Microarchitecture

Digital Design and Computer Architecture
Mohammad Sadrosadati
Frank K. Gürkaynak

http://safari.ethz.ch/ddca
In This Lecture

- Overview of the MIPS Microprocessor
- RTL Design Revisited
- Single-Cycle Processor
- Performance Analysis
## Introduction

**Microarchitecture:**
- How to implement an architecture in hardware

**Processor:**
- Datapath: functional blocks
- Control: control signals

<table>
<thead>
<tr>
<th>Application Software</th>
<th>programs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operating Systems</td>
<td>device drivers</td>
</tr>
<tr>
<td>Architecture</td>
<td>instructions registers</td>
</tr>
<tr>
<td>Micro-architecture</td>
<td>datapaths controllers</td>
</tr>
<tr>
<td>Logic</td>
<td>adders memories</td>
</tr>
<tr>
<td>Digital Circuits</td>
<td>AND gates NOT gates</td>
</tr>
<tr>
<td>Analog Circuits</td>
<td>amplifiers filters</td>
</tr>
<tr>
<td>Devices</td>
<td>transistors diodes</td>
</tr>
<tr>
<td>Physics</td>
<td>electrons</td>
</tr>
</tbody>
</table>
Microarchitecture

- Multiple implementations for a single architecture:

  - **Single-cycle** *(this week)*
    - Each instruction executes in a single cycle

  - **Multicycle** *(in the book, we will not cover it)*
    - Each instruction is broken up into a series of shorter steps
    - One instruction is executed piece-wise
    - Allows re-use of components (ALU, memory)

  - **Pipelined** *(just after we finish this one)*
    - Each instruction is broken up into a series of steps
    - Multiple instructions execute at once
    - Better for performance
What does our Microprocessor do?

- Program is in memory, stored as a series of instructions.

- Basic execution is:
  - Read one instruction
  - Decode what it wants
  - Find the operands either from registers or from memory
  - Perform the operation as dictated by the instruction
  - Write the result (if necessary)
  - Go to the next instruction
What are the Basic Components of MIPS?

- The main pieces
  - Memory to store program
  - Registers to access data fast
  - Data memory to store more data
  - An ALU to perform the core function
What are the Basic Components of MIPS?

- **The main pieces**
  - Memory to store program
  - Registers to access data fast
  - Data memory to store more data
  - An ALU to perform the core function

- **Some auxiliary pieces**
  - A program counter to tell where to find next instruction
  - Some logic to decode instructions
  - A way to manipulate the program counter for branches
What are the Basic Components of MIPS?

- The main pieces
  - Memory to store program
  - Registers to access data fast
  - Data memory to store more data
  - An ALU to perform the core function

- Some auxiliary pieces
  - A program counter to tell where to find next instruction
  - Some logic to decode instructions
  - A way to manipulate the program counter for branches

- Today we will see how it all comes together
RTL Design Revisited

- Registers store the current state

- Combinational logic determines next state
  - Data moves from one register to another (datapath)
  - Control signals move data through the datapath
  - Control signals depend on state and input

- Clock moves us from state to another
Let us Start with Building the Processor

- Program is stored in a (read-only) memory as 32-bit binary values.

- Memory addresses are also 32-bit wide, allowing (in theory) $2^{32} = 4$ Gigabytes of addressed memory.

- The actual address is stored in a register called PC.

<table>
<thead>
<tr>
<th>Address</th>
<th>Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>0040000C</td>
<td>0 1 6 D 4 0 2 2</td>
</tr>
<tr>
<td>00400008</td>
<td>2 2 6 8 F F F 4</td>
</tr>
<tr>
<td>00400004</td>
<td>0 2 3 2 8 0 2 0</td>
</tr>
<tr>
<td>00400000</td>
<td>8 C 0 A 0 0 2 0</td>
</tr>
</tbody>
</table>

- Stored Program

<table>
<thead>
<tr>
<th>Address</th>
<th>Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>0040000C</td>
<td>lw $t2, 32($0)</td>
</tr>
<tr>
<td>00400008</td>
<td>add $s0, $s1, $s2</td>
</tr>
<tr>
<td>00400004</td>
<td>addi $t0, $s3, -12</td>
</tr>
<tr>
<td>00400000</td>
<td>sub $t0, $t3, $t5</td>
</tr>
</tbody>
</table>

- Assembly Code
  - lw $t2, 32($0)
  - add $s0, $s1, $s2
  - addi $t0, $s3, -12
  - sub $t0, $t3, $t5

- Machine Code
  - 0x8C0A0020
  - 0x02328020
  - 0x2268FFF4
  - 0x016D4022

- Main Memory

- PC

- PC
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 have an Instruction Now What?

- There are different types of instructions
  - \textbf{R type} (three registers)
  - \textbf{I type} (the one with immediate values)
  - \textbf{J type} (for changing the flow)

- First only a subset of MIPS instructions:
  - R type instructions: \texttt{and, or, add, sub, slt}
  - Memory instructions: \texttt{lw, sw}
  - Branch instructions: \texttt{beq}

- Later we will add \texttt{addi} and \texttt{j}
R-Type MIPS instructions

- **Register-type:** 3 register operands:
  - rs, rt: source registers
  - rd: destination register

- **Other fields:**
  - **op:** the operation code or opcode (0 for R-type instructions)
  - **funct:** the function together, the opcode and function tell the computer what operation to perform
  - **shamt:** the shift amount for shift instructions, otherwise it’s 0

### 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>
R-Type Examples

Assembly Code

- add $s0, $s1, $s2
- sub $t0, $t3, $t5

Field Values

<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>0</td>
<td>17</td>
<td>18</td>
<td>16</td>
<td>0</td>
<td>32</td>
</tr>
<tr>
<td>0</td>
<td>11</td>
<td>13</td>
<td>8</td>
<td>0</td>
<td>34</td>
</tr>
</tbody>
</table>

Machine Code

<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>000000</td>
<td>10001</td>
<td>10010</td>
<td>10000</td>
<td>00000</td>
<td>100000</td>
</tr>
<tr>
<td>000000</td>
<td>01011</td>
<td>01101</td>
<td>01000</td>
<td>00000</td>
<td>100010</td>
</tr>
</tbody>
</table>

Note the order of registers in the assembly code:

- add rd, rs, rt
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

```verilog
input [4:0] a_rs, a_rt, a_rd;
input [31:0] di_rd;
in
put 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

```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; 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
```
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>
Review: ALU

<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>
So far so good

- We have most of the components for MIPS
  - We still do not have data memory
So far so good

- **We have most of the components for MIPS**
  - We still do not have data memory

- **We can implement all R-type instructions**
  - Assuming our ALU can handle all of the functions
  - At the moment we do not have the shifter
  - We know what to do for more functions.
So far so good

- We have most of the components for MIPS
  - We still do not have data memory

- We can implement all R-type instructions
  - Assuming our ALU can handle all of the functions
  - At the moment we do not have the shifter
  - We know what to do for more functions.

- ...but not much else
  - No memory access
  - No immediate values
  - No conditional branches, no absolute jumps
Data Memory

- 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
```
Let us Read (then Write) to Memory

- The main memory can have up to $2^{32}$ bytes
  - It needs an 32-bit address for the entire memory.
  - In practice much smaller memories are used

- We need the lw instruction to read from memory
  - I-type instruction
  - We need to calculate the address from an immediate value and a register.
  - Maybe we can use the ALU to add immediate to the register?
I-Type

- **Immediate-type: 3 operands:**
  - rs, rt: register operands
  - imm: 16-bit two’s complement immediate

- **Other fields:**
  - $op$: the opcode
  - Simplicity favors regularity: all instructions have opcode
  - Operation is completely determined by the opcode

<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>
Differences between R-type and I-type

- Changes to Register file
  - $RS + sign\ extended\ immediate$ is the address
  - RS is always read.
  - For some instructions RT is read as well
  - If one register is written it is RT, not RD
  - The $write\_data\_in$ can come from Memory or ALU
  - Not all I-type instructions write to register file

- Changes to ALU
  - Used to calculate memory address, add function needed during non R-type operations
  - Result is also used as address for the data memory
  - One input is no longer RT, but the immediate value
Now Writing to Memory is Similar

- We need the `sw` instruction to write to memory
  - I-type instruction
  - Address calculated the same way
  - Unlike, previous step, there is no writing back to register.
  - But we need to enable the write enable of the memory
Our MIPS Datapath has Several Options

- **ALU inputs**
  - Either RT or Immediate

- **Write Address of Register File**
  - Either RD or RT

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

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

- **Write enable of Memory**
  - Only when writing to memory (sw)
Our MIPS Datapath has Several Options

- **ALU inputs**
  - Either RT or Immediate \textit{(MUX)}

- **Write Address of Register File**
  - Either RD or RT \textit{(MUX)}

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

- **Write enable of Register File**
  - Not always a register write \textit{(MUX)}

- **Write enable of Memory**
  - Only when writing to memory (sw) \textit{(MUX)}

\textit{All these options are our control signals}
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&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>0000000</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
Now we are much better

- We have almost all components for MIPS

- We already had (nearly) all R-type instructions
  - Our ALU can implement some functions
  - We know what to do for more functions.

- Now we can also Read and Write to memory

- Conditional Branches are still missing
  - Until know, PC just increases by 4 every time.
  - We need to somehow be able to influence this.
Branch if equal: BEQ

- From Appendix B:
  
  Opcode: 000100
  
  If ([rs]==[rt]) PC = BTA
  
  BTA = Branch Target Address
  = PC+4 +(SignImm <<2)
Branch if equal: BEQ

- **From Appendix B:**
  
  Opcode: 000100
  
  If ([rs] == [rt]) PC = BTA
  
  BTA = Branch Target Address
  = PC+4 + (SignImm << 2)

- **We need ALU for comparison**
  - Subtract RS – RT
  - We need a ‘zero’ flag, if RS equals RT
Branch if equal: BEQ

- **From Appendix B:**
  
  Opcode: 000100
  
  If ([rs]==[rt]) PC= BTA
  
  BTA = Branch Target Address
  = PC+4 +(SignImm <<2)

- **We need ALU for comparison**
  - Subtract RS – RT
  - We need a ‘zero’ flag, if RS equals RT

- **We also need a Second Adder**
  - The immediate value has to be added to PC+4
Branch if equal: BEQ

- From Appendix B:
  Opcode: 000100
  If ([rs] == [rt]) PC = BTA
  BTA = Branch Target Address
  = PC+4 + (SignImm << 2)

- We need ALU for comparison
  - Subtract RS – RT
  - We need a ‘zero’ flag, if RS equals RT

- We also need a Second Adder
  - The immediate value has to be added to PC+4

- PC has now two options
  - Either PC+4 or the new BTA
## More Control Signals

<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</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?
We are almost done

- We have almost all components for MIPS

- We already had (nearly) all R-type instructions
  - Our ALU can implement some functions
  - We know what to do for more functions.

- Now we can also Read and Write to memory

- We also have conditional branches

- Absolute jumps are still missing
  - J-type instructions
  - The next PC address is directly determined
Machine Language: J-Type

- Jump-type
  - 26-bit address operand (addr)
  - Used for jump instructions (j)

J-Type

<table>
<thead>
<tr>
<th>Field Values</th>
<th>Machine Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>0x0100028</td>
</tr>
<tr>
<td>imm</td>
<td>000011 00 0001 0000 0000 0010 1000 (0xC100028)</td>
</tr>
</tbody>
</table>

| JTA | 0000 0000 0100 0000 0000 0000 1010 0000 (0x004000A0) |
| 26-bit addr | 0000 0000 0100 0000 0000 0000 1010 0000 (0x0100028) |

6 bits 26 bits
Jump: J

- **From Appendix B:**
  - Opcode: 000010
  - PC = JTA
  - JTA = Jump Target Address
    = \{(PC+4)[31:28], addr, 2'b0\}

- **No need for ALU**
  - Assemble the JTA from PC and 26-bit immediate

- **No need for memory, register file**

- **One more option (mux + control signal) for PC**
## Even 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>
<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>funct</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>add</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>add</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>sub</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>X</td>
<td>1</td>
</tr>
</tbody>
</table>

### New Control Signal
- **Jump**: Direct jump yes or no
Now we are Almost Done

- We have all major parts
  - We can implement, R, I, J type instructions
  - We can calculate, branch, and jump
  - We would be able to run most programs
Now we are Almost Done

- **We have all major parts**
  - We can implement, R, I, J type instructions
  - We can calculate, branch, and jump
  - We would be able to run most programs

- **Still we can improve:**
  - ALU can be improved for more ‘funct’
  - We do not have a shifter.
  - Or a multiplier.
  - More I-type functions (only lw and sw supported)
  - More J-type functions (jal, jr)
Now the original slides once again

- The following slides will repeat what we just did
- Consistent with the text book
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.
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

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

**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: \texttt{lw} immediate

\textbf{STEP 3:} Sign-extend the immediate

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

\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: lw memory read**

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

```plaintext
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
```

### l-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: sw

- Write data in rt to memory

```plaintext
sw $t7, 44($0)  # write t7 into memory address 44
```

**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: 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>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 << 2) + (PC+4)

```
beq $s0, $s1, target  # branch is taken
```
Complete Single-Cycle Processor
Control Unit

![Control Unit Diagram]

### ALU Op 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>

### Control Unit Fields

- **Opcode (5:0)**
  - MemtoReg
  - MemWrite
  - Branch
  - ALUSrc
  - RegDst
  - RegWrite

- **Funct (5:0)**
  - ALU Op
  - ALU Control (2:0)
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>
Review: ALU

<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>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>
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_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>0</td>
<td>X</td>
<td>XX</td>
<td>1</td>
</tr>
</tbody>
</table>
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.
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

\[
N \times \text{CPI} \times (1/f) = N \times \text{CPI} \times T \text{ seconds}
\]
How can I Make the Program Run Faster?

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

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

- 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 (lw)
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>$t_{setup}$</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:

Execution Time = # instructions x CPI x TC
= (100 \times 10^9)(1)(925 \times 10^{-12} \text{ s})
= 92.5 \text{ seconds}
What Did We Learn?

- How to determine the performance of processors
  - CPI
  - Clock speed
  - Number of instructions

- Single-cycle MIPS architecture
  - Learned individual components
    - ALU
    - Data Memory
    - Instruction Memory
    - Register File
  - Discussed implementation of some instructions
  - Explained the control flow