Readings

This week
- Introduction to microarchitecture and single-cycle microarchitecture
  - H&H, Chapter 7.1-7.3
  - P&P, Appendices A and C
- Multi-cycle microarchitecture
  - H&H, Chapter 7.4
  - P&P, Appendices A and C

Next week
- Pipelining
  - H&H, Chapter 7.5
- Pipelining Issues
  - H&H, Chapter 7.7, 7.8.1-7.8.3
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
- Pipelining
- Issues in Pipelining: Control & Data Dependence Handling, State Maintenance and Recovery, ...
- Out-of-Order Execution
Recall: The von Neumann Model

INPUT
Keyboard, Mouse, Disk...

PROCESSING UNIT
ALU
TEMP

MEMORY
Mem Addr Reg
Mem Data Reg

CONTROL UNIT
PC or IP
Inst Register

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: 8 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, ...
  - x86: Intel 80486, Pentium, Pentium Pro, Pentium 4, Kaby Lake, Coffee Lake, Comet Lake, ... AMD K5, K7, K9, Bulldozer, BobCat, ...
  - IBM POWER 4, 5, 6, 7, 8, 9, 10
  - ARM Cortex-M*, ARM Cortex-A*, NVIDIA Denver, Apple A*, M1, ...
  - Alpha 21064, 21164, 21264, 21364, ...
  - ...

(A Bit More on) ISA Design and Tradeoffs
The von Neumann Model/Architecture

- Von Neumann model is also called *stored program computer* (instructions in memory). It has 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**

When is a value interpreted as an instruction?
Recall: The Instruction Cycle

Whether a value fetched from memory is interpreted as an instruction depends on when that value is fetched in the instruction processing cycle.
The von Neumann Model/Architecture

- Von Neumann model is also called *stored program computer* (instructions in memory). It has 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, completed) at a time
    - Program counter (instruction pointer) identifies the current instruction
    - 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)

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

- **PROCESSING UNIT**
  - ALU
  - TEMP

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

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

- **OUTPUT**
  - Monitor, Printer, Disk...
The Von Neumann Model (of a Computer)

Q: Is this the only way that a computer can process computer programs?

A: No.

Qualified Answer: No. But, it has been the dominant way
- i.e., the dominant paradigm for computing
- for N decades

Let's examine a completely different model for processing computer programs
The Dataflow Execution Model of a Computer
The Dataflow Model (of a Computer)

- Von Neumann model: An instruction is fetched and executed in control flow order
  - As specified by the program counter (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 program counter (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?

\[
\begin{align*}
v &\leq a + b; \\
w &\leq b \times 2; \\
x &\leq v - w \\
y &\leq v + w \\
z &\leq x \times y
\end{align*}
\]

Sequential

Dataflow

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

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

- Dataflow node and its ISA representation
Example Dataflow Nodes

*Conditional

*Relational

*Barrier Synch
A Simple Example Dataflow Program

DIAGRAM: A dataflow program consists of nodes representing operations and edges indicating data flow. The diagram shows nodes labeled with operations such as 'Copy', 'BR', and 'DEC', with edges labeled with Boolean values ('T' for true and 'F' for false). The program starts with an input 'I' and proceeds through various operations, finally outputting 'OUT'.
ISA-level Tradeoff: Program Counter

Do we need a Program Counter (PC or IP) in the ISA?

- Yes: Control-driven, sequential execution
  - An instruction is executed when the PC points to it
  - PC automatically changes sequentially (except for control flow instructions)
- No: Data-driven, parallel execution
  - An instruction is executed when all its operand values are available (dataflow)

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 dataflow 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)
Lecture Video on Dataflow Model

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, RISC-V, ...

- 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:
    - Intel 80486, Pentium, Pentium Pro, Pentium 4, Kaby Lake, Coffee Lake, Comet Lake, AMD K5, K7, K9, Bulldozer, BobCat, ...

- Microarchitecture usually changes faster than ISA
  - Few ISAs (x86, ARM, SPARC, MIPS, Alpha, RISC-V) 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
- ...

Intel® 64 and IA-32 Architectures Software Developer’s Manual
Volume 1: Basic Architecture
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
- Bit-serial adder vs. Ripple-carry adder
- 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...

Other examples of the application space that continue to drive the need for unique design points are the following:

1) **scientific applications** such as those whose computations control nuclear power plants, determine where to drill for oil, and predict the weather;
2) **transaction-based applications** such as those that handle ATM transfers and e-commerce business;
3) **business data processing** applications, such as those that handle inventory control, payrolls, IRS activity, and various personnel record keeping, whether the personnel are employees, students, or voters;
4) **network applications**, such as high-speed routing of Internet packets, that enable the connection of your home system to take advantage of the Internet;
5) **guaranteed delivery** (a.k.a. real time) applications that require the result of a computation by a certain critical deadline;
6) **embedded applications**, where the processor is a component of a larger system that is used to solve the (usually) dedicated application;
7) **media applications** such as those that decode video and audio files;
8) random software packages that desktop users would like to run on their PCs.

Each of these application areas has a very different set of characteristics. Each application area demands a different set of tradeoffs to be made in specifying the microprocessor to do the job.


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)

New demands from the top (Look Up)

New demands and personalities of users (Look Up)

New issues and capabilities at the bottom (Look Down)
And, the future is not constant (it changes)!
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?
Mühle Tiefenbrunnen

- 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**
- Today is a **center for a variety of activities**: theater, conferences, restaurants, shops, museum...

http://www.muehle-tiefenbrunnen.ch/
Another Example (I)

Photo credit: Prof. Can Alkan
Another Example (II)

Photo credit: Prof. Can Alkan
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
  - “Translation of ISAs”
  - A virtual ISA can be converted by “software” into an implementation ISA

- We will assume “hardware” implementation 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…

PROCESSING UNIT
ALU
TEMP

MEMORY
Mem Addr Reg
Mem Data Reg

CONTROL UNIT
PC or IP
Inst Register

OUTPUT
Monitor, Printer, Disk…
Recall: Programmer Visible (Architectural) State

- **Memory**: array of storage locations indexed by an address

- **Program Counter**: memory address of the current (or next) instruction

- **Instructions (and programs)** specify how to transform the values of programmer visible state

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

\[\text{AS} = \text{Architectural (programmer visible) state at the beginning of a clock cycle}\]

\[\downarrow\]

\[\text{Process instruction in one clock cycle}\]

\[\downarrow\]

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

- Single-cycle machine

- What is the *clock cycle time* determined by?
- What is the *critical path* (i.e., longest delay path) of the combinational logic determined by?

AS: Architectural 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 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, muxes, decoders, tri-state bufs) 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
Recall: LC-3: A von Neumann Machine

**Diagram Description:**
- **Program Counter:**\[25x491\]
- **Control signals:**\[185x491\]
- **Data:**\[572x258\]
- **ALU:** 2 inputs, 1 output
- **Memory Data Register:** 16-bit addressable
- **Register:**
- **Memory Address Register:**
- **Finite State Machine (for Generating Control Signals):**
- **Instruction Register:**
- **ALU operation:**
- **GateALU:**
- **Keyboard:** KBDR (data), KBSR (status)
- **Monitor:** DDR (data), DSR (status)
- **8 General Purpose Registers (GPR):**

**Figure 4.3** The LC-3 as an example of the von Neumann model
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).

- See P&P Appendix C for more (microprogrammed multi-cycle microarchitecture).
Many Ways of Datapath and Control Design

- There are many ways of designing the datapath and control logic

- Example ways
  - 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 a single instruction
  - \(\{\text{CPI}\} \times \{\text{clock cycle time}\}\)

- Execution time of an entire 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
  - CPI = 1
  - Clock cycle time = long

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

In multi-cycle, we have two degrees of freedom to optimize independently.
A Single-Cycle Microarchitecture
A Closer Look
Remember...

- Single-cycle machine

AS: Architectural State
Let’s Start with the State Elements

- Data and control inputs

**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:**
  If the write enable, WE, is 1, it writes 32-bit data WD into memory location at 32-bit address A on the rising edge of the clock.
  If the write enable is 0, it reads 32-bit data from 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 signal: 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.]**
We Need to Provide the Datapath+Control Logic to Execute All ISA Instructions
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></th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>0</th>
<th>add (32)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>6 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
```
(R-Type) ALU Datapath

if MEM[PC] == ADD rd rs rt
   GPR[rd] ← GPR[rs] + GPR[rt]
   PC ← PC + 4

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

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

### Circuit Diagram

- The diagram illustrates the logic gates and connections for the ALU operation.

### Truth Table

<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; ~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)

```plaintext
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>1</td>
<td>1</td>
<td>4</td>
<td>-2</td>
</tr>
</tbody>
</table>

**Machine Code**

<table>
<thead>
<tr>
<th>OP</th>
<th>DR</th>
<th>SR</th>
<th>imm5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**Diagram**

- **Register file**
  - **DR**
  - **SR**
- **Instruction register**
  - **IR**
  - **ADD R1, R4, # -2**
- **Instruction register**
  - **OP**
  - **DR**
  - **SR**
  - **imm5**
- **Register file**
  - **SR**
  - **DR**
- **ALU**
  - **ADD**
  - **From FSM**
- **Machine Code**
  - **OP**
  - **DR**
  - **SR**
  - **imm5**
  - **15 12 11 9 8 6 5 4 0**
Single-Cycle Datapath for Data Movement Instructions
Load Instructions

- Load 4-byte word

**MIPS assembly**

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

**Machine Encoding**

<table>
<thead>
<tr>
<th>op</th>
<th>rs=base</th>
<th>rt</th>
<th>imm=offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw (35)</td>
<td>base</td>
<td>rt</td>
<td>offset</td>
</tr>
</tbody>
</table>

| 31 | 26 | 25 | 21 | 20 | 16 | 15 | 0 |

**I-Type**

- Semantics

\[
\text{if MEM}[\text{PC}] = \text{lw rt offset}_{16} \text{ (base)} \\
\text{PC} \leftarrow \text{PC} + 4 \\
\text{EA} = \text{sign-extend}(\text{offset}) + \text{GPR}(\text{base}) \\
\text{GPR}[rt] \leftarrow \text{MEM} \left[ \text{translate}(\text{EA}) \right]
\]
if MEM[PC]==LW rt offset_{16} (base)
EA = sign-extend(offset) + GPR[base]
GPR[rt] ← MEM[translate(EA)]
PC ← PC + 4

Combinational state update logic

IF  ID  EX  MEM  WB
Store Instructions

- Store 4-byte word

  MIPS assembly

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

  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 \leftarrow PC + 4
  
  EA = \text{sign-extend}(\text{offset}) + \text{GPR}(\text{base})
  
  MEM[ \text{translate}(EA) ] \leftarrow \text{GPR}[rt]
if MEM[PC] == SW rt offset[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

```plaintext
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

```plaintext
if MEM[PC] == j immediate_{26}
    target = { PC +[31:28], immediate_{26}, 2'b00 }
    PC ← target
```

\[^{\dagger}\text{This is the incremented PC}\]
Unconditional Jump Datapath

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$_{26}$
    
    
    
    $ra \leftarrow PC + 4$
    
    
    
    target = \{ PC$^\dagger$[31:28], \text{immediate$_{26}$}, 2'b00 \}
    
    
    
    PC $\leftarrow$ target

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

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

$^\dagger$ This is the incremented PC
### MIPS Reference Data

**MIPS Instruction Set**

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>00100</td>
<td>Add</td>
</tr>
<tr>
<td>00101</td>
<td>Add Immediate</td>
</tr>
<tr>
<td>01000</td>
<td>Add</td>
</tr>
<tr>
<td>01001</td>
<td>Add Immediate</td>
</tr>
<tr>
<td>01010</td>
<td>Add Uncond</td>
</tr>
<tr>
<td>01011</td>
<td>Add Uncond</td>
</tr>
<tr>
<td>01100</td>
<td>Add Uncond</td>
</tr>
<tr>
<td>01101</td>
<td>Add Uncond</td>
</tr>
<tr>
<td>01110</td>
<td>Add Uncond</td>
</tr>
<tr>
<td>01111</td>
<td>Add Uncond</td>
</tr>
<tr>
<td>10000</td>
<td>Branch On FP Flags</td>
</tr>
<tr>
<td>10001</td>
<td>Branch On FP Flags</td>
</tr>
<tr>
<td>10010</td>
<td>Branch On FP Flags</td>
</tr>
<tr>
<td>10011</td>
<td>Branch On FP Flags</td>
</tr>
<tr>
<td>10100</td>
<td>Branch On FP Flags</td>
</tr>
<tr>
<td>10101</td>
<td>Branch On FP Flags</td>
</tr>
<tr>
<td>10110</td>
<td>Branch On FP Flags</td>
</tr>
<tr>
<td>10111</td>
<td>Branch On FP Flags</td>
</tr>
<tr>
<td>11000</td>
<td>Branch On FP Flags</td>
</tr>
<tr>
<td>11001</td>
<td>Branch On FP Flags</td>
</tr>
<tr>
<td>11010</td>
<td>Branch On FP Flags</td>
</tr>
<tr>
<td>11011</td>
<td>Branch On FP Flags</td>
</tr>
<tr>
<td>11100</td>
<td>Branch On FP Flags</td>
</tr>
<tr>
<td>11101</td>
<td>Branch On FP Flags</td>
</tr>
<tr>
<td>11110</td>
<td>Branch On FP Flags</td>
</tr>
<tr>
<td>11111</td>
<td>Branch On FP Flags</td>
</tr>
<tr>
<td>00000</td>
<td>Jump And Link</td>
</tr>
<tr>
<td>00001</td>
<td>Jump And Link</td>
</tr>
<tr>
<td>00010</td>
<td>Jump And Link</td>
</tr>
<tr>
<td>00011</td>
<td>Jump And Link</td>
</tr>
<tr>
<td>00100</td>
<td>Jump And Link</td>
</tr>
<tr>
<td>00101</td>
<td>Jump And Link</td>
</tr>
<tr>
<td>00110</td>
<td>Jump And Link</td>
</tr>
<tr>
<td>00111</td>
<td>Jump And Link</td>
</tr>
<tr>
<td>01000</td>
<td>Jump And Link</td>
</tr>
<tr>
<td>01001</td>
<td>Jump And Link</td>
</tr>
<tr>
<td>01010</td>
<td>Jump And Link</td>
</tr>
<tr>
<td>01011</td>
<td>Jump And Link</td>
</tr>
<tr>
<td>01100</td>
<td>Jump And Link</td>
</tr>
<tr>
<td>01101</td>
<td>Jump And Link</td>
</tr>
<tr>
<td>01110</td>
<td>Jump And Link</td>
</tr>
<tr>
<td>01111</td>
<td>Jump And Link</td>
</tr>
<tr>
<td>10000</td>
<td>Jump Reg</td>
</tr>
<tr>
<td>10001</td>
<td>Jump Reg</td>
</tr>
<tr>
<td>10010</td>
<td>Jump Reg</td>
</tr>
<tr>
<td>10011</td>
<td>Jump Reg</td>
</tr>
<tr>
<td>11000</td>
<td>Jump Reg</td>
</tr>
<tr>
<td>11001</td>
<td>Jump Reg</td>
</tr>
<tr>
<td>11010</td>
<td>Jump Reg</td>
</tr>
<tr>
<td>11011</td>
<td>Jump Reg</td>
</tr>
<tr>
<td>00000</td>
<td>Jump Uncond</td>
</tr>
<tr>
<td>00001</td>
<td>Jump Uncond</td>
</tr>
<tr>
<td>00010</td>
<td>Jump Uncond</td>
</tr>
<tr>
<td>00011</td>
<td>Jump Uncond</td>
</tr>
<tr>
<td>00100</td>
<td>Jump Uncond</td>
</tr>
<tr>
<td>00101</td>
<td>Jump Uncond</td>
</tr>
<tr>
<td>00110</td>
<td>Jump Uncond</td>
</tr>
<tr>
<td>00111</td>
<td>Jump Uncond</td>
</tr>
<tr>
<td>01000</td>
<td>Jump Uncond</td>
</tr>
<tr>
<td>01001</td>
<td>Jump Uncond</td>
</tr>
<tr>
<td>01010</td>
<td>Jump Uncond</td>
</tr>
<tr>
<td>01011</td>
<td>Jump Uncond</td>
</tr>
<tr>
<td>01100</td>
<td>Jump Uncond</td>
</tr>
<tr>
<td>01101</td>
<td>Jump Uncond</td>
</tr>
<tr>
<td>01110</td>
<td>Jump Uncond</td>
</tr>
<tr>
<td>01111</td>
<td>Jump Uncond</td>
</tr>
</tbody>
</table>

**Arithmetic Core Instruction Set**

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>Add</td>
</tr>
<tr>
<td>00001</td>
<td>Add Immediate</td>
</tr>
<tr>
<td>00100</td>
<td>Add</td>
</tr>
<tr>
<td>00101</td>
<td>Add Immediate</td>
</tr>
<tr>
<td>01000</td>
<td>Add</td>
</tr>
<tr>
<td>01001</td>
<td>Add Immediate</td>
</tr>
</tbody>
</table>

**Opcodes, Base Conversion, ASCII Symbols**

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Base Conversion</th>
<th>ASCII Symbols</th>
</tr>
</thead>
<tbody>
<tr>
<td>01000</td>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>01001</td>
<td>1</td>
<td>B</td>
</tr>
<tr>
<td>01010</td>
<td>1</td>
<td>C</td>
</tr>
<tr>
<td>01011</td>
<td>1</td>
<td>D</td>
</tr>
<tr>
<td>01100</td>
<td>1</td>
<td>E</td>
</tr>
<tr>
<td>01101</td>
<td>1</td>
<td>F</td>
</tr>
</tbody>
</table>

**Floating-Point Instruction Formats**

<table>
<thead>
<tr>
<th>Format</th>
<th>Integer</th>
</tr>
</thead>
<tbody>
<tr>
<td>01000</td>
<td>1</td>
</tr>
<tr>
<td>01001</td>
<td>1</td>
</tr>
<tr>
<td>01010</td>
<td>1</td>
</tr>
<tr>
<td>01011</td>
<td>1</td>
</tr>
<tr>
<td>01100</td>
<td>1</td>
</tr>
<tr>
<td>01101</td>
<td>1</td>
</tr>
<tr>
<td>01110</td>
<td>1</td>
</tr>
<tr>
<td>01111</td>
<td>1</td>
</tr>
</tbody>
</table>

**Register Name, Number, Usage, Conversion**

<table>
<thead>
<tr>
<th>Name</th>
<th>Number</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>SRC1</td>
<td>0</td>
<td>Source</td>
</tr>
<tr>
<td>SRC2</td>
<td>1</td>
<td>Source</td>
</tr>
<tr>
<td>DST1</td>
<td>2</td>
<td>Destination</td>
</tr>
<tr>
<td>DST2</td>
<td>3</td>
<td>Destination</td>
</tr>
<tr>
<td>IMM</td>
<td>4</td>
<td>Immediate</td>
</tr>
</tbody>
</table>

**IEEE 754 Floating-Point Standard**

- **Single Precision**
- **Double Precision**
- **Exception Codes**

**memory Allocation**

- **Stack Frame**
- **Register**
- **Memory Allocation**

**Data Alignment**

- **Word Align**
- **Double Word Align**

**Exception Control, Registers: Cause and Status**

- **Interrupts**
- **MIPS Interrupts**

**size prefix**

- **16-bit**
- **32-bit**

---

**Aside: MIPS Cheat Sheet**


- On the course website
Conditional Branch Instructions

- **beq (Branch if Equal)**
  
  ```
  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 MEM[PC] == beq rs rt immediate_{16} 
  target = PC^\dagger + sign-extend(immediate) \times 4
  if GPR[rs] == GPR[rt] then PC ← target
  else PC ← PC + 4
  ```

- **Variations: beq, bne, blez, bgtz**

^\dagger 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
Single-Cycle Control Logic
Single-Cycle Hardwired Control

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

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>shamt</td>
<td>funct</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

  - **R-Type**
    - 6 bits
    - 5 bits
    - 5 bits
    - 5 bits
    - 5 bits
    - 6 bits

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>opcode</td>
<td>rs</td>
<td>rt</td>
<td>immediate</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

  - **I-Type**
    - 6 bits
    - 5 bits
    - 5 bits
    - 16 bits

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
</tr>
</thead>
<tbody>
<tr>
<td>opcode</td>
<td>immediate</td>
<td></td>
</tr>
</tbody>
</table>

  - **J-Type**
    - 6 bits
    - 26 bits

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

<table>
<thead>
<tr>
<th></th>
<th>When De-asserted</th>
<th>When asserted</th>
<th>Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>RegDest</strong></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><strong>ALUSrc</strong></td>
<td>2\textsuperscript{nd} ALU input from 2\textsuperscript{nd} GPR read port</td>
<td>2\textsuperscript{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><strong>MemtoReg</strong></td>
<td>Steer ALU result to GPR write port</td>
<td>steer memory load to GPR write port</td>
<td>opcode==LW</td>
</tr>
<tr>
<td><strong>RegWrite</strong></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>

JAL and JALR require additional RegDest and MemtoReg options
## Single-Bit Control Signals (II)

<table>
<thead>
<tr>
<th></th>
<th>When De-asserted</th>
<th>When asserted</th>
<th>Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>MemRead</strong></td>
<td>Memory read disabled</td>
<td>Memory read port return load value</td>
<td>(\text{opcode}==\text{LW})</td>
</tr>
<tr>
<td><strong>MemWrite</strong></td>
<td>Memory write disabled</td>
<td>Memory write enabled</td>
<td>(\text{opcode}==\text{SW})</td>
</tr>
<tr>
<td>**PCSrc}_1</td>
<td>According to (\text{PCSrc}_2)</td>
<td>next PC is based on 26-bit immediate jump target</td>
<td>((\text{opcode}==\text{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>((\text{opcode}==\text{Bxx}) \ &amp; \ &amp; \ &quot;\text{bcond is satisfied}&quot;)</td>
</tr>
</tbody>
</table>

JR and JALR require additional PCSrc options
R-Type ALU

**Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]**
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.
Some control signals are dependent on the processing of data.
Jump
What is in That Control Box?

- **Combinational Logic → Hardwired Control**
  - Idea: Most control signals generated combinationally based on bits in instruction encoding

- **Sequential Logic → Sequential Control**
  - Idea: A memory structure contains the control signals associated with an instruction
    - Called Control Store

- **Both types of control structure can be used in single-cycle processors**
  - Choice depends on latency of each structure + how much on the critical path control signal generation is, etc.
**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 Single-Cycle MIPS Processor (from H&H)

See backup slides to reinforce the concepts we have covered. They are to complement your reading:
H&H, Chapter 7.1-7.3, 7.6
Another Complete Single-Cycle Processor

Single-cycle processor. Harris and Harris, Chapter 7.3.
Example: Single-Cycle Datapath: `lw` fetch

**STEP 1:** Fetch instruction

```asm
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
```
Single-Cycle Datapath: \texttt{lw} immediate

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

\[
\text{lw} \ $s3, 1($0) \quad \# \text{read memory word 1 into $s3}
\]

### 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: \texttt{lw} memory read

\textbf{STEP 5:} Read from memory and write back to register file

\texttt{lw} $\$s3, 1(\$0)$  # read memory word 1 into $\$s3

\textbf{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>
Similarly, We Need to Design the Control Unit

- **Control signals** are 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>
Another Complete Single-Cycle Processor (H&H)
Your Reading Assignment

- Please read the Lecture Slides and the Backup Slides

- Please do your readings from the H&H Book
  - H&H, Chapter 7.1-7.3, 7.6
Single-Cycle Uarch I (We Developed in Lectures)

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

JAL, JR, JALR omitted
Single-Cycle Uarch II (In Your Readings)
Evaluating the Single-Cycle Microarchitecture
A Single-Cycle Microarchitecture

- Is *this* a good idea/design?

- When is this a good design?

- When is this a bad design?

- How can we design a better microarchitecture?
Performance Analysis Basics
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
  ▪ *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.
Processor Performance

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

- Our program executes in

  $$N \times \text{CPI} \times (1/f) =$$
  
  $$N \times \text{CPI} \times T \text{ seconds}$$
Performance Analysis Basics

- Execution time of a single instruction
  - \( \{\text{CPI}\} \times \{\text{clock cycle time}\} \)
    - CPI: Number of cycles it takes to execute an instruction

- Execution time of an entire program
  - Sum over all instructions \([\{\text{CPI}\} \times \{\text{clock cycle time}\}]\]
  - \( \{\text{# of instructions}\} \times \{\text{Average CPI}\} \times \{\text{clock cycle time}\} \)
Performance Analysis of Our Single-Cycle Design
A Single-Cycle Microarchitecture: Analysis

- Every instruction takes 1 cycle to execute
  - CPI (Cycles per instruction) is strictly 1

- How long each instruction takes is determined by how long the slowest instruction takes to execute
  - Even though many instructions do not need that long to execute

- Clock cycle time of the microarchitecture is determined by how long it takes to complete the slowest instruction
  - Critical path of the design is determined by the processing time of the slowest instruction
What is the Slowest Instruction to Process?

- Let’s go back to the basics

- All six phases of the instruction processing cycle take a *single* *machine* *clock* *cycle* to complete
  - Fetch
  - Decode
  - Evaluate Address
  - Fetch Operands
  - Execute
  - Store Result

- Do each of the above phases take the same time (latency) for all instructions?
Let’s Find the Critical Path
## Example Single-Cycle Datapath Analysis

- **Assume (for the design in the previous slide)**
  - memory units (read or write): 200 ps
  - ALU and adders: 100 ps
  - register file (read or write): 50 ps
  - other combinational logic: 0 ps

<table>
<thead>
<tr>
<th>steps</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
<th>Delay</th>
</tr>
</thead>
<tbody>
<tr>
<td>resources</td>
<td>mem</td>
<td>RF</td>
<td>ALU</td>
<td>mem</td>
<td>RF</td>
<td></td>
</tr>
<tr>
<td>R-type</td>
<td>200</td>
<td>50</td>
<td>100</td>
<td>50</td>
<td>400</td>
<td></td>
</tr>
<tr>
<td>I-type</td>
<td>200</td>
<td>50</td>
<td>100</td>
<td>50</td>
<td>400</td>
<td></td>
</tr>
<tr>
<td>LW</td>
<td>200</td>
<td>50</td>
<td>100</td>
<td>200</td>
<td>50</td>
<td>600</td>
</tr>
<tr>
<td>SW</td>
<td>200</td>
<td>50</td>
<td>100</td>
<td>200</td>
<td></td>
<td>550</td>
</tr>
<tr>
<td>Branch</td>
<td>200</td>
<td>50</td>
<td>100</td>
<td></td>
<td></td>
<td>350</td>
</tr>
<tr>
<td>Jump</td>
<td></td>
<td></td>
<td>200</td>
<td></td>
<td></td>
<td>200</td>
</tr>
</tbody>
</table>
Let’s Find the Critical Path
R-Type and I-Type ALU

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

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

[Based on original figure from P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]
What About Control Logic?

- How does that affect the critical path?

- Food for thought for you:
  - Can control logic be on the critical path?
  - Historical example:
    - CDC 5600: control store access too long...
What is the Slowest Instruction to Process?

- Real world: **Memory is slow (not magic)**

- What if memory *sometimes* takes 100ms to access?

- Does it make sense to have a simple register to register add or jump to take \{100ms+all else to do a memory operation\}? 

- And, what if you need to access memory more than once to process an instruction? 
  - Which instructions need this? 
  - Do you provide multiple ports to memory?
Single Cycle uArch: Complexity

- Contrived
  - All instructions run as slow as the slowest instruction

- Inefficient
  - All instructions run as slow as the slowest instruction
  - Must provide worst-case combinational resources in parallel as required by any instruction
  - Need to replicate a resource if it is needed more than once by an instruction during different parts of the instruction processing cycle

- Not necessarily the simplest way to implement an ISA
  - Single-cycle implementation of REP MOVS (x86) or INDEX (VAX)?

- Not easy to optimize/improve performance
  - Optimizing the common case does not work (e.g. common instructions)
  - Need to optimize the worst case all the time
(Micro)architecture Design Principles

- **Critical path design**
  - Find and **decrease the maximum combinational logic delay**
  - Break a path into multiple cycles if it takes too long

- **Bread and butter (common case) design**
  - Spend time and resources on where it matters most
    - i.e., improve what the machine is really designed to do
  - Common case vs. uncommon case

- **Balanced design**
  - **Balance** instruction/data flow through hardware components
  - **Design to eliminate bottlenecks**: balance the hardware for the work
Single-Cycle Design vs. Design Principles

- Critical path design
- Bread and butter (common case) design
- Balanced design

*How does a single-cycle microarchitecture fare with respect to these principles?*
Aside: System Design Principles

- When designing computer systems/architectures, it is important to follow good principles
  - Actually, this is true for *any* system design
    - Real architectures, buildings, bridges, ...
    - Good consumer products
    - ...

- Remember: “principled design” from our second lecture
  - Frank Lloyd Wright: “architecture [...] based upon principle, and not upon precedent”
Aside: From Lecture 2

- “architecture [...] based upon principle, and not upon precedent”
This
That
Recall: Takeaways

- It all starts from the **basic building blocks and design principles**

- And, **knowledge of how to use, apply, enhance them**

- **Underlying technology might change** (e.g., steel vs. wood)
  - but **methods of taking advantage of technology bear resemblance**
  - **methods used for design depend on the principles** employed
Aside: System Design Principles

- We will continue to cover key principles in this course
- Here are some references where you can learn more

- Gene M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," AFIPS Conference, April 1967. (Amdahl’s Law → Common-case design)
A Key System Design Principle

- Keep it simple

- “Everything should be made as simple as possible, but no simpler.”
  - Albert Einstein

- And, keep it low cost: “An engineer is a person who can do for a dime what any fool can do for a dollar.”

- For more, see:
Multi-Cycle Microarchitectures
Backup Slides on Single-Cycle Uarch for Your Own Study

Please study these to reinforce the concepts we covered in lectures.

Please do the readings together with these slides:
H&H, Chapter 7.1-7.3, 7.6
Another Single-Cycle MIPS Processor (from H&H)

These are slides for your own study. They are to complement your reading H&H, Chapter 7.1-7.3, 7.6
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 \texttt{0x0040000}\texttt{0}

```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

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'b00000)? // is address 0?
    R_arr[a_rs] : 0; // Read RS or 0

assign do_rt = (a_rt != 5'b00000)? // 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 [31:0] M_arr [0:65535]; // 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
```

<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: \texttt{lw} immediate

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

\texttt{lw} \ $s3, 1($0) \ # \ read \ memory \ word \ 1 \ into \ $s3

\textbf{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: \texttt{lw} memory read

\textbf{STEP 5:} Read from memory and write back to register file

\texttt{lw} $s3, 1($0) # read memory word 1 into $s3

\begin{tabular}{|c|c|c|c|}
\hline
\textbf{op} & \textbf{rs} & \textbf{rt} & \textbf{imm} \\
\hline
6 bits & 5 bits & 5 bits & 16 bits \\
\hline
\end{tabular}
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: \textit{sw}

- Write data in rt to memory

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

\textbf{I-Type}

\begin{tabular}{|c|c|c|c|}
\hline
\textbf{op} & \textbf{rs} & \textbf{rt} & \textbf{imm} \\
\hline
6 bits & 5 bits & 5 bits & 16 bits \\
\hline
\end{tabular}
Single-Cycle Datapath: R-type Instructions

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

```
add t, b, c  # t = b + c
```

**R-Type**

<table>
<thead>
<tr>
<th></th>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<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

beq $s0, $s1, target  # branch is taken

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

Main Decoder

MemtoReg
MemWrite
Branch
ALUSrc
RegDst
RegWrite

ALUOp1:0

ALU Decoder

ALUOp
ALUControl2:0

Funct5:0

Opcode5:0

ALUOp
<table>
<thead>
<tr>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
</tr>
<tr>
<td>01</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>11</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</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; ~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>
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</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>
Control Unit: ALU Decoder

<table>
<thead>
<tr>
<th>ALUOp&lt;sub&gt;1:0&lt;/sub&gt;</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>ALUOp&lt;sub&gt;1:0&lt;/sub&gt;</th>
<th>Funct</th>
<th>ALUControl&lt;sub&gt;2:0&lt;/sub&gt;</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&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></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_{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>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
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>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>
</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>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>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>
</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>
</tr>
</tbody>
</table>

■ New Control Signal

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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Op_{5:0}</th>
<th>RegWrite</th>
<th>RegDst</th>
<th>AluSrc</th>
<th>Branch</th>
<th>MemWrite</th>
<th>MemtoReg</th>
<th>ALUOp_{1:0}</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>
</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>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>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>0</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_{5:0}</th>
<th>RegWrite</th>
<th>RegDst</th>
<th>AluSrc</th>
<th>Branch</th>
<th>MemWrite</th>
<th>MemtoReg</th>
<th>ALUOp_{1:0}</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>0000000</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
  - {CPI} × {clock cycle time}

- Execution time of a program
  - Sum over all instructions [{CPI} × {clock cycle time}]
  - {# of instructions} × {Average CPI} × {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

$$N \times CPI \times (1/f) = N \times CPI \times T \text{ seconds}$$
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?

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

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

\[ N \times \text{CPI} \times (1/f) \]

- **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?

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

- **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 (1w)
Single-Cycle Performance

- Single-cycle critical path:
  - $T_c = t_{pcq\_PC} + t_{mem} + \max(t_{RFread}, t_{sext} + t_{mux}) + t_{ALU} + t_{mem} + t_{mux} + t_{RFsetup}$

- In most implementations, limiting paths are:
  - memory, ALU, register file.
  - $T_c = t_{pcq\_PC} + 2t_{mem} + t_{RFread} + t_{mux} + t_{ALU} + t_{RFsetup}$
## 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>$t_{\text{setup}}$</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>

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

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

\[
= 925 \text{ ps}
\]
Example:

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

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

Execution Time = \# instructions \times CPI \times TC

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

= 92.5 seconds