Readings

- **This week**
  - Introduction to microarchitecture and single-cycle microarchitecture
    - P&P, Appendices A and C
    - H&H, Chapter 7.1-7.3, 7.6

- **Next week**
  - Multi-cycle microarchitecture
    - P&P, Appendices A and C
    - H&H, Chapter 7.4
  - Microprogramming
    - P&P, Appendices A and C
  - Pipelining
    - H&H, Chapter 7.5
Agenda for Today & Next Few Lectures

- Instruction Set Architectures (ISA): LC-3 and MIPS
- Assembly programming: LC-3 and MIPS
- Microarchitecture (principles & single-cycle uarch)
- Multi-cycle microarchitecture
- Microprogramming
- Pipelining
- Issues in Pipelining: Control & Data Dependence Handling, State Maintenance and Recovery, ...
- Out-of-Order Execution
Recall: The Von Neumann Model

**CONTROL UNIT**
- PC or IP
- Inst Register

**MEMORY**
- Mem Addr Reg
- Mem Data Reg

**PROCESSING UNIT**
- ALU
- TEMP

**INPUT**
- Keyboard, Mouse, Disk...

**OUTPUT**
- Monitor, Printer, Disk...

**INPUT**
- Keyboard, Mouse, Disk...

**OUTPUT**
- Monitor, Printer, Disk...
Recall: LC-3: A Von Neumann Machine

Figure 4.3  The LC-3 as an example of the von Neumann model
Recall: The Instruction Cycle

- FETCH
- DECODE
- EVALUATE ADDRESS
- FETCH OPERANDS
- EXECUTE
- STORE RESULT
Recall: The Instruction Set Architecture

- The ISA is the **interface between** what the **software** commands and what the **hardware** carries out.

- The ISA specifies:
  - The **memory organization**
    - Address space (LC-3: $2^{16}$, MIPS: $2^{32}$)
    - Addressability (LC-3: 16 bits, MIPS: 32 bits)
    - Word- or Byte-addressable
  
  - The **register set**
    - R0 to R7 in LC-3
    - 32 registers in MIPS

  - The **instruction set**
    - Opcodes
    - Data types
    - Addressing modes
    - Semantics of instructions
Microarchitecture

- An implementation of the ISA

- How do we implement the ISA?
  - We will discuss this for many lectures

- There can be many implementations of the same ISA
  - MIPS R2000, R10000, ...
  - Intel 80486, Pentium, Pentium Pro, Pentium 4, Kaby Lake, Coffee Lake, ...
(A Bit More on)
ISA Design and Tradeoffs
The Von Neumann Model/Architecture

- Also called *stored program computer* (instructions in memory). Two key properties:
  - Stored program
    - Instructions stored in a linear memory array
    - Memory is unified between instructions and data
      - The interpretation of a stored value depends on the control signals

- Sequential instruction processing
  - One instruction processed (fetched, executed, and completed) at a time
  - Program counter (instruction pointer) identifies the current instr.
  - Program counter is advanced sequentially except for control transfer instructions

When is a value interpreted as an instruction?
The Von Neumann Model/Architecture

- **Recommended reading**
  - Burks, Goldstein, von Neumann, “Preliminary discussion of the logical design of an electronic computing instrument,” 1946.

- **Required reading**
  - Patt and Patel book, Chapter 4, “The von Neumann Model”

- **Stored program**

- **Sequential instruction processing**
The Von Neumann Model (of a Computer)
The Von Neumann Model (of a Computer)

- Q: Is this the only way that a computer can operate?
- A: No.
- Qualified Answer: But, it has been the dominant way
  - i.e., the dominant paradigm for computing
  - for N decades
The Dataflow Model (of a Computer)

- Von Neumann model: An instruction is fetched and executed in **control flow order**
  - As specified by the **instruction pointer**
  - Sequential unless explicit control flow instruction

- Dataflow model: An instruction is fetched and executed in **data flow order**
  - i.e., when its operands are ready
  - i.e., there is **no instruction pointer**
  - Instruction ordering specified by data flow dependence
    - Each instruction specifies “who” should receive the result
    - An instruction can “fire” whenever all operands are received
  - Potentially many instructions can execute at the same time
    - Inherently more parallel
Consider a Von Neumann program

- What is the significance of the program order?
- What is the significance of the storage locations?

V <= a + b;
W <= b * 2;
x <= v - w
y <= v + w
z <= x * y

Which model is more natural to you as a programmer?
More on Data Flow

- In a data flow machine, a program consists of data flow nodes
  - A data flow node fires (fetched and executed) when all its inputs are ready
    - i.e. when all inputs have tokens

- Data flow node and its ISA representation
Data Flow Nodes

*Conditional

*Relational

*Barrier Synch
An Example Data Flow Program
ISA-level Tradeoff: Instruction Pointer

- Do we need an instruction pointer in the ISA?
  - Yes: Control-driven, sequential execution
    - An instruction is executed when the IP points to it
    - IP automatically changes sequentially (except for control flow instructions)
  - No: Data-driven, parallel execution
    - An instruction is executed when all its operand values are available (data flow)

- Tradeoffs: MANY high-level ones
  - Ease of programming (for average programmers)?
  - Ease of compilation?
  - Performance: Extraction of parallelism?
  - Hardware complexity?
ISA vs. Microarchitecture Level Tradeoff

- A similar tradeoff (control vs. data-driven execution) can be made at the microarchitecture level

- ISA: Specifies how the **programmer sees** the instructions to be executed
  - Programmer sees a sequential, control-flow execution order vs.
  - Programmer sees a data-flow execution order

- Microarchitecture: How the **underlying implementation actually executes** instructions
  - Microarchitecture can execute instructions in any order as long as it obeys the semantics specified by the ISA when making the instruction results visible to software
  - Programmer should see the order specified by the ISA
Let’s Get Back to the Von Neumann Model

- But, if you want to learn more about dataflow...


- A later lecture

- If you are really impatient:
  - [http://www.youtube.com/watch?v=D2uue7izU2c](http://www.youtube.com/watch?v=D2uue7izU2c)
The Von-Neumann Model

- All major *instruction set architectures* today use this model
  - x86, ARM, MIPS, SPARC, Alpha, POWER

- Underneath (at the microarchitecture level), the execution model of almost all *implementations (or, microarchitectures)* is very different
  - Pipelined instruction execution: *Intel 80486 uarch*
  - Multiple instructions at a time: *Intel Pentium uarch*
  - Out-of-order execution: *Intel Pentium Pro uarch*
  - Separate instruction and data caches

- But, what happens underneath that is *not consistent* with the von Neumann model is *not exposed* to software
  - Difference between ISA and microarchitecture
What is Computer Architecture?

- **ISA+implementation definition:** The science and art of designing, selecting, and interconnecting hardware components and designing the hardware/software interface to create a computing system that meets functional, performance, energy consumption, cost, and other specific goals.

- **Traditional (ISA-only) definition:** “The term architecture is used here to describe the attributes of a system as seen by the programmer, i.e., the conceptual structure and functional behavior as distinct from the organization of the dataflow and controls, the logic design, and the physical implementation.”

  *Gene Amdahl, IBM Journal of R&D, April 1964*
ISA vs. Microarchitecture

- **ISA**
  - Agreed upon interface between software and hardware
    - SW/compiler assumes, HW promises
  - What the software writer needs to know to write and debug system/user programs

- **Microarchitecture**
  - Specific implementation of an ISA
  - Not visible to the software

- **Microprocessor**
  - **ISA, uarch**, circuits
  - “Architecture” = ISA + microarchitecture
ISA vs. Microarchitecture

- What is part of ISA vs. Uarch?
  - Gas pedal: interface for “acceleration”
  - Internals of the engine: implement “acceleration”

- Implementation (uarch) can be various as long as it satisfies the specification (ISA)
  - Add instruction vs. Adder implementation
    - Bit serial, ripple carry, carry lookahead adders are all part of microarchitecture *(see H&H Chapter 5.2.1)*
  - x86 ISA has many implementations: 286, 386, 486, Pentium, Pentium Pro, Pentium 4, Core, Kaby Lake, Coffee Lake, ...

- Microarchitecture usually changes faster than ISA
  - Few ISAs (x86, ARM, SPARC, MIPS, Alpha) but many uarchs
  - *Why?*
ISA

- Instructions
  - Opcodes, Addressing Modes, Data Types
  - Instruction Types and Formats
  - Registers, Condition Codes

- Memory
  - Address space, Addressability, Alignment
  - Virtual memory management

- Call, Interrupt/Exception Handling
- Access Control, Priority/Privilege
- I/O: memory-mapped vs. instr.
- Task/thread Management
- Power and Thermal Management
- Multi-threading support, Multiprocessor support

...
Microarchitecture

- Implementation of the ISA under specific design constraints and goals
- Anything done in hardware without exposure to software
  - Pipelining
  - In-order versus out-of-order instruction execution
  - Memory access scheduling policy
  - Speculative execution
  - Superscalar processing (multiple instruction issue?)
  - Clock gating
  - Caching? Levels, size, associativity, replacement policy
  - Prefetching?
  - Voltage/frequency scaling?
  - Error correction?
Property of ISA vs. Uarch?

- ADD instruction’s opcode
- Booth multiplier vs. Wallace-tree multiplier
- Number of general purpose registers
- Number of cycles to execute the MUL instruction
- Number of ports to the register file
- Whether or not the machine employs pipelined instruction execution

Remember
  - Microarchitecture: Implementation of the ISA under specific design constraints and goals
Design Point

- A set of design considerations and their importance
  - leads to tradeoffs in both ISA and uarch
- Example considerations:
  - Cost
  - Performance
  - Maximum power consumption, thermal
  - Energy consumption (battery life)
  - Availability
  - Reliability and Correctness
  - Time to Market
  - Security, safety, predictability, …

- Design point determined by the “Problem” space (application space), the intended users/market
Application Space

Dream, and they will appear...


Many other workloads:
- Genome analysis
- Machine learning
- Robotics
- Web search
- Graph analytics

…
Increasingly Demanding Applications

Dream

and, they will come

As applications push boundaries, computing platforms will become increasingly strained.
Tradeoffs: Soul of Computer Architecture

- ISA-level tradeoffs

- Microarchitecture-level tradeoffs

- System and Task-level tradeoffs
  - How to divide the labor between hardware and software

- Computer architecture is the science and art of making the appropriate trade-offs to meet a design point
  - Why art?
Why Is It (Somewhat) Art?

- We do not (fully) know the future (applications, users, market)

![Diagram showing the relationship between problem, algorithm, program/language, runtime system, and user.](image)

- New demands from the top (Look Up)
- New issues and capabilities at the bottom (Look Down)
- New demands and personalities of users (Look Up)

- Microarchitecture
- ISA
- Runtime System (VM, OS, MM)
- Program/Language
- Algorithm
- Problem
- User
- Circuits
- Logic
- Electrons
- New demands and new issues and capabilities
Why Is It (Somewhat) Art?

And, the future is not constant (it changes)!

- Changing demands at the top (Look Up and Forward)
- Changing issues and capabilities at the bottom (Look Down and Forward)
- Changing demands and personalities of users (Look Up and Forward)
Analogue from Macro-Architecture

- Future is not constant in macro-architecture, either

- Example: Can a mill be later used as a theater + restaurant + conference room?
Originally built as a brewery in 1889, part of it was converted into a mill in 1913, and the other part into a cold store.

Nowadays is a center for a variety of activities: theater, conferences, restaurants, shops, museum...

http://www.muehle-tiefenbrunnen.ch/
Another Example (I)
Another Example (II)
Implementing the ISA: Microarchitecture Basics
Now That We Have an ISA

- How do we implement it?

- i.e., how do we design a system that obeys the hardware/software interface?

- Aside: “System” can be solely hardware or a combination of hardware and software
  - Remember “Translation of ISAs” (Transmeta example in Lec. 2)
  - A virtual ISA can be converted by “software” into an implementation ISA

- We will assume “hardware” for most lectures
How Does a Machine Process Instructions?

- What does processing an instruction mean?
- We will assume the von Neumann model (for now)

\[ AS = \text{Architectural (programmer visible) state before an instruction is processed} \]

\[ \text{Process instruction} \]

\[ AS' = \text{Architectural (programmer visible) state after an instruction is processed} \]

- Processing an instruction: Transforming AS to AS’ according to the ISA specification of the instruction
The Von Neumann Model/Architecture

Stored program

Sequential instruction processing
Recall: The Von Neumann Model

INPUT
Keyboard, Mouse, Disk…

OUTPUT
Monitor, Printer, Disk…

MEMORY
Mem Addr Reg
Mem Data Reg

PROCESSING UNIT
ALU
TEMP

CONTROL UNIT
PC or IP
Inst Register
The “Process Instruction” Step

- ISA specifies abstractly what AS’ should be, given an instruction and AS
  - It defines an abstract finite state machine where
    - State = programmer-visible state
    - Next-state logic = instruction execution specification
  - From ISA point of view, there are no “intermediate states” between AS and AS’ during instruction execution
    - One state transition per instruction

- Microarchitecture implements how AS is transformed to AS’
  - There are many choices in implementation
  - We can have programmer-invisible state to optimize the speed of instruction execution: **multiple** state transitions per instruction
    - Choice 1: AS $\rightarrow$ AS’ (transform AS to AS’ in a single clock cycle)
    - Choice 2: AS $\rightarrow$ AS+MS1 $\rightarrow$ AS+MS2 $\rightarrow$ AS+MS3 $\rightarrow$ AS’ (take multiple clock cycles to transform AS to AS’)
A Very Basic Instruction Processing Engine

- Each instruction takes a single clock cycle to execute
- Only combinational logic is used to implement instruction execution
  - No intermediate, programmer-invisible state updates

\[ AS = \text{Architectural (programmer visible) state at the beginning of a clock cycle} \]
\[ \text{Process instruction in one clock cycle} \]
\[ AS' = \text{Architectural (programmer visible) state at the end of a clock cycle} \]
A Very Basic Instruction Processing Engine

- Single-cycle machine

- What is the *clock cycle time* determined by?
- What is the *critical path* of the combinational logic determined by?
Recall: Programmer Visible (Architectural) State

Memory
array of storage locations
indexed by an address

Program Counter
memory address
of the current instruction

Registers
- given special names in the ISA
  (as opposed to addresses)
- general vs. special purpose

Instructions (and programs) specify how to transform the values of programmer visible state
Single-cycle vs. Multi-cycle Machines

- **Single-cycle machines**
  - Each instruction takes a single clock cycle
  - All state updates made at the end of an instruction’s execution
  - **Big disadvantage:** The slowest instruction determines cycle time → long clock cycle time

- **Multi-cycle machines**
  - Instruction processing broken into multiple cycles/stages
  - State updates can be made during an instruction’s execution
  - Architectural state updates made only at the end of an instruction’s execution
  - **Advantage over single-cycle:** The slowest “stage” determines cycle time

- Both single-cycle and multi-cycle machines literally follow the von Neumann model at the microarchitecture level
Instruction Processing “Cycle”

- Instructions are processed under the direction of a “control unit” step by step.
- Instruction cycle: Sequence of steps to process an instruction.
- Fundamentally, there are six steps:
  - Fetch
  - Decode
  - Evaluate Address
  - Fetch Operands
  - Execute
  - Store Result

- Not all instructions require all six steps (see P&P Ch. 4)
Recall: The Instruction Processing “Cycle”

- FETCH
- DECODE
- EVALUATE ADDRESS
- FETCH OPERANDS
- EXECUTE
- STORE RESULT
Instruction Processing “Cycle” vs. Machine Clock Cycle

- Single-cycle machine:
  - All six phases of the instruction processing cycle take a *single machine clock cycle* to complete

- Multi-cycle machine:
  - All six phases of the instruction processing cycle can take *multiple machine clock cycles* to complete
  - In fact, *each phase can take multiple clock cycles to complete*
Instruction Processing Viewed Another Way

- Instructions transform Data (AS) to Data’ (AS’)
- This transformation is done by functional units
  - Units that “operate” on data
- These units need to be told what to do to the data

An instruction processing engine consists of two components

- **Datapath**: Consists of hardware elements that deal with and transform data signals
  - functional units that operate on data
  - hardware structures (e.g. wires and muxes) that enable the flow of data into the functional units and registers
  - storage units that store data (e.g., registers)
- **Control logic**: Consists of hardware elements that determine control signals, i.e., signals that specify what the datapath elements should do to the data
Single-cycle vs. Multi-cycle: Control & Data

- Single-cycle machine:
  - Control signals are generated in the same clock cycle as the one during which data signals are operated on.
  - Everything related to an instruction happens in one clock cycle (serialized processing).

- Multi-cycle machine:
  - Control signals needed in the next cycle can be generated in the current cycle.
  - Latency of control processing can be overlapped with latency of datapath operation (more parallelism).

- We will see the difference clearly in microprogrammed multi-cycle microarchitectures.
Many Ways of Datapath and Control Design

- There are many ways of designing the data path and control logic

- Single-cycle, multi-cycle, pipelined datapath and control

- Single-bus vs. multi-bus datapaths

- Hardwired/combinational vs. microcoded/microprogrammed control
  - Control signals generated by combinational logic versus
  - Control signals stored in a memory structure

- Control signals and structure depend on the datapath design
Flash-Forward: Performance Analysis

- Execution time of an instruction
  - \(\{\text{CPI}\} \times \{\text{clock cycle time}\}\)

- Execution time of a program
  - Sum over all instructions \(\{\text{CPI}\} \times \{\text{clock cycle time}\}\)
  - \(\{\text{# of instructions}\} \times \{\text{Average CPI}\} \times \{\text{clock cycle time}\}\)

- Single-cycle microarchitecture performance
  - \(\text{CPI} = 1\)
  - Clock cycle time = long

- Multi-cycle microarchitecture performance
  - \(\text{CPI} = \text{different for each instruction}\)
    - Average CPI \(\rightarrow\) hopefully small
  - Clock cycle time = short

Here, we have two degrees of freedom to optimize independently
A Single-Cycle Microarchitecture
A Closer Look
Remember…

- Single-cycle machine
Let’s Start with the State Elements

- **Data and control inputs**

![Diagram showing data and control inputs with PC, Instruction memory, Instruction address, Instruction, Instruction memory, and ALU control with registers and ALU results.]

**Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]**
MIPS State Elements

- **Program counter:**
  32-bit register

- **Instruction memory:**
  Takes input 32-bit address A and reads the 32-bit data (i.e., instruction) from that address to the read data output RD.

- **Register file:**
  The 32-element, 32-bit register file has 2 read ports and 1 write port

- **Data memory:**
  Has a single read/write port. If the write enable, WE, is 1, it writes data WD into address A on the rising edge of the clock. If the write enable is 0, it reads address A onto RD.

This notation is used in H&H single-cycle MIPS implementation (H&H Chapter 7.3)
For Now, We Will Assume

- “Magic” memory and register file

- Combinational read
  - output of the read data port is a combinational function of the register file contents and the corresponding read select port

- Synchronous write
  - the selected register is updated on the positive edge clock transition when write enable is asserted
    - Cannot affect read output in between clock edges

- Single-cycle, synchronous memory
  - Contrast this with memory that tells when the data is ready
  - i.e., Ready bit: indicating the read or write is done
    - See P&P Appendix C (LC3-b) for multi-cycle memory
Instruction Processing

- 5 generic steps (P&H book)
  - Instruction fetch (IF)
  - Instruction decode and register operand fetch (ID/RF)
  - Execute/Evaluate memory address (EX/AG)
  - Memory operand fetch (MEM)
  - Store/writeback result (WB)

**Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]**
What Is To Come: The Full MIPS Datapath

**Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

JAL, JR, JALR omitted
Another Complete Single-Cycle Processor

Single-cycle processor. Harris and Harris, Chapter 7.3.
Single-Cycle Datapath for Arithmetic and Logical Instructions
R-Type ALU Instructions

- **R-type:** 3 register operands

MIPS assembly (e.g., register-register signed addition)

```
add $s0, $s1, $s2  #$s0=rd, $s1=rs, $s2=rt
```

**Machine Encoding**

<table>
<thead>
<tr>
<th>0</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>0</th>
<th>add (32)</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>6 bits</td>
</tr>
</tbody>
</table>

- **Semantics**

if MEM[PC] == add rd rs rt

GPR[rd] ← GPR[rs] + GPR[rt]

PC ← PC + 4
if MEM[PC] == ADD rd rs rt
GPR[rd] ← GPR[rs] + GPR[rt]
PC ← PC + 4
Example: ALU Design

- ALU operation \( (F_{2:0}) \) comes from the control logic

<table>
<thead>
<tr>
<th>( F_{2:0} )</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>A &amp; B</td>
</tr>
<tr>
<td>001</td>
<td>A</td>
</tr>
<tr>
<td>010</td>
<td>A + B</td>
</tr>
<tr>
<td>011</td>
<td>not used</td>
</tr>
<tr>
<td>100</td>
<td>A &amp; ( \sim B )</td>
</tr>
<tr>
<td>101</td>
<td>A</td>
</tr>
<tr>
<td>110</td>
<td>A - B</td>
</tr>
<tr>
<td>111</td>
<td>SLT</td>
</tr>
</tbody>
</table>
I-Type ALU Instructions

- **I-type:** 2 register operands and 1 immediate

MIPS assembly (e.g., register-immediate signed addition)

```
addi $s0, $s1, 5  #$s0=rt, $s1=rs
```

**Machine Encoding**

<table>
<thead>
<tr>
<th>addi (0)</th>
<th>rs</th>
<th>rt</th>
<th>immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>

**Semantics**

if MEM[PC] == addi rs rt immediate

- PC ← PC + 4
- GPR[rt] ← GPR[rs] + sign-extend(immediate)
Datapath for R and I-Type ALU Insts.

if MEM[PC] == ADDI rt rs immediate
  GPR[rt] ← GPR[rs] + sign-extend (immediate)
  PC ← PC + 4

**Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]**
Recall: ADD with one Literal in LC-3

- ADD assembly and machine code

**LC-3 assembly**

```
ADD R1, R4, # -2
```

**Field Values**

<table>
<thead>
<tr>
<th>OP</th>
<th>DR</th>
<th>SR</th>
<th>imm5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>-2</td>
</tr>
</tbody>
</table>

**Machine Code**

```
000 1 001 100 1 1111 10
```

![Diagram of register file and instruction register showing the process of adding register R1 and R4, with a literal of -2, and the resulting machine code. The diagram illustrates the flow of information from the instruction register to the ALU, with a sign-extend operation applied.]
Single-Cycle Datapath for Data Movement Instructions
Load Instructions

- **Load 4-byte word**

  **MIPS assembly**

  \[ \text{lw} \quad \$s3, \quad 8(\$s0) \quad \#\$s0=rs, \quad \$s3=rt \]

  **Machine Encoding**

  | op | rs=base | rt | imm=offset |
  | 31 | 26 | 25 | 21 | 20 | 16 | 15 | 0 |
  |-lw (35)| base| rt | offset |

  **I-Type**

- **Semantics**

  \[
  \begin{align*}
  \text{if MEM}[\text{PC}] &= \text{lw} \ \text{rt} \ \text{offset}_{16} (\text{base}) \\
  \text{PC} &\leftarrow \text{PC} + 4 \\
  \text{EA} &= \text{sign-extend}(\text{offset}) + \text{GPR}(\text{base}) \\
  \text{GPR}[\text{rt}] &\leftarrow \text{MEM}[ \text{translate}(\text{EA}) ]
  \end{align*}
  \]
if MEM[PC] == LW rt offset_{16} (base)  
EA = sign-extend(offset) + GPR[base]  
GPR[rt] ← MEM[ translate(EA) ]  
PC ← PC + 4
Store Instructions

- Store 4-byte word

MIPS assembly

\[
\text{sw} \quad $s3, \ 8($s0) \quad \#s0=rs, \ s3=rt
\]

Machine Encoding

<table>
<thead>
<tr>
<th>op</th>
<th>rs=base</th>
<th>rt</th>
<th>imm=offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>sw (43)</td>
<td>base</td>
<td>rt</td>
<td>offset</td>
</tr>
</tbody>
</table>

I-Type

- Semantics

if Mem[PC] == sw rt offset_{16} (base)
  PC ← PC + 4
  EA = sign-extend(offset) + GPR(base)
  MEM[ translate(EA) ] ← GPR[rt]
if MEM[PC]==SW rt offset\textsubscript{16} (base)  
EA = sign-extend(offset) + GPR[base]  
MEM[ translate(EA) ] ← GPR[rt]  
PC ← PC + 4
Load-Store Datapath

**Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]**
Datapath for Non-Control-Flow Insts.

**Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]**
Single-Cycle Datapath for Control Flow Instructions
Jump Instruction

- Unconditional branch or jump

```
j target
```

<table>
<thead>
<tr>
<th>j (2)</th>
<th>immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>26 bits</td>
</tr>
</tbody>
</table>

- 2 = opcode
- immediate (target) = target address

Semantics

if MEM[PC] == j immediate_{26}

\[
\text{target} = \{ \text{PC}^{\dagger}[31:28], \text{immediate}_{26}, 2'b00 \}
\]

PC ← target

\(^\dagger\) This is the incremented PC
Unconditional Jump Datapath

**Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

if MEM[PC]==J immediate26
PC = { PC[31:28], immediate26, 2’ b00 }

What about JR, JAL, JALR?
Other Jumps in MIPS

- **jal**: jump and link (function calls)
  - Semantics
    - if MEM[PC] == jal immediate_{26}
      - \$ra ← PC + 4
      - target = \{ PC \^{31:28}, immediate_{26}, 2'b00 \}
      - PC ← target

- **jr**: jump register
  - Semantics
    - if MEM[PC] == jr rs
      - PC ← GPR(rs)

- **jalr**: jump and link register
  - Semantics
    - if MEM[PC] == jalr rs
      - \$ra ← PC + 4
      - PC ← GPR(rs)

\^ This is the incremented PC
Aside: MIPS Cheat Sheet


- On the course website
Conditional Branch Instructions

- beq (Branch if Equal)

```plaintext
beq $s0, $s1, offset #$s0=rs,$s1=rt
```

<table>
<thead>
<tr>
<th>beq (4)</th>
<th>rs</th>
<th>rt</th>
<th>immediate=offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>

- Semantics (assuming no branch delay slot)

  if \( \text{MEM}[\text{PC}] == \text{beq} \) \( \text{rs} \) \( \text{rt} \) \( \text{immediate}_{16} \)
  
  target = \( \text{PC}^\dagger + \text{sign-extend(Immediate)} \times 4 \)
  
  if \( \text{GPR[rs]} == \text{GPR[rt]} \) then \( \text{PC} \leftarrow \text{target} \)
  
  else \( \text{PC} \leftarrow \text{PC} + 4 \)

- Variations: beq, bne, blez, bgtz

\( ^\dagger \text{This is the incremented PC} \)
Conditional Branch Datapath (for you to finish)

How to uphold the delayed branch semantics?
Putting It All Together

**Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

JAL, JR, JALR omitted
We did not cover the following slides in lecture. These are for your preparation for the next lecture.
Single-Cycle Control Logic
Single-Cycle Hardwired Control

- As combinational function of \( \text{Inst} = \text{MEM}[\text{PC}] \)

<table>
<thead>
<tr>
<th>Field</th>
<th>Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>opcode</td>
<td>6</td>
</tr>
<tr>
<td>rs</td>
<td>5</td>
</tr>
<tr>
<td>rt</td>
<td>5</td>
</tr>
<tr>
<td>rd</td>
<td>5</td>
</tr>
<tr>
<td>shamt</td>
<td>5</td>
</tr>
<tr>
<td>funct</td>
<td>6</td>
</tr>
</tbody>
</table>

**R-Type**

<table>
<thead>
<tr>
<th>Field</th>
<th>Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>opcode</td>
<td>6</td>
</tr>
<tr>
<td>rs</td>
<td>5</td>
</tr>
<tr>
<td>rt</td>
<td>5</td>
</tr>
<tr>
<td>immediate</td>
<td>16</td>
</tr>
</tbody>
</table>

**I-Type**

<table>
<thead>
<tr>
<th>Field</th>
<th>Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>opcode</td>
<td>6</td>
</tr>
<tr>
<td>immediate</td>
<td>26</td>
</tr>
</tbody>
</table>

**J-Type**

Consider

- All R-type and I-type ALU instructions
- \( \text{lw} \) and \( \text{sw} \)
- \( \text{beq} \), \( \text{bne} \), \( \text{blez} \), \( \text{bgtz} \)
- \( \text{j} \), \( \text{jr} \), \( \text{jal} \), \( \text{jalr} \)
## Single-Bit Control Signals

<table>
<thead>
<tr>
<th></th>
<th>When De-asserted</th>
<th>When asserted</th>
<th>Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>RegDest</td>
<td>GPR write select according to rt, i.e., inst[20:16]</td>
<td>GPR write select according to rd, i.e., inst[15:11]</td>
<td>opcode==0</td>
</tr>
<tr>
<td>ALUSrc</td>
<td>2(^{nd}) ALU input from 2(^{nd}) GPR read port</td>
<td>2(^{nd}) ALU input from sign-extended 16-bit immediate</td>
<td>(opcode!=0) &amp;&amp; (opcode!=BEQ) &amp;&amp; (opcode!=BNE)</td>
</tr>
<tr>
<td>MemtoReg</td>
<td>Steer ALU result to GPR write port</td>
<td>steer memory load to GPR wr. port</td>
<td>opcode==LW</td>
</tr>
<tr>
<td>RegWrite</td>
<td>GPR write disabled</td>
<td>GPR write enabled</td>
<td>(opcode!=SW) &amp;&amp; (opcode!=Bxx) &amp;&amp; (opcode!=J) &amp;&amp; (opcode!=JR))</td>
</tr>
</tbody>
</table>
# Single-Bit Control Signals

<table>
<thead>
<tr>
<th></th>
<th>When De-asserted</th>
<th>When asserted</th>
<th>Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>MemRead</td>
<td>Memory read disabled</td>
<td>Memory read port return load value</td>
<td>opcode==LW</td>
</tr>
<tr>
<td>MemWrite</td>
<td>Memory write disabled</td>
<td>Memory write enabled</td>
<td>opcode==SW</td>
</tr>
<tr>
<td>PCSrc(_1)</td>
<td>According to PCSrc(_2)</td>
<td>next PC is based on 26-bit immediate jump target</td>
<td>(opcode==J)</td>
</tr>
<tr>
<td>PCSrc(_2)</td>
<td>next PC = PC + 4</td>
<td>next PC is based on 16-bit immediate branch target</td>
<td>(opcode==Bxx) &amp;&amp; “bcond is satisfied”</td>
</tr>
</tbody>
</table>
ALU Control

- case opcode
  - ‘0’ ⇒ select operation according to funct
  - ‘ALUi’ ⇒ selection operation according to opcode
  - ‘LW’ ⇒ select addition
  - ‘SW’ ⇒ select addition
  - ‘Bxx’ ⇒ select bcond generation function
  - __ ⇒ don’t care

- Example ALU operations
  - ADD, SUB, AND, OR, XOR, NOR, etc.
  - bcond on equal, not equal, LE zero, GT zero, etc.
I-Type ALU

**Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]**
Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]
**Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]**
Some control signals are dependent on the processing of data.
**Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]**

Some control signals are dependent on the processing of data.
Jump

**Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]**
What is in That Control Box?

- **Combinational Logic** → **Hardwired Control**
  - Idea: Control signals generated combinatorially based on instruction
  - Necessary in a single-cycle microarchitecture

- **Sequential Logic** → **Sequential/Microprogrammed Control**
  - Idea: A memory structure contains the control signals associated with an instruction
  - Control Store
Review: Complete Single-Cycle Processor

**Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

JAL, JR, JALR omitted
Another Complete Single-Cycle Processor

Single-cycle processor. Harris and Harris, Chapter 7.3.
Extended Functionality: j

Single-cycle processor. Harris and Harris, Chapter 7.3.
## Control Unit

- **Control signals** generated by the decoder in control unit

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Op&lt;sub&gt;5:0&lt;/sub&gt;</th>
<th>RegWrite</th>
<th>RegDst</th>
<th>AluSrc</th>
<th>Branch</th>
<th>MemWrite</th>
<th>MemtoReg</th>
<th>ALUOp&lt;sub&gt;1:0&lt;/sub&gt;</th>
<th>Jump</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-type</td>
<td>000000</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>lw</td>
<td>100011</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>sw</td>
<td>101011</td>
<td>0</td>
<td>X</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>X</td>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>beq</td>
<td>000100</td>
<td>0</td>
<td>X</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>X</td>
<td>01</td>
<td>0</td>
</tr>
<tr>
<td>addi</td>
<td>001000</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>j</td>
<td>000010</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>0</td>
<td>X</td>
<td>XX</td>
<td>1</td>
</tr>
</tbody>
</table>

*Single-cycle processor*. Harris and Harris, Chapter 7.3.
Another Single-Cycle MIPS Processor (from H&H)
What to do with the Program Counter?

- The PC needs to be incremented by 4 during each cycle (for the time being).
- Initial PC value (after reset) is 0x00400000

```verilog
reg [31:0] PC_p, PC_n;       // Present and next state of PC

// [...]

assign PC_n <= PC_p + 4;     // Increment by 4;

always @ (posedge clk, negedge rst)
begin
    if (rst == '0') PC_p <= 32'h00400000;  // default
    else           PC_p <= PC_n;          // when clk
end
```
We Need a Register File

- **Store 32 registers, each 32-bit**
  - $2^5 = 32$, we need 5 bits to address each

- **Every R-type instruction uses 3 register**
  - Two for reading (RS, RT)
  - One for writing (RD)

- **We need a special memory with:**
  - 2 read ports (address x2, data out x2)
  - 1 write port (address, data in)
Register File

```vhdl
input [4:0] a_rs, a_rt, a_rd;
input [31:0] di_rd;
input we_rd;
output [31:0] do_rs, do_rt;

reg [31:0] R_arr [31:0]; // Array that stores regs

// Circuit description
assign do_rs = R_arr[a_rs]; // Read RS

assign do_rt = R_arr[a_rt]; // Read RT

always @ (posedge clk)
    if (we_rd) R_arr[a_rd] <= di_rd; // write RD
```
Register File

```verilog
input [4:0] a_rs, a_rt, a_rd;
input [31:0] di_rd;
input we_rd;
output [31:0] do_rs, do_rt;

reg [31:0] R_arr [31:0]; // Array that stores regs

// Circuit description; add the trick with $0
assign do_rs = (a_rs != 5'bx00000)? // is address 0?
  R_arr[a_rs] : 0; // Read RS or 0
assign do_rt = (a_rt != 5'bx00000)? // is address 0?
  R_arr[a_rt] : 0; // Read RT or 0

always @(posedge clk)
  if (we_rd) R_arr[a_rd] <= di_rd; // write RD
```
Data Memory Example

- Will be used to store the bulk of data

```verilog
input [15:0] addr; // Only 16 bits in this example
input [31:0] di;
input we;
output [31:0] do;

reg [65535:0] M_arr [31:0]; // Array for Memory

// Circuit description
assign do = M_arr[addr]; // Read memory
always @ (posedge clk)
    if (we) M_arr[addr] <= di; // write memory
```
Single-Cycle Datapath: `lw` fetch

- **STEP 1:** Fetch instruction

```
lw $s3, 1($0) # read memory word 1 into $s3
```

**I-Type**

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>imm</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>
**Single-Cycle Datapath: lw register read**

- **STEP 2:** Read source operands from register file

```
lw $s3, 1($0)  # read memory word 1 into $s3
```

**I-Type**

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>imm</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>
Single-Cycle Datapath: `lw` immediate

**STEP 3:** Sign-extend the immediate

`lw $s3, 1($0)`  # read memory word 1 into $s3

**I-Type**

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>imm</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>
Single-Cycle Datapath: `lw` address

**STEP 4:** Compute the memory address

```
lw $s3, 1($0)  # read memory word 1 into $s3
```

**I-Type**

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>imm</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>
Single-Cycle Datapath: `lw` memory read

**STEP 5:** Read from memory and write back to register file

```
lw $s3, 1(0)  # read memory word 1 into $s3
```

**I-Type**

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>imm</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>
Single-Cycle Datapath: lw PC increment

**STEP 6:** Determine address of next instruction

```
lw $s3, 1($0)  # read memory word 1 into $s3
```

**I-Type**

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>imm</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>
Single-Cycle Datapath: \texttt{sw}

- Write data in \texttt{rt} to memory

\texttt{sw} $\texttt{t7, 44}(\texttt{0})$ # write \texttt{t7} into memory address 44

\textbf{I-Type}

<table>
<thead>
<tr>
<th>\textbf{op}</th>
<th>\textbf{rs}</th>
<th>\textbf{rt}</th>
<th>\textbf{imm}</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>
Single-Cycle Datapath: R-type Instructions

- Read from rs and rt, write ALUResult to register file

\[
\text{add } t, b, c \quad \# t = b + c
\]

**R-Type**

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>6 bits</td>
</tr>
</tbody>
</table>
Single-Cycle Datapath: beq

Determine whether values in rs and rt are equal
Calculate BTA = (sign-extended immediate $\ll 2) + (PC+4)$

beq $s0, s1, target$  # branch is taken
Complete Single-Cycle Processor
Our MIPS Datapath has Several Options

- **ALU inputs**
  - Either RT or Immediate *(MUX)*

- **Write Address of Register File**
  - Either RD or RT *(MUX)*

- **Write Data In of Register File**
  - Either ALU out or Data Memory Out *(MUX)*

- **Write enable of Register File**
  - Not always a register write *(MUX)*

- **Write enable of Memory**
  - Only when writing to memory (sw) *(MUX)*

*All these options are our control signals*
Control Unit

- Control Unit
- Opcode
  - MemtoReg
  - MemWrite
  - Branch
  - ALUSrc
  - RegDst
  - RegWrite

- Main Decoder
- ALU Control
- ALU Operation
  - ALUOp (1:0)
  - ALUControl (2:0)

- ALU Decoder

- Table:
<table>
<thead>
<tr>
<th>ALUOp</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>add</td>
</tr>
<tr>
<td>01</td>
<td>subtract</td>
</tr>
<tr>
<td>10</td>
<td>look at funct field</td>
</tr>
<tr>
<td>11</td>
<td>n/a</td>
</tr>
</tbody>
</table>
ALU Does the Real Work in a Processor

<table>
<thead>
<tr>
<th>$F_{2:0}$</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>$A &amp; B$</td>
</tr>
<tr>
<td>001</td>
<td>$A \mid B$</td>
</tr>
<tr>
<td>010</td>
<td>$A + B$</td>
</tr>
<tr>
<td>011</td>
<td>not used</td>
</tr>
<tr>
<td>100</td>
<td>$A &amp; \neg B$</td>
</tr>
<tr>
<td>101</td>
<td>$A \mid \neg B$</td>
</tr>
<tr>
<td>110</td>
<td>$A - B$</td>
</tr>
<tr>
<td>111</td>
<td>SLT</td>
</tr>
</tbody>
</table>
ALU Internals

<table>
<thead>
<tr>
<th>$F_{2:0}$</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>$A &amp; B$</td>
</tr>
<tr>
<td>001</td>
<td>$A \mid B$</td>
</tr>
<tr>
<td>010</td>
<td>$A + B$</td>
</tr>
<tr>
<td>011</td>
<td>not used</td>
</tr>
<tr>
<td>100</td>
<td>$A &amp; \sim B$</td>
</tr>
<tr>
<td>101</td>
<td>$A \mid \sim B$</td>
</tr>
<tr>
<td>110</td>
<td>$A - B$</td>
</tr>
<tr>
<td>111</td>
<td>SLT</td>
</tr>
</tbody>
</table>
Control Unit: ALU Decoder

<table>
<thead>
<tr>
<th>ALUOp₁:₀</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Add</td>
</tr>
<tr>
<td>01</td>
<td>Subtract</td>
</tr>
<tr>
<td>10</td>
<td>Look at Funct</td>
</tr>
<tr>
<td>11</td>
<td>Not Used</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ALUOp₁:₀</th>
<th>Funct</th>
<th>ALUControl₂:₀</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>X</td>
<td>010 (Add)</td>
</tr>
<tr>
<td>X1</td>
<td>X</td>
<td>110 (Subtract)</td>
</tr>
<tr>
<td>1X</td>
<td>100000 (add)</td>
<td>010 (Add)</td>
</tr>
<tr>
<td>1X</td>
<td>100010 (sub)</td>
<td>110 (Subtract)</td>
</tr>
<tr>
<td>1X</td>
<td>100100 (and)</td>
<td>000 (And)</td>
</tr>
<tr>
<td>1X</td>
<td>100101 (or)</td>
<td>001 (Or)</td>
</tr>
<tr>
<td>1X</td>
<td>101010 (slt)</td>
<td>111 (SLT)</td>
</tr>
</tbody>
</table>
Let us Develop our Control Table

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Op$_{5:0}$</th>
<th>RegWrite</th>
<th>RegDst</th>
<th>AluSrc</th>
<th>MemWrite</th>
<th>MemtoReg</th>
<th>ALUOp</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **RegWrite**: Write enable for the register file
- **RegDst**: Write to register RD or RT
- **AluSrc**: ALU input RT or immediate
- **MemWrite**: Write Enable
- **MemtoReg**: Register data in from Memory or ALU
- **ALUOp**: What operation does ALU do
Let us Develop our Control Table

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Op&lt;sub&gt;5:0&lt;/sub&gt;</th>
<th>RegWrite</th>
<th>RegDst</th>
<th>AluSrc</th>
<th>MemWrite</th>
<th>MemtoReg</th>
<th>ALUOp</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-type</td>
<td>000000</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>funct</td>
</tr>
</tbody>
</table>

- **RegWrite:** Write enable for the register file
- **RegDst:** Write to register RD or RT
- **AluSrc:** ALU input RT or immediate
- **MemWrite:** Write Enable
- **MemtoReg:** Register data in from Memory or ALU
- **ALUOp:** What operation does ALU do
### Let us Develop our Control Table

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Op&lt;sub&gt;5:0&lt;/sub&gt;</th>
<th>RegWrite</th>
<th>RegDst</th>
<th>AluSrc</th>
<th>MemWrite</th>
<th>MemtoReg</th>
<th>ALUOp</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-type</td>
<td>000000</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>funct</td>
</tr>
<tr>
<td>lw</td>
<td>100011</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>add</td>
</tr>
</tbody>
</table>

- **RegWrite**: Write enable for the register file
- **RegDst**: Write to register RD or RT
- **AluSrc**: ALU input RT or immediate
- **MemWrite**: Write Enable
- **MemtoReg**: Register data in from Memory or ALU
- **ALUOp**: What operation does ALU do

---

**Note:**

The table above outlines the control signals for different instruction types in a processor. Each row represents a different instruction format, with columns indicating whether the operation involves writing to registers, memory, or performing ALU operations. The `op<sub>5:0</sub>` column specifies the instruction's operation code, while the other columns indicate the status of various control signals for the processor's execution stage.
Let us Develop our Control Table

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Op&lt;sub&gt;5:0&lt;/sub&gt;</th>
<th>RegWrite</th>
<th>RegDst</th>
<th>AluSrc</th>
<th>MemWrite</th>
<th>MemtoReg</th>
<th>ALUOp</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-type</td>
<td>000000</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>funct</td>
</tr>
<tr>
<td>lw</td>
<td>100011</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>add</td>
</tr>
<tr>
<td>sw</td>
<td>101011</td>
<td>0</td>
<td>X</td>
<td>1</td>
<td>1</td>
<td>X</td>
<td>add</td>
</tr>
</tbody>
</table>

- **RegWrite:** Write enable for the register file
- **RegDst:** Write to register RD or RT
- **AluSrc:** ALU input RT or immediate
- **MemWrite:** Write Enable
- **MemtoReg:** Register data in from Memory or ALU
- **ALUOp:** What operation does ALU do
### More Control Signals

<table>
<thead>
<tr>
<th>Instruction</th>
<th>$\text{Op}_{5:0}$</th>
<th>RegWrite</th>
<th>RegDst</th>
<th>AluSrc</th>
<th>Branch</th>
<th>MemWrite</th>
<th>MemtoReg</th>
<th>ALUOp</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-type</td>
<td>000000</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>funct</td>
</tr>
<tr>
<td>lw</td>
<td>100011</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>add</td>
</tr>
<tr>
<td>sw</td>
<td>101011</td>
<td>0</td>
<td>X</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>X</td>
<td>add</td>
<td></td>
</tr>
<tr>
<td>beq</td>
<td>000100</td>
<td>0</td>
<td>X</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>X</td>
<td>sub</td>
<td></td>
</tr>
</tbody>
</table>

- **New Control Signal**
  - **Branch**: Are we jumping or not?
Control Unit: Main Decoder

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Op&lt;sub&gt;5:0&lt;/sub&gt;</th>
<th>RegWrite</th>
<th>RegDst</th>
<th>AluSrc</th>
<th>Branch</th>
<th>MemWrite</th>
<th>MemtoReg</th>
<th>ALUOp&lt;sub&gt;1:0&lt;/sub&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-type</td>
<td>000000</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>lw</td>
<td>100011</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>00</td>
</tr>
<tr>
<td>sw</td>
<td>101011</td>
<td>0</td>
<td>X</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>X</td>
<td>00</td>
</tr>
<tr>
<td>beq</td>
<td>000100</td>
<td>0</td>
<td>X</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>X</td>
<td>01</td>
</tr>
</tbody>
</table>
Single-Cycle Datapath Example: or
Extended Functionality: addi

- No change to datapath
Control Unit: addi

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Op&lt;sub&gt;5:0&lt;/sub&gt;</th>
<th>RegWrite</th>
<th>RegDst</th>
<th>AluSrc</th>
<th>Branch</th>
<th>MemWrite</th>
<th>MemtoReg</th>
<th>ALUOp&lt;sub&gt;1:0&lt;/sub&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-type</td>
<td>000000</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>lw</td>
<td>100011</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>00</td>
</tr>
<tr>
<td>sw</td>
<td>101011</td>
<td>0</td>
<td>X</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>X</td>
<td>00</td>
</tr>
<tr>
<td>beq</td>
<td>000100</td>
<td>0</td>
<td>X</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>X</td>
<td>01</td>
</tr>
<tr>
<td>addi</td>
<td>001000</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>00</td>
</tr>
</tbody>
</table>
Extended Functionality: j
# Control Unit: Main Decoder

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Op&lt;sub&gt;5:0&lt;/sub&gt;</th>
<th>RegWrite</th>
<th>RegDst</th>
<th>AluSrc</th>
<th>Branch</th>
<th>MemWrite</th>
<th>MemtoReg</th>
<th>ALUOp&lt;sub&gt;1:0&lt;/sub&gt;</th>
<th>Jump</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-type</td>
<td>000000</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>lw</td>
<td>100011</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>sw</td>
<td>101011</td>
<td>0</td>
<td>X</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>X</td>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>beq</td>
<td>000100</td>
<td>0</td>
<td>X</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>X</td>
<td>01</td>
<td>0</td>
</tr>
<tr>
<td>j</td>
<td>000100</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>0</td>
<td>X</td>
<td>XX</td>
</tr>
</tbody>
</table>
A Bit More on
Performance Analysis
Processor Performance

How fast is my program?

- Every program consists of a series of instructions
- Each instruction needs to be executed.
Processor Performance

- How fast is my program?
  - Every program consists of a series of instructions
  - Each instruction needs to be executed.

- So how fast are my instructions?
  - Instructions are realized on the hardware
  - They can take one or more clock cycles to complete
  - \textit{Cycles per Instruction} = CPI
Processor Performance

- How fast is my program?
  - Every program consists of a series of instructions
  - Each instruction needs to be executed.

- So how fast are my instructions?
  - Instructions are realized on the hardware
  - They can take one or more clock cycles to complete
  - *Cycles per Instruction* = *CPI*

- How much time is one clock cycle?
  - The critical path determines how much time one cycle requires = *clock period*.
  - 1/clock period = *clock frequency* = how many cycles can be done each second.
Performance Analysis

- Execution time of an instruction
  - \( \text{CPI} \times \text{clock cycle time} \)

- Execution time of a program
  - Sum over all instructions \([\text{CPI} \times \text{clock cycle time}]\)
  - \( \text{# of instructions} \times \text{Average CPI} \times \text{clock cycle time} \)
Processor Performance

Now as a general formula

- Our program consists of executing $N$ instructions.
- Our processor needs CPI cycles for each instruction.
- The maximum clock speed of the processor is $f$, and the clock period is therefore $T=1/f$
Processor Performance

- Now as a general formula
  - Our program consists of executing $N$ instructions.
  - Our processor needs CPI cycles for each instruction.
  - The maximum clock speed of the processor is $f$, and the clock period is therefore $T=1/f$

- Our program will execute in

  $$N \times CPI \times \frac{1}{f} = N \times CPI \times T \text{ seconds}$$
How can I Make the Program Run Faster?

\[ N \times CPI \times \left(\frac{1}{f}\right) \]
How can I Make the Program Run Faster?

\[ N \times CPI \times (1/f) \]

- Reduce the number of instructions
  - Make instructions that ‘do’ more (CISC)
  - Use better compilers
How can I Make the Program Run Faster?

\[ N \times \text{CPI} \times \left(\frac{1}{f}\right) \]

- **Reduce the number of instructions**
  - Make instructions that ‘do’ more (CISC)
  - Use better compilers

- **Use less cycles to perform the instruction**
  - Simpler instructions (RISC)
  - Use multiple units/ALUs/cores in parallel
How can I Make the Program Run Faster?

\[ N \times CPI \times \left(\frac{1}{f}\right) \]

- **Reduce the number of instructions**
  - Make instructions that ‘do’ more (CISC)
  - Use better compilers

- **Use less cycles to perform the instruction**
  - Simpler instructions (RISC)
  - Use multiple units/ALUs/cores in parallel

- **Increase the clock frequency**
  - Find a ‘newer’ technology to manufacture
  - Redesign time critical components
  - Adopt pipelining
Single-Cycle Performance

- $T_C$ is limited by the critical path ($1\omega$)
Single-Cycle Performance

- Single-cycle critical path:
  \[ T_c = t_{pcq\_PC} + t_{mem} + \max(t_{RF\_read}, t_{sext} + t_{mux}) + t_{ALU} + t_{mem} + t_{mux} + t_{RF\_setup} \]

- In most implementations, limiting paths are:
  - memory, ALU, register file.
  - \[ T_c = t_{pcq\_PC} + 2t_{mem} + t_{RF\_read} + t_{mux} + t_{ALU} + t_{RF\_setup} \]
## Single-Cycle Performance Example

<table>
<thead>
<tr>
<th>Element</th>
<th>Parameter</th>
<th>Delay (ps)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register clock-to-Q</td>
<td>$t_{pcq_PC}$</td>
<td>30</td>
</tr>
<tr>
<td>Register setup</td>
<td>$t_{\text{setup}}$</td>
<td>20</td>
</tr>
<tr>
<td>Multiplexer</td>
<td>$t_{\text{mux}}$</td>
<td>25</td>
</tr>
<tr>
<td>ALU</td>
<td>$t_{\text{ALU}}$</td>
<td>200</td>
</tr>
<tr>
<td>Memory read</td>
<td>$t_{\text{mem}}$</td>
<td>250</td>
</tr>
<tr>
<td>Register file read</td>
<td>$t_{\text{RFread}}$</td>
<td>150</td>
</tr>
<tr>
<td>Register file setup</td>
<td>$t_{\text{RFsetup}}$</td>
<td>20</td>
</tr>
</tbody>
</table>

$$T_c =$$
## Single-Cycle Performance Example

<table>
<thead>
<tr>
<th>Element</th>
<th>Parameter</th>
<th>Delay (ps)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register clock-to-Q</td>
<td>$t_{pcq_PC}$</td>
<td>30</td>
</tr>
<tr>
<td>Register setup</td>
<td>$t_{setup}$</td>
<td>20</td>
</tr>
<tr>
<td>Multiplexer</td>
<td>$t_{mux}$</td>
<td>25</td>
</tr>
<tr>
<td>ALU</td>
<td>$t_{ALU}$</td>
<td>200</td>
</tr>
<tr>
<td>Memory read</td>
<td>$t_{mem}$</td>
<td>250</td>
</tr>
<tr>
<td>Register file read</td>
<td>$t_{RFread}$</td>
<td>150</td>
</tr>
<tr>
<td>Register file setup</td>
<td>$t_{RFsetup}$</td>
<td>20</td>
</tr>
</tbody>
</table>

\[
T_c = t_{pcq\_PC} + 2t_{mem} + t_{RFread} + t_{mux} + t_{ALU} + t_{RFsetup}
\]

\[
= [30 + 2(250) + 150 + 25 + 200 + 20] \text{ ps}
\]

\[
= 925 \text{ ps}
\]
Single-Cycle Performance Example

- Example:

  For a program with 100 billion instructions executing on a single-cycle MIPS processor:
Single-Cycle Performance Example

Example:

For a program with 100 billion instructions executing on a single-cycle MIPS processor:

\[
\text{Execution Time} = \# \text{ instructions} \times \text{CPI} \times \text{TC}
\]

\[
= (100 \times 10^9)(1)(925 \times 10^{-12} \text{ s})
\]

\[
= 92.5 \text{ seconds}
\]