EC 513
Computer Architecture

Single-cycle ISA Implementation

Prof. Michel A. Kinsky
Computer System View

- Applications
- Operating System
  - Compiler
  - Firmware

ISA

- Processor
- Memory organization
- I/O system

Datapath & Control

- Digital Design
- Circuit Design
- Layout
Role of the Instruction Set
Instruction Set Architecture (ISA)

- Instructions are the language the computer understand
- Instruction Set is the vocabulary of that language
- It serves as the hardware/software interface
  - Defines instruction format (bit encoding)
    - Number of explicit operands per instruction
    - Operand location
    - Number of bits per instruction
    - Instruction length: fixed, short, long, or variable, ...
  - Examples: MIPS, Alpha, x86, IBM 360, VAX, ARM, JVM
Instruction Set Architecture (ISA)

- Many possible implementations of the same ISA
  - 360 implementations: model 30 (c. 1964), z900 (c. 2001)
  - x86 implementations: 8086 (c. 1978), 80186, 286, 386, 486, Pentium, Pentium Pro, Pentium-4, Core i7, AMD Athlon, AMD Opteron, Transmeta Crusoe, SoftPC
  - MIPS implementations: R2000, R4000, R10000, ...
  - JVM: HotSpot, PicoJava, ARM Jazelle, ...
ISA and Performance

- Instructions per program depends on source code, compiler technology and ISA
- Cycles per instructions (CPI) depends upon the ISA and the microarchitecture
- Time per cycle depends upon the microarchitecture and the base technology

\[
\text{Time} = \frac{\text{Instructions}}{\text{Cycles}} \times \text{Time}
\]

Program Program * Instruction * Cycle
Processor Performance

- Instructions per program depends on source code, compiler technology and ISA
- Cycles per instructions (CPI) depends upon the ISA and the microarchitecture
- Time per cycle depends upon the microarchitecture and the base technology

<table>
<thead>
<tr>
<th>Microarchitecture</th>
<th>CPI</th>
<th>cycle time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Microcoded</td>
<td>&gt;1</td>
<td>short</td>
</tr>
<tr>
<td><strong>Single-cycle unpipelined</strong></td>
<td>1</td>
<td><strong>long</strong></td>
</tr>
<tr>
<td>Pipelined</td>
<td>1</td>
<td>short</td>
</tr>
</tbody>
</table>
Brief Overview of the RISC-V ISA

- A new, open, free ISA from Berkeley
- Several variants
  - RV32, RV64, RV128 – Different data widths
  - ‘I’ – Base Integer instructions
  - ‘M’ – Multiply and Divide
  - ‘A’ – Atomic memory instructions
  - ‘F’ and ‘D’ – Single and Double precision floating point
  - ‘V’ – Vector extension
  - And many other modular extensions
- We will focus on the RV32I the base 32-bit variant
RV32I Register State

- 32 general purpose registers (GPR)
  - x0, x1, …, x31
  - 32-bit wide integer registers
  - x0 is hard-wired to zero
- Program counter (PC)
  - 32-bit wide
- CSR (Control and Status Registers)
  - User-mode
    - cycle (clock cycles) // read only
    - instret (instruction counts) // read only
  - Machine-mode
    - hartid (hardware thread ID) // read only
    - mepc, mcause etc. used for exception handling
  - Custom
    - mtohost (output to host) // write only – custom extension
Instruction Types

- Register-to-Register Arithmetic and Logical operations
- Control Instructions alter the sequential control flow
- Memory Instructions move data to and from memory
- CSR Instructions move data between CSRs and GPRs; the instructions often perform read-modify-write operations on CSRs
- Privileged Instructions are needed by the operating systems, and most cannot be executed by user programs
Instruction Formats

- **R-type instruction**

<table>
<thead>
<tr>
<th>7</th>
<th>5</th>
<th>5</th>
<th>3</th>
<th>5</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>funct7</td>
<td>rs2</td>
<td>rs1</td>
<td>funct3</td>
<td>rd</td>
<td>opcode</td>
</tr>
</tbody>
</table>

- **I-type instruction & I-immediate (32 bits)**

<table>
<thead>
<tr>
<th>12</th>
<th>5</th>
<th>3</th>
<th>5</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>imm[11:0]</td>
<td>rs1</td>
<td>funct3</td>
<td>rd</td>
<td>opcode</td>
</tr>
</tbody>
</table>

  - I-imm = signExtend(inst[31:20])

- **S-type instruction & S-immediate (32 bits)**

<table>
<thead>
<tr>
<th>7</th>
<th>5</th>
<th>5</th>
<th>3</th>
<th>5</th>
<th>7</th>
</tr>
</thead>
</table>

  - S-imm = signExtend({inst[31:25], inst[11:7]})
Datapath: Reg-Reg ALU Instructions

rd ← (rs) func (rt)
Datapath: Reg-Imm ALU Instructions

\[
\begin{array}{cccc}
\text{imm}[11:0] & \text{rs1} & \text{funct3} & \text{rd} & \text{opcode} \\
12 & 5 & 3 & 5 & 7 \\
\end{array}
\]

\(rt \leftarrow (rs) \text{ op immediate}\)
Conflicts in Merging Datapath

\[ \text{rd} \leftarrow (\text{rs}) \text{ func (rt)} \]

\[ \text{rt} \leftarrow (\text{rs}) \text{ op immediate} \]
Conflicts in Merging Datapath

### Opcode Analysis

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>funct7</td>
<td>7</td>
</tr>
<tr>
<td>rs2</td>
<td>5</td>
</tr>
<tr>
<td>rs1</td>
<td>5</td>
</tr>
<tr>
<td>funct3</td>
<td>3</td>
</tr>
<tr>
<td>rd</td>
<td>5</td>
</tr>
<tr>
<td>imm[11:0]</td>
<td>7</td>
</tr>
<tr>
<td>rs1</td>
<td>5</td>
</tr>
<tr>
<td>funct3</td>
<td>3</td>
</tr>
<tr>
<td>rd</td>
<td>5</td>
</tr>
<tr>
<td>opcode</td>
<td>7</td>
</tr>
</tbody>
</table>

**Legend:**
- \( rt \leftarrow (rs) \text{ op immediate} \)
- \( rd \leftarrow (rs) \text{ func (rt)} \)
Instruction Formats

- **SB-type instruction & B-immediate (32 bits)**

<table>
<thead>
<tr>
<th>b7</th>
<th>b6</th>
<th>b5</th>
<th>b4</th>
<th>b3</th>
<th>b2</th>
<th>b1</th>
<th>b0</th>
</tr>
</thead>
</table>

  - B-imm = signExtend({inst[31], inst[7], inst[30:25], inst[11:8], 1'b0})

- **U-type instruction & U-immediate (32 bits)**

<table>
<thead>
<tr>
<th>b20</th>
<th>b5</th>
<th>b7</th>
</tr>
</thead>
<tbody>
<tr>
<td>imm[31:12]</td>
<td>rd</td>
<td>opcode</td>
</tr>
</tbody>
</table>

  - U-imm = signExtend({inst[31:12], 12'b0})

- **UJ-type instruction & J-immediate (32 bits)**

<table>
<thead>
<tr>
<th>b1</th>
<th>b10</th>
<th>b8</th>
<th>b5</th>
<th>b7</th>
</tr>
</thead>
</table>

  - J-imm = signExtend({inst[31], inst[19:12], inst[20], inst[30:21], 1'b0})
Computational Instructions

- Register-Register instructions (R-type)

<table>
<thead>
<tr>
<th>7</th>
<th>5</th>
<th>5</th>
<th>3</th>
<th>5</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>funct7</td>
<td>rs2</td>
<td>rs1</td>
<td>funct3</td>
<td>rd</td>
<td>opcode</td>
</tr>
</tbody>
</table>

- **opcode=OP**: \( \text{rd} \leftarrow \text{rs1 (funct3, funct7)} \ \text{rs2} \)
- **funct3 = SLT/SLTU/AND/OR/XOR/SLL**
- **funct3 = ADD**
  - funct7 = 0000000: \( \text{rs1} + \text{rs2} \)
  - funct7 = 0100000: \( \text{rs1} - \text{rs2} \)
- **funct3 = SRL**
  - funct7 = 0000000: logical shift right
  - funct7 = 0100000: arithmetic shift right
Computational Instructions

- Register-immediate instructions (I-type)

<table>
<thead>
<tr>
<th></th>
<th>12</th>
<th>5</th>
<th>3</th>
<th>5</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>imm[11:0]</td>
<td>rs1</td>
<td>funct3</td>
<td>rd</td>
<td>opcode</td>
<td></td>
</tr>
</tbody>
</table>

- opcode = OP-IMM: rd \( \leftarrow \) rs1 (funct3) I-imm
- I-imm = signExtend(inst[31:20])
- funct3 = ADDI/SLTI/SLTIU/ANDI/ORI/XORI

- A slight variant in coding for shift instructions - SLLI / SRLI / SRAI
  - rd \( \leftarrow \) rs1 (funct3, inst[30]) I-imm[4:0]
Computational Instructions

- Register-immediate instructions (U-type)

<table>
<thead>
<tr>
<th>20</th>
<th>5</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>imm[31:12]</td>
<td>rd</td>
<td>opcode</td>
</tr>
</tbody>
</table>

- opcode = LUI : rd $\leftarrow$ U-imm
- opcode = AUIPC : rd $\leftarrow$ pc + U-imm
- U-imm = \{inst[31:12], 12'b0\}
Control Instructions

- Unconditional jump and link (UJ-type)

<table>
<thead>
<tr>
<th>1</th>
<th>10</th>
<th>1</th>
<th>8</th>
<th>5</th>
<th>7</th>
</tr>
</thead>
</table>

- opcode = JAL: rd ← pc + 4; pc ← pc + J-imm
- J-imm = signExtend({inst[31], inst[19:12], inst[20], inst[30:21], 1'b0})
- Jump ±1MB range

- Unconditional jump via register and link (l-type)

<table>
<thead>
<tr>
<th>12</th>
<th>5</th>
<th>3</th>
<th>5</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>imm[11:0]</td>
<td>rs1</td>
<td>funct3</td>
<td>rd</td>
<td>opcode</td>
</tr>
</tbody>
</table>

- opcode = JALR: rd ← pc + 4; pc ← (rs1 + l-imm) & ~0x01
- l-imm = signExtend(inst[31:20])
Control Instructions

- Conditional branches (SB-type)

<table>
<thead>
<tr>
<th>1</th>
<th>6</th>
<th>5</th>
<th>5</th>
<th>3</th>
<th>4</th>
<th>1</th>
<th>7</th>
</tr>
</thead>
</table>

- opcode = BRANCH: pc ← compare(funct3, rs1, rs2) ? pc + B-imm : pc + 4
- B-imm = signExtend({inst[31], inst[7], inst[30:25], inst[11:8], 1'b0})
- Jump ±4KB range
- funct3 = BEQ/BNE/BLT/BLTU/BGE/BGEU
Conditional Branches (BEQ/BNE/BLT/BGE/BLTU/BGEU)
# Load & Store Instructions

## Load (l-type)

<table>
<thead>
<tr>
<th></th>
<th>12</th>
<th>5</th>
<th>3</th>
<th>5</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>imm[11:0]</td>
<td>rs1</td>
<td>funct3</td>
<td>rd</td>
<td>opcode</td>
</tr>
</tbody>
</table>

- **opcode** = LOAD: \( rd \leftarrow \text{mem}[rs1 + l\text{-imm}] \)
- **l-imm** = signExtend(inst[31:20])
- **funct3** = LW/LB/LBU/LH/LHU

## Store (S-type)

<table>
<thead>
<tr>
<th></th>
<th>7</th>
<th>5</th>
<th>5</th>
<th>3</th>
<th>5</th>
<th>7</th>
</tr>
</thead>
</table>

- **opcode** = STORE: \( \text{mem}[rs1 + S\text{-imm}] \leftarrow rs2 \)
- **S-imm** = signExtend({inst[31:25], inst[11:7]})
- **funct3** = SW/SB/SH
Datapath for Memory Instructions

- Should program and data memory be separate?
  - Harvard style: separate (Aiken and Mark 1 influence)
    - read-only program memory
    - read/write data memory
  - Princeton style: the same (von Neumann’s influence)
    - single read/write memory for program and data
      - Executing a Load or Store instruction requires accessing the memory more than once
Harvard Architecture

Store (rs) + displacement
Load
Hardwired Control

- Hardwired Control is pure Combinational Logic

![Diagram of Hardwired Control](image)

- op code
- equal?

Combinational Logic

- ImmSel
- Op2Sel
- FuncSel
- MemWrite
- WBSel
- RegDst
- RegWrite
- PCSel
ALU Control & Immediate Extension

Inst<14:12> (Func3)
Inst<6:0> (Opcode)

+ 0?

FuncSel (Func, Op, +, 0?)

ImmSel (IType_{12}, SType_{12}, UType_{20})

ALUop

Decode Map
# Hardwired Control Table

<table>
<thead>
<tr>
<th>Opcode</th>
<th>ImmSel</th>
<th>Op2Sel</th>
<th>FuncSel</th>
<th>MemWr</th>
<th>RFWen</th>
<th>WBSel</th>
<th>WASel</th>
<th>PCSel</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU</td>
<td>*</td>
<td>Reg</td>
<td>Func</td>
<td>no</td>
<td>yes</td>
<td>ALU</td>
<td>rd</td>
<td>pc+4</td>
</tr>
<tr>
<td>ALUi</td>
<td>IType\textsubscript{12}</td>
<td>Imm</td>
<td>Op</td>
<td>no</td>
<td>yes</td>
<td>ALU</td>
<td>rd</td>
<td>pc+4</td>
</tr>
<tr>
<td>LW</td>
<td>IType\textsubscript{12}</td>
<td>Imm</td>
<td>+</td>
<td>no</td>
<td>yes</td>
<td>Mem</td>
<td>rd</td>
<td>pc+4</td>
</tr>
<tr>
<td>SW</td>
<td>SType\textsubscript{12}</td>
<td>Imm</td>
<td>+</td>
<td>yes</td>
<td>no</td>
<td>*</td>
<td>*</td>
<td>pc+4</td>
</tr>
<tr>
<td>BEQ\textsubscript{true}</td>
<td>SBTType\textsubscript{12}</td>
<td>*</td>
<td>*</td>
<td>no</td>
<td>no</td>
<td>*</td>
<td>*</td>
<td>br</td>
</tr>
<tr>
<td>BEQ\textsubscript{false}</td>
<td>SBTType\textsubscript{12}</td>
<td>*</td>
<td>*</td>
<td>no</td>
<td>no</td>
<td>*</td>
<td>*</td>
<td>pc+4</td>
</tr>
<tr>
<td>J</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>no</td>
<td>no</td>
<td>*</td>
<td>*</td>
<td>jabs</td>
</tr>
<tr>
<td>JAL</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>no</td>
<td>yes</td>
<td>PC</td>
<td>X1</td>
<td>jabs</td>
</tr>
<tr>
<td>JALR</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>no</td>
<td>yes</td>
<td>PC</td>
<td>rd</td>
<td>rind</td>
</tr>
</tbody>
</table>
Single-Cycle Hardwired Control

- Harvard architecture: we will assume that
  - clock period is sufficiently long for all of and the following steps to be “completed”:
    - 1. instruction fetch
    - 2. decode and register fetch
    - 3. ALU operation
    - 4. data fetch if required
    - 5. register write-back setup time
  - \( t_C > t_{IFetch} + t_{RFetch} + t_{ALU} + t_{DMem} + t_{RWB} \)
Princeton Microarchitecture

Fetch phase

Department of Electrical & Computer Engineering
Two-State Controller

- In the Princeton Microarchitecture, a flipflop can be used to remember the phase

**fetch phase**

- AddrSel = PC
- IRen = on
- PCen = off
- Wen = off

**execute phase**

- AddrSel = ALU
- IRen = off
- PCen = on
- Wen = on
Hardwired Controller

IR → op code

old combinational logic (Harvard)

equal?

new combinational logic

S → 1-bit Toggle FF
1-fetch / Execute

MemWrite, ImmSel, Op2Sel, FuncSel, MemWrite, WBSel, RegDst, RegWrite, PCSel

RegWrite

Wen

Pcen

Iren

AddrSel
Clock Period

- Princeton architecture
  - $t_{C-Princeton} > \max\{t_M, t_{RF} + t_{ALU} + t_M + t_{WB}\}$
  - $t_{C-Princeton} > t_{RF} + t_{ALU} + t_M + t_{WB}$

- while in the hardwired Harvard architecture
  - $t_{C-Harvard} > t_M + t_{RF} + t_{ALU} + t_M + t_{WB}$

- which will execute instructions faster?
Clock Rate vs CPI

- Suppose $t_M \gg t_{RF} + t_{ALU} + t_{WB}$
  - $t_{C-Princeton} = 0.5 \times t_{C-Harvard}$

- $CPI_{Princeton} = 2$
- $CPI_{Harvard} = 1$

- No difference in performance!
Can we overlap instruction fetch and execute?
Princeton Microarchitecture

- Only one of the phases is active in any cycle
  - a lot of datapath is not in use at any given time

*fetch phase*

*execute phase*

The same
(mux not shown)
Stalling the instruction fetch

- When stall condition is indicated
  - don’t fetch a new instruction and don’t change the PC
  - insert a nop in the IR
  - set the Memory Address mux to ALU (not shown)
Pipelined Princeton Architecture

- **Clock:** \( t_{C\text{-Princeton}} > t_{RF} + t_{ALU} + t_M \)

- **CPI:** \((1 - f) + 2f\) cycles per instruction where \( f \) is the fraction of instructions that cause a stall
Instructions to Read and Write CSR

- opcode = `SYSTEM`
- `CSRW rs1, csr (funct3 = CSRRW, rd = x0): csr ← rs1`
- `CSRR csr, rd (funct3 = CSRRS, rs1 = x0): rd ← csr`
GCD in C

```c
// require: x >= 0 && y > 0
int gcd(int a, int b) {
    int t;
    while(a != 0) {
        if(a >= b) {
            a = a - b;
        } else {
            t = a; a = b; b = t;
        }
    }
    return b;
}
```
GCD in RISC-V Assembler

// a: x1, b: x2, t: x3
begin:
    beqz x1, done       // if(x1 == 0) goto done
    blt x1, x2, bigger  // if(x1 < x2) goto b_bigger
    sub x1, x1, x2     // x1 := x1 - x2
    j begin            // goto begin
bigger:
    mv x3, x1          // x3 := x1
    mv x1, x2          // x1 := x2
    mv x2, x3          // x2 := x3
    j begin            // goto begin
done:                // now x2 contains the gcd
GCD in RISC-V Assembler

// a: a0, b: a1, t: t0

gcd:
  beqz a0, done
  blt a0, a1, bigger
  sub a0, a0, a1
  j gcd

bigger:
  mv t0, a0
  mv a0, a1
  mv a1, t0
  j gcd

done:        // now a1 contains the gcd
  mv a0, a1   // move to a0 for returning
  ret         // jr ra
Exception handling

- When an exception is caused
  - Hardware saves the information about the exception in CSRs:
    - mepc – exception PC
    - mcause – cause of the exception
    - mstatus.mpp – privilege mode of exception
  - Processor jumps to the address of the trap handler (stored in the mtvec CSR) and increases the privilege level
- An exception handler, a software program, takes over and performs the necessary action
Software for interrupt handling

- Hardware transfers control to the common software interrupt handler (CH) which:
  1. Saves all GPRs into the memory pointed by mscratch
  2. Passes mcause, mepc, stack pointer to the IH (a C function) to handle the specific interrupt
  3. On the return from the IH, writes the return value to mepc
  4. Loads all GPRs from the memory
  5. Execute ERET, which does:
     - set pc to mepc
     - pop mstatus (mode, enable) stack
Single Cycle RISC-V Processor
Single Cycle RISC-V Processor

<table>
<thead>
<tr>
<th>ALU control lines</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>AND</td>
</tr>
<tr>
<td>0001</td>
<td>OR</td>
</tr>
<tr>
<td>0010</td>
<td>add</td>
</tr>
<tr>
<td>0110</td>
<td>subtract</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction opcode</th>
<th>ALUOp</th>
<th>operation</th>
<th>Funct7 field</th>
<th>Funct3 field</th>
<th>Desired ALU action</th>
<th>ALU control input</th>
</tr>
</thead>
<tbody>
<tr>
<td>Id</td>
<td>00</td>
<td>load doubleword</td>
<td>XXXXXXXX</td>
<td>XXX</td>
<td>add</td>
<td>0010</td>
</tr>
<tr>
<td>sd</td>
<td>00</td>
<td>store doubleword</td>
<td>XXXXXXXX</td>
<td>XXX</td>
<td>add</td>
<td>0010</td>
</tr>
<tr>
<td>beq</td>
<td>01</td>
<td>branch if equal</td>
<td>XXXXXXXX</td>
<td>XXX</td>
<td>subtract</td>
<td>0110</td>
</tr>
<tr>
<td>R-type</td>
<td>10</td>
<td>add</td>
<td>00000000</td>
<td>000</td>
<td>add</td>
<td>0010</td>
</tr>
<tr>
<td>R-type</td>
<td>10</td>
<td>sub</td>
<td>01000000</td>
<td>000</td>
<td>subtract</td>
<td>0110</td>
</tr>
<tr>
<td>R-type</td>
<td>10</td>
<td>and</td>
<td>00000000</td>
<td>111</td>
<td>AND</td>
<td>0000</td>
</tr>
<tr>
<td>R-type</td>
<td>10</td>
<td>or</td>
<td>00000000</td>
<td>110</td>
<td>OR</td>
<td>0001</td>
</tr>
</tbody>
</table>
### Single Cycle RISC-V Processor

#### ALUOp Table

<table>
<thead>
<tr>
<th>ALUOp</th>
<th>ALUOp0</th>
<th>Funct7 field</th>
<th>Funct3 field</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>X</td>
<td>1</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>1</td>
<td>X</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>X</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>X</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>X</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

#### Instruction Table

<table>
<thead>
<tr>
<th>Instruction</th>
<th>ALUSrc</th>
<th>Memto-Reg</th>
<th>Reg-Write</th>
<th>Mem-Read</th>
<th>Mem-Write</th>
<th>Branch</th>
<th>ALUOp1</th>
<th>ALUOp0</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-format</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>ld</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>sd</td>
<td>1</td>
<td>X</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>beq</td>
<td>0</td>
<td>X</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
Single Cycle RISC-V Processor

<table>
<thead>
<tr>
<th>Input or output</th>
<th>Signal name</th>
<th>R-format</th>
<th>ld</th>
<th>sd</th>
<th>beq</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inputs</td>
<td>I[6]</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>I[5]</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>I[4]</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>I[3]</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>I[2]</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>I[1]</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>I[0]</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Outputs</td>
<td>ALUSrc</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>MemtoReg</td>
<td>0</td>
<td>1</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>RegWrite</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>MemRead</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>MemWrite</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Branch</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>ALUOp1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>ALUOp0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
## Single Cycle RISC-V Processor

<table>
<thead>
<tr>
<th>Signal name</th>
<th>Effect when deasserted</th>
<th>Effect when asserted</th>
</tr>
</thead>
<tbody>
<tr>
<td>RegWrite</td>
<td>None.</td>
<td>The register on the Write register input is written with the value on the Write data input.</td>
</tr>
<tr>
<td>ALUSrc</td>
<td>The second ALU operand comes from the second register file output (Read data 2).</td>
<td>The second ALU operand is the sign-extended, 12 bits of the instruction.</td>
</tr>
<tr>
<td>PCSrc</td>
<td>The PC is replaced by the output of the adder that computes the value of PC + 4.</td>
<td>The PC is replaced by the output of the adder that computes the branch target.</td>
</tr>
<tr>
<td>MemRead</td>
<td>None.</td>
<td>Data memory contents designated by the address input are put on the Read data output.</td>
</tr>
<tr>
<td>MemWrite</td>
<td>None.</td>
<td>Data memory contents designated by the address input are replaced by the value on the Write data input.</td>
</tr>
<tr>
<td>MemtoReg</td>
<td>The value fed to the register Write data input comes from the ALU.</td>
<td>The value fed to the register Write data input comes from the data memory.</td>
</tr>
</tbody>
</table>
Next Class

- Instruction Pipelining and Hazards