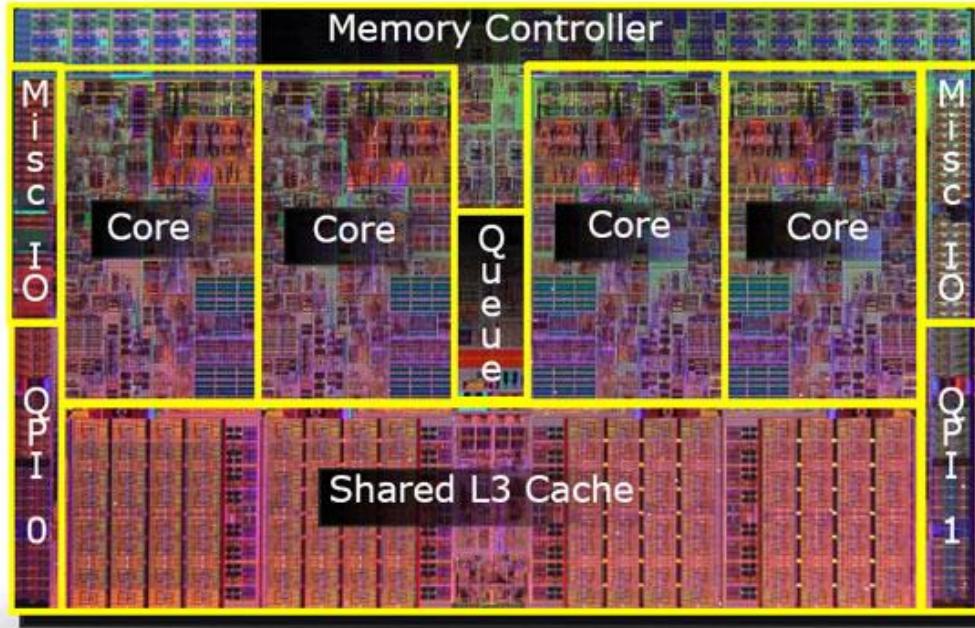


- Increased parallelism over Core MicroArchitecture
- Increased resources for higher performance
- Intel turbo Bust Technology – increases frequency when possible
 - Higher performance on demand
- Intel Hyper-Threading Technology – more threads than cores
- New Instructions – SSE 4.2

| Structure | Merom (Core) | Nehalem |
|---------------------|---------------------------------|---------------------------------|
| Reservation Station | 32 | 36 |
| Load buffers | 32 | 48 |
| Store buffers | 20 | 32 |
| Cache Layers | 2 | 3 |
| L1 cache | 32 KB instruction 32 KB data | 32 KB instruction 32 KB data |
| L2 cache | 256 KB instr+data | 256 KB instr+data |
| L3 cache – shared | None | Up to 8 MB |



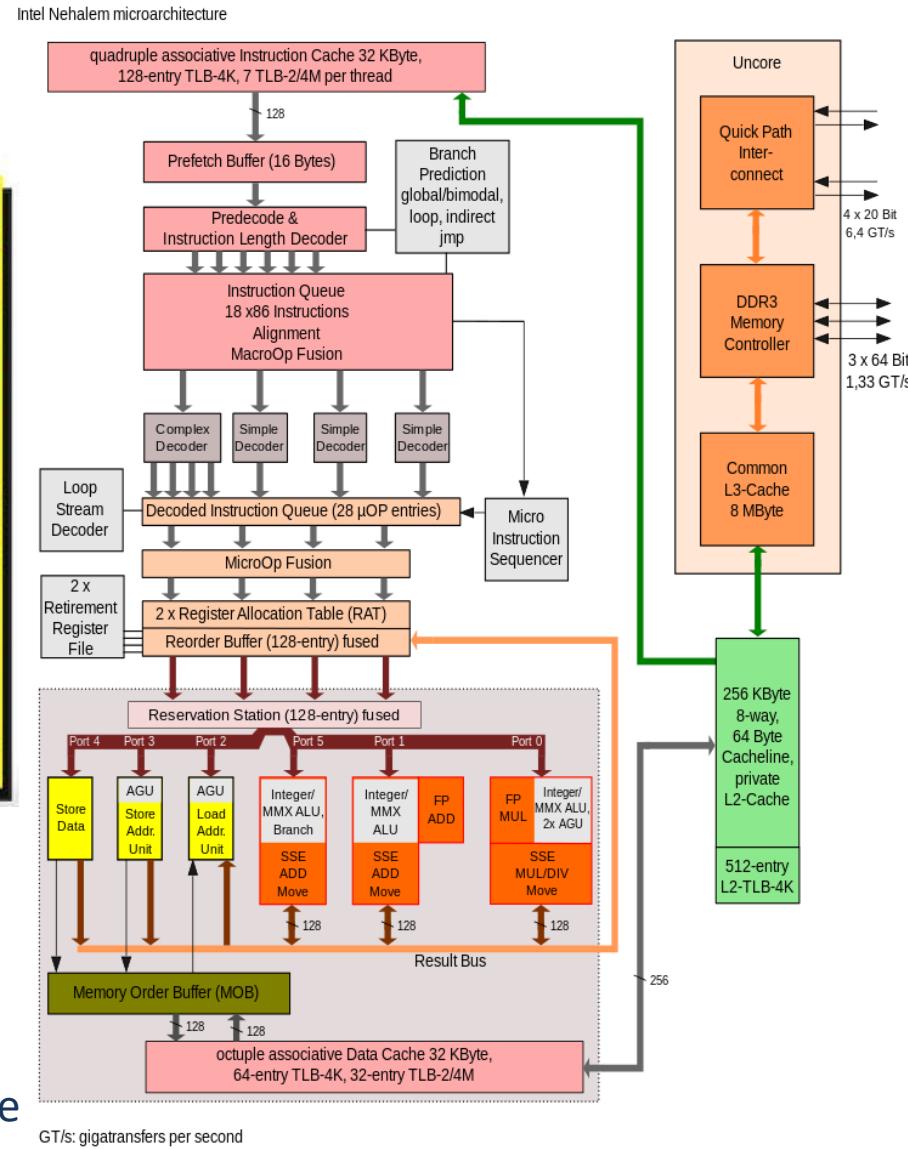
Nehalem MicroArchitecture



Nehalem MicroArchitecture

Part of the NorthBridge → CPU
QPI – Quick Path Interconnect

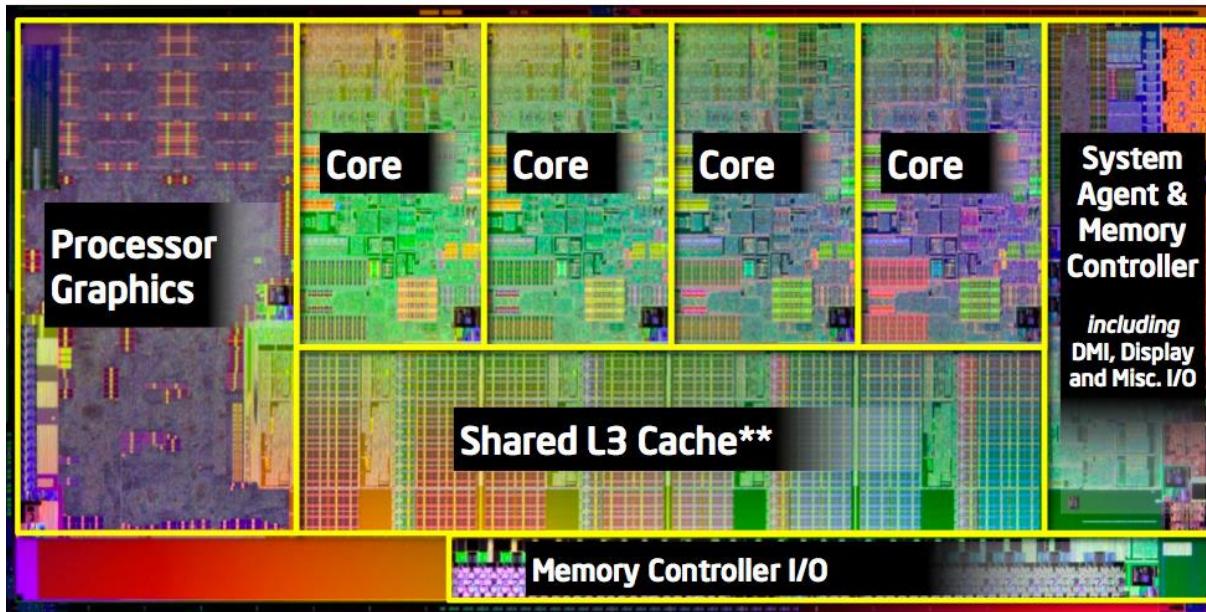
Nehalem's Core Architecture





Intel Sandy Bridge

Sandy Bridge
Tock – 32 nm

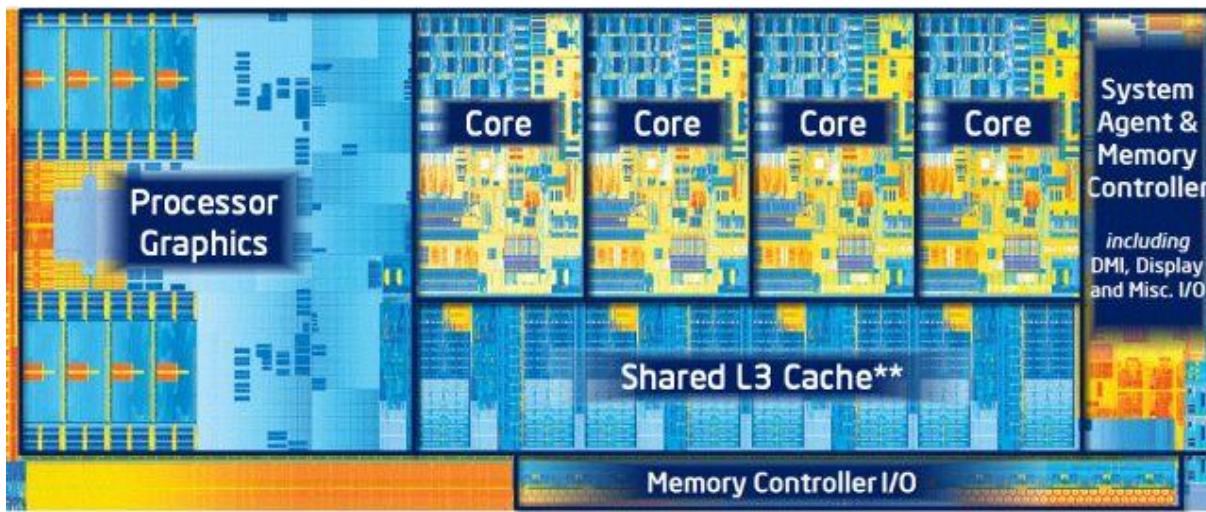


Memory
Controller
and
Graphics

inside CPU



Ivy Bridge
Tick – 22 nm



System on Chip
SoC



Intel Sandy Bridge



- Increased graphics performance
- Enhanced security capability
- New instructions – Advanced Vector Extension (AVX)
- Intel Dynamic Execution for Core
 - In-order issue
 - Speculative, super-scalar, out-of-order execution
 - 14 stage pipeline

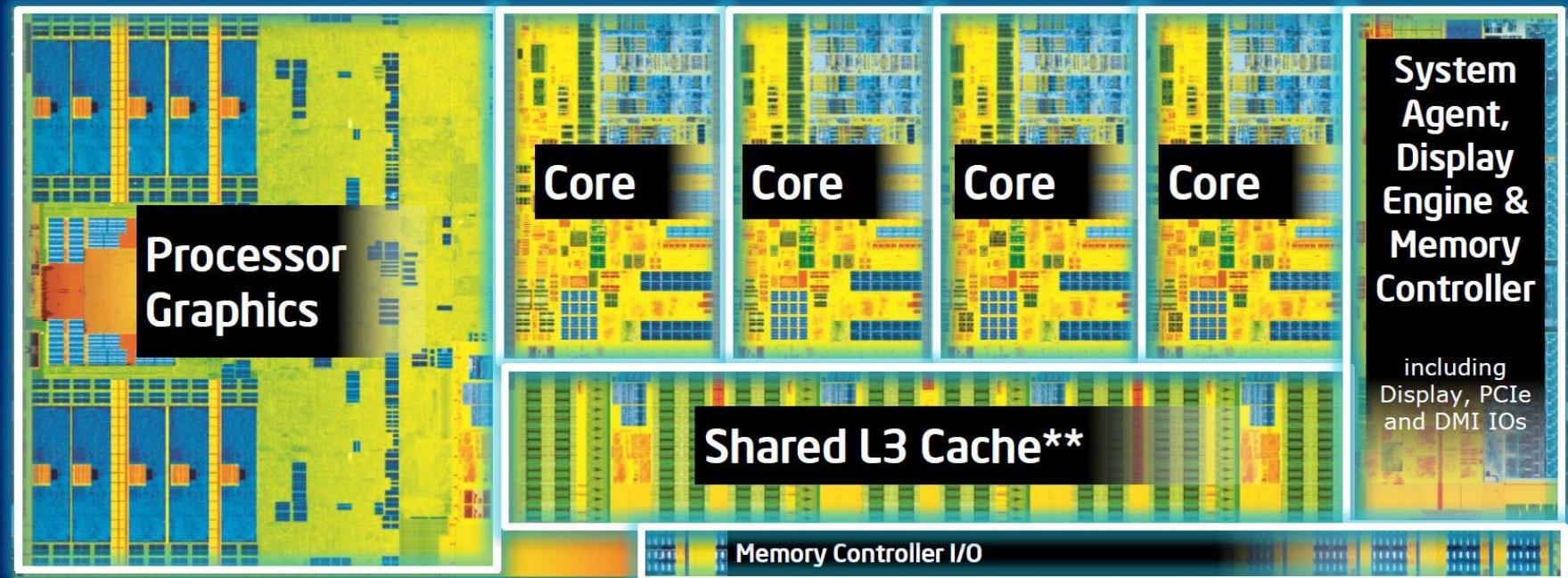
| Feature | Nehalem | Sandy Bridge |
|------------------------------|---------|--------------|
| Load Buffers | 48 | 64 |
| Store Buffers | 32 | 36 |
| Scheduler Entries | 36 | 54 |
| Integer Register File | N/A | 160 |
| Floating Point Register File | N/A | 144 |
| ROB Entries | 128 | 168 |



Intel Haswell MicroArchitecture



4th Generation Intel® Core™ Processor Die Map 22nm Tri-Gate 3-D Transistors



Quad core die shown above

Transistor count: 1.4 Billion

Die size: 177mm²

** Cache is shared across all 4 cores and processor graphics



Intel Haswell MicroArchitecture

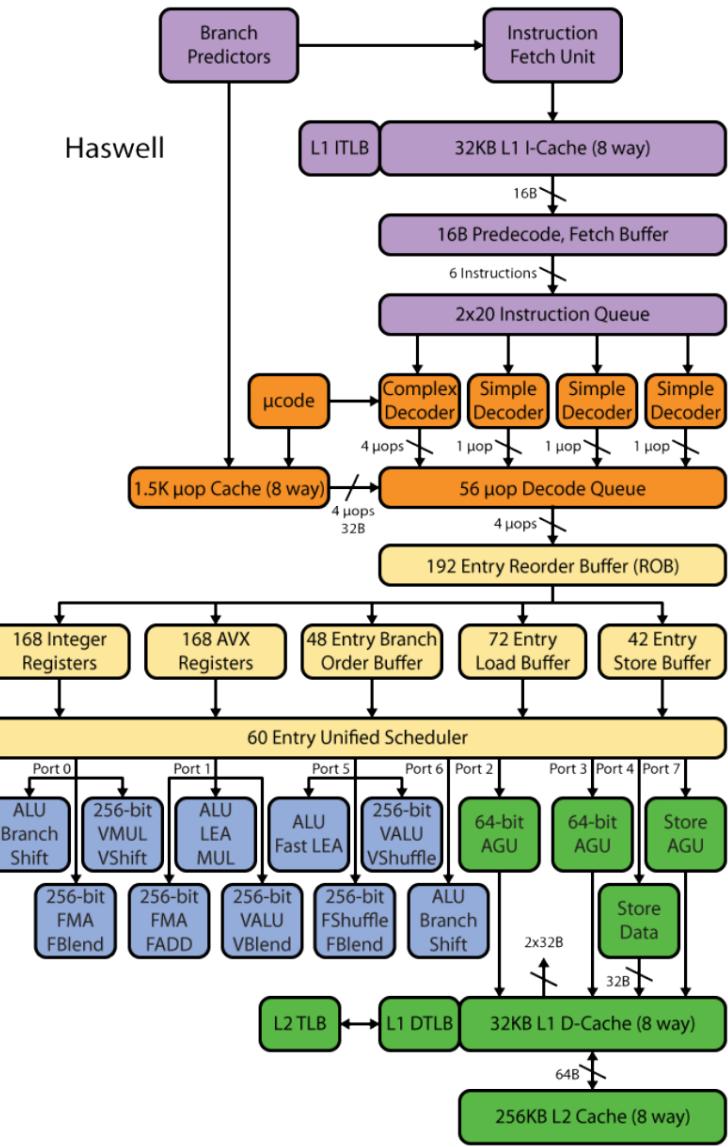
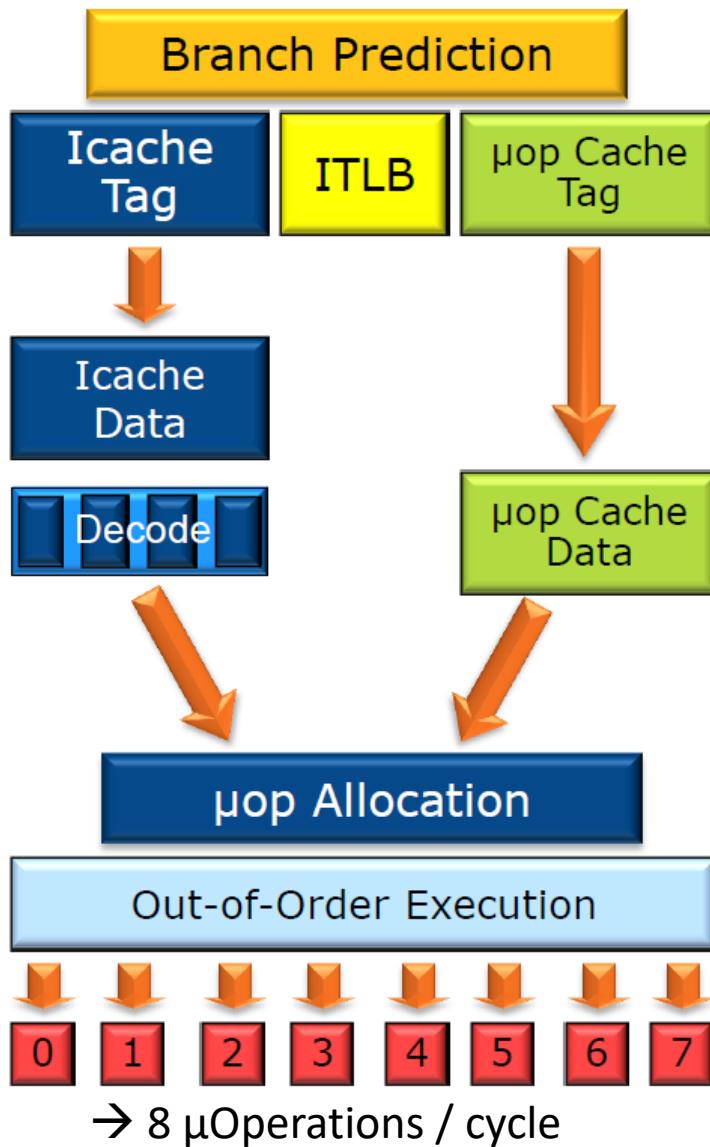


- More parallelism in every generation
- No pipeline growth – still 14 stages
- “Stacked” System on a Chip (SoC)
- Active idle – less power consumption, but instant resume
- New instructions – Advanced Vector Extension 2 (AVX2)

| Feature | Nehalem | Sandy Bridge | Haswell |
|------------------------------|-----------|--------------|-----------|
| Out-of-Order window (ROB) | 128 | 168 | 192 |
| Load Buffers | 48 | 64 | 72 |
| Store Buffers | 32 | 36 | 42 |
| Scheduler Entries | 36 | 54 | 60 |
| Integer Register File | N/A | 160 | 168 |
| Floating Point Register File | N/A | 144 | 168 |
| Allocation Queue | 28/thread | 28/thread | 56/thread |



Intel Haswell MicroArchitecture

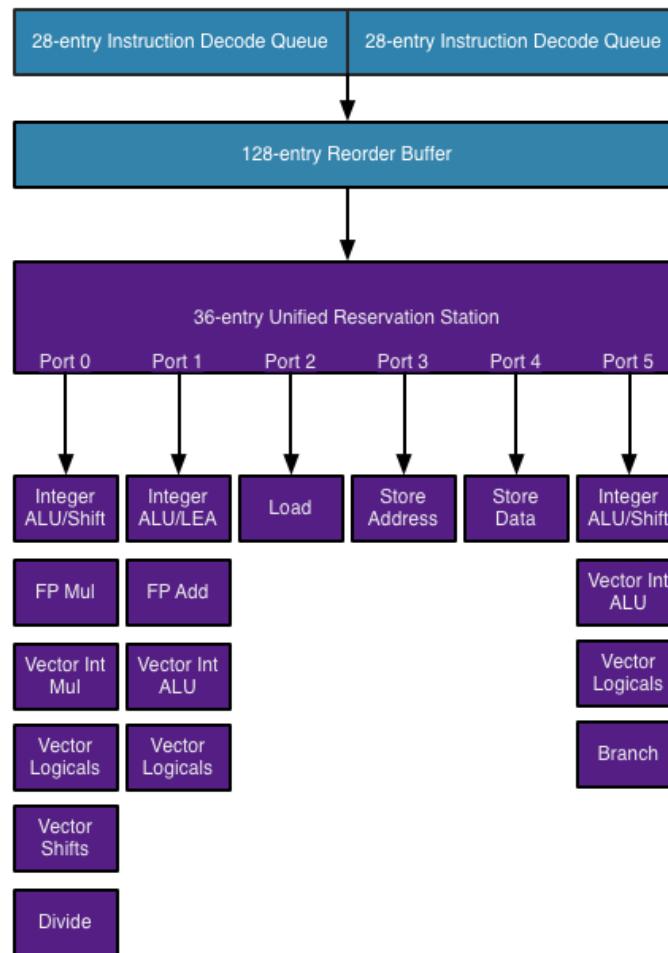




Execution Engine – Nehalem



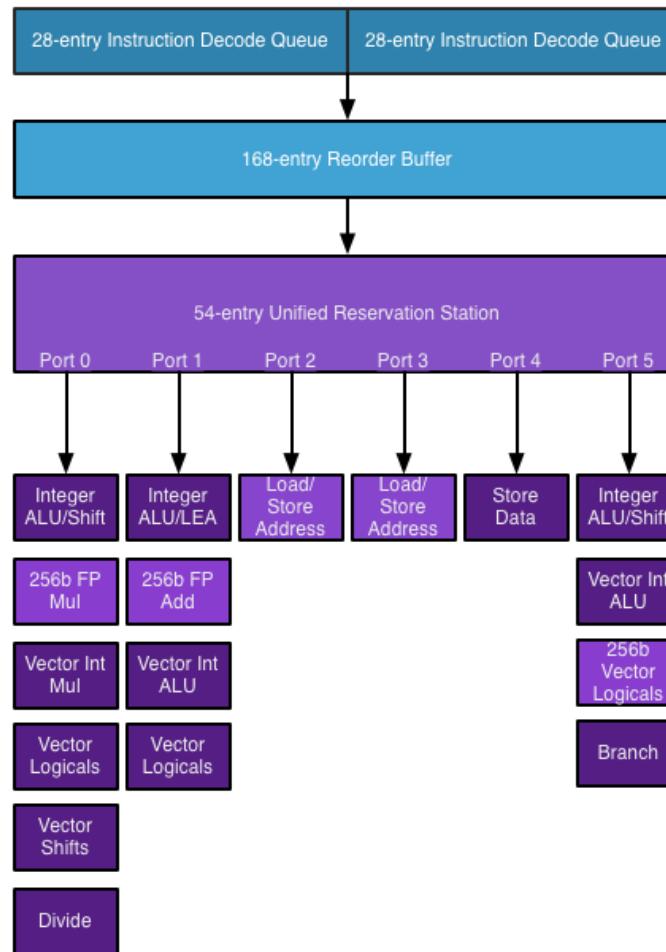
Intel Nehalem Execution Engine





Execution Engine – Sandy Bridge

Intel Sandy Bridge Execution Engine

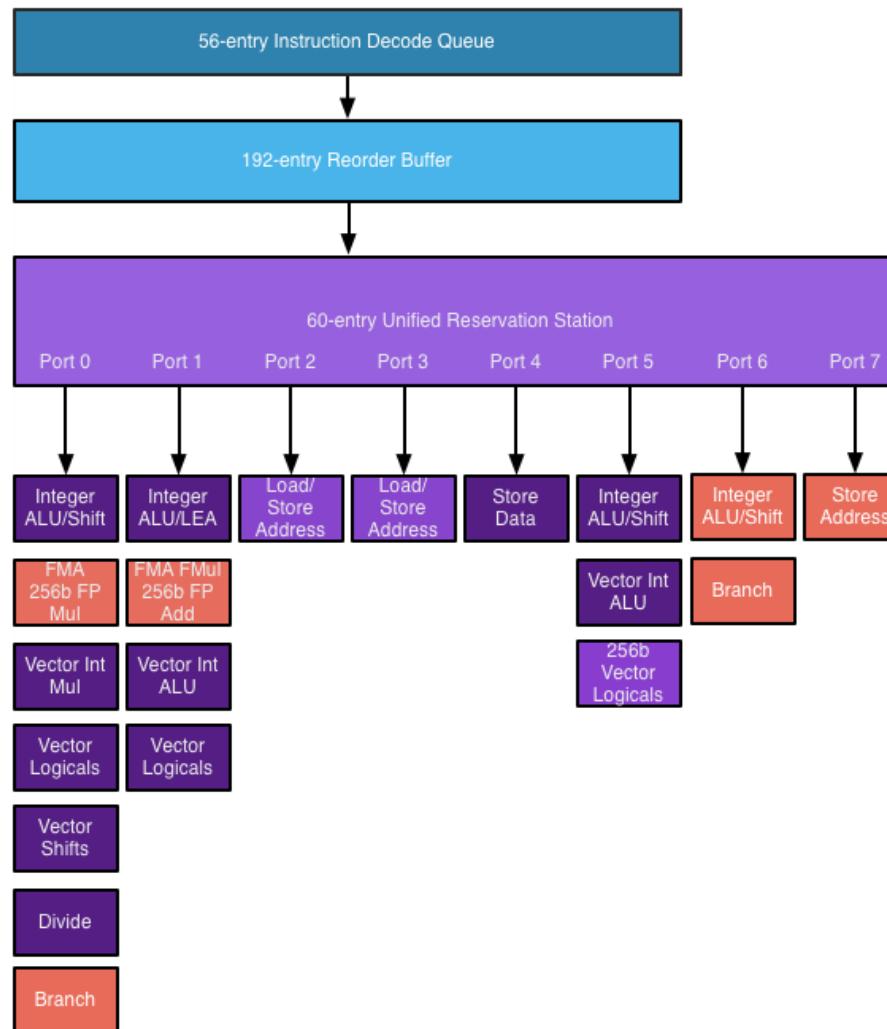




Execution Engine – Haswell



Intel Haswell Execution Engine





Intel Skylake MicroArchitecture



- More parallelism in every generation
- No pipeline growth – still 14 stages
- 4-way associative layer 2 cache!
- Improved Hyper-Threading performance
- Improved Branch Predictor – higher capacity, AVX2

| Feature | Nehalem | Sandy Bridge | Haswell | Skylake |
|------------------------------|-----------|--------------|-----------|-----------|
| Out-of-Order window (ROB) | 128 | 168 | 192 | 224 |
| Load Buffers | 48 | 64 | 72 | 72 |
| Store Buffers | 32 | 36 | 42 | 56 |
| Scheduler Entries | 36 | 54 | 60 | 97 |
| Integer Register File | N/A | 160 | 168 | 180 |
| Floating Point Register File | N/A | 144 | 168 | 168 |
| Allocation Queue | 28/thread | 28/thread | 56/thread | 64/thread |



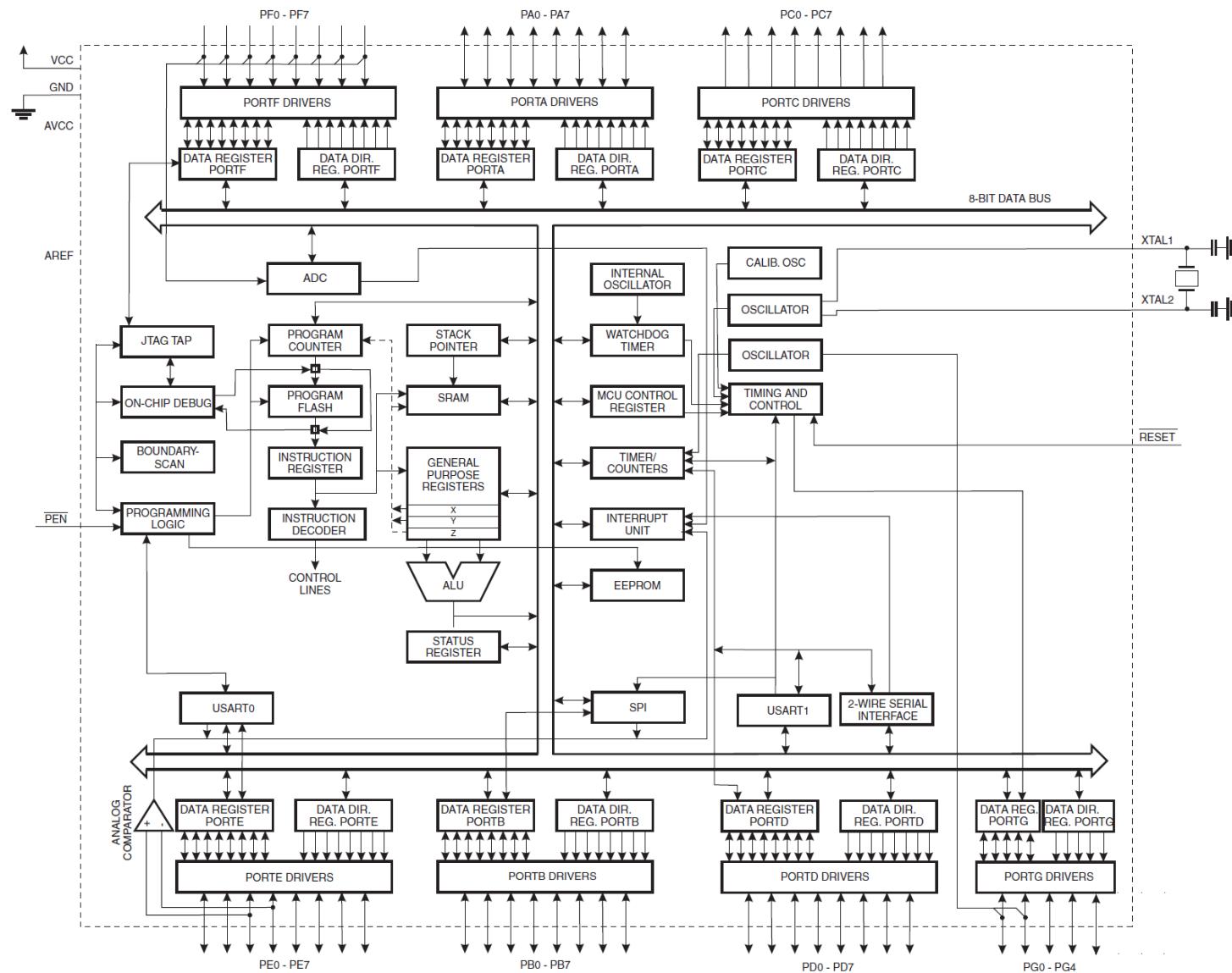
Other Architectures



- Micro-Controllers
- FPGAs
- SoC
- GPUs
-

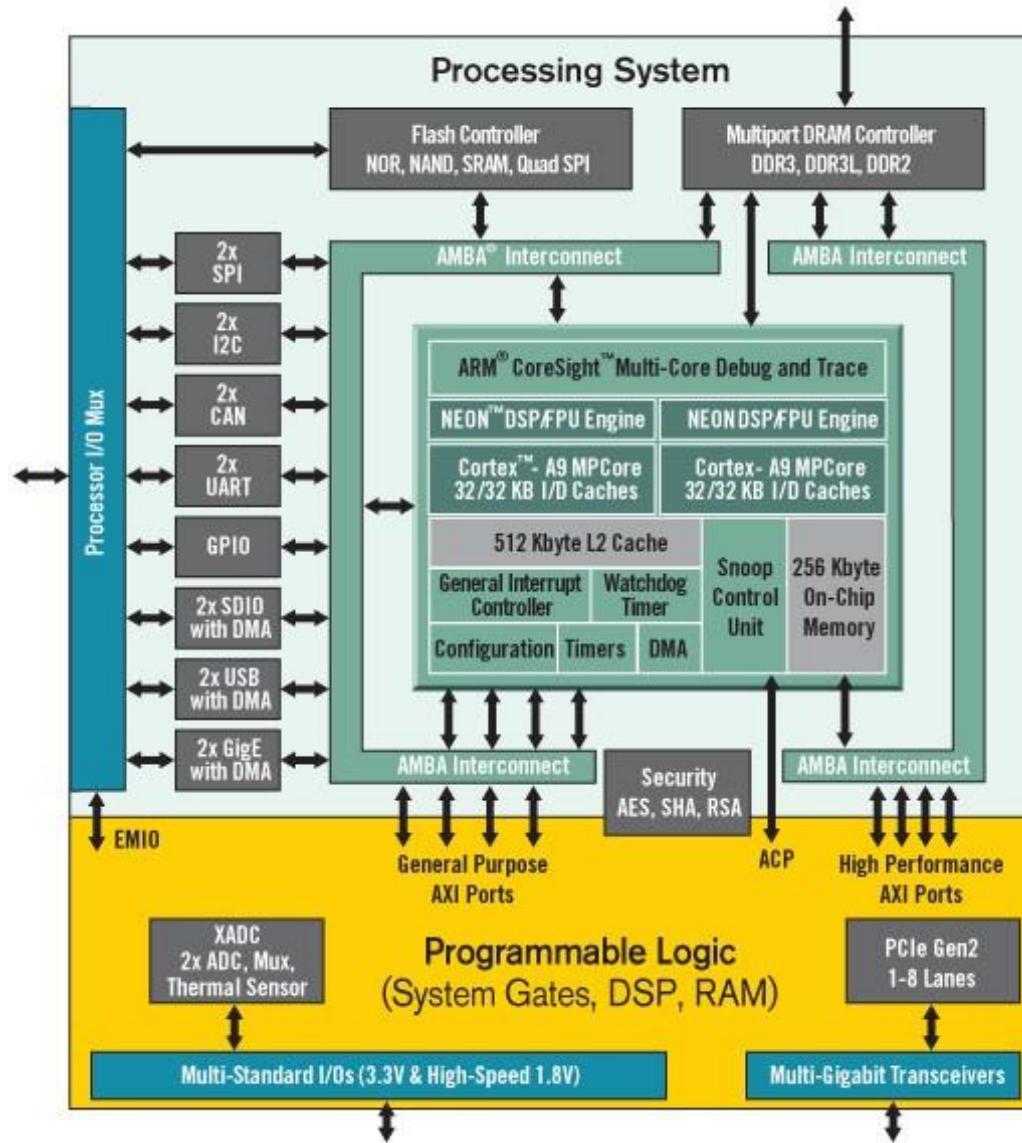


ATMEL ATMega 64 Micro-Controller





Xilinx ZYNQ – SoC





LEARN FOR THE EXAM!
IT'S NOT GOING TO BE EASY!



References



1. Intel presentations available on the web: <http://www.intel.com/>
2. http://en.wikipedia.org/wiki/List_of_Intel_microprocessors
3. <http://www.hardwaresecrets.com/article/Inside-Intel-Nehalem-Microarchitecture/535/1>
4. http://www.slideshare.net/am_sharifian/nehalem-microarchitecture
5. <http://research.engineering.wustl.edu/~songtian/pdf/intel-haswell.pdf>
6. http://www.hotchips.org/wp-content/uploads/hc_archives/hc25/HC25.80-Processors2-epub/HC25.27.820-Haswell-Hammarlund-Intel.pdf
7. <http://www.anandtech.com/>