CSE220 - MIPS Instruction Set Architecture & Single-Cycle Datapath

- We have studied how to build various building blocks, such as the ALU.
- Next we will examine how to connect these building blocks together to build a working processor and how to control the movement of data through the datapath (control signals). The way the components are connected creates the Processor Datapath.
- The Goal of the Datapath design:
  - Use minimal amounts of hardware (cost)
  - Execute the instructions (specified by ISA) efficiently (best performance).

MIPS Instruction Set Architecture (MIPS ISA) - Recap
- MIPS stands for “Microprocessor without Interlocked Pipeline Stages"
- Designed in early 1980’s by John Hennessy (SB Alum, current president of Stanford University)
- MIPS is known as a Reduced Instruction Set Computer (RISC) machine. This is opposed to a CISC machine (Complex Instruction Set Computer).
  - RISC machines have simple instructions, simple memory access, and simple instruction formats to keep the hardware simple to try and reduce the number of clock cycles required to execute an instruction. This results in more instructions required per program, since each line of code is broken in to small simple pieces.
  - CISC machines have complex instructions, formats, and more ways to access memory. This increases the complexity of the hardware, but reduces the number of instructions required per program, since the hardware can compute on more complex statements.
  - There are other differences between RISC and CISC. A good illustrative resource article is available at: [http://cs.stanford.edu/people/eroberts/courses/soco/projects/risc/risccisc/](http://cs.stanford.edu/people/eroberts/courses/soco/projects/risc/risccisc/)
- Has been used in many products: Routers, cable and ADSL modems, smartcards, laser printer engines, set-top boxes, robots, handheld computers, Sony PS2 and PSP.
- For us, a major reason to study it is the simplicity of the instruction set.

- It is a load-store architecture, which means only load and store operations can access memory.
- It is a Register machine (32 general purpose registers), which means that the CPU only operates on data in the registers.
- The hardware provides at 32 general purpose registers plus 32 separate floating-point registers. These are fast memory on the CPU, and due to their speed are very expensive.
- All instructions are fixed in length, 32-bits wide. There are 3 instruction formats: Register, Immediate, Jump.
- MIPS supports 2 basic addressing modes: displacement (with an address offset size of 16 bits), and immediate (16 bits)
- MIPS supports these data sizes and types: 8-bit, 16-bit, 32-bit integers; and 32-bit and 64-bit IEEE 754 floating point numbers.
- We will study three versions of the MIPS ISA implementation:
  - Basic simplified version, single cycle datapath
  - Multi-cycle datapath
  - More realistic, pipelined multi-cycle datapath
- We will assume the MIPS ISA, but only a small subset of the MIPS instructions:
  - Memory instructions: load (lw)/store (sw)
- ALU instructions: add, sub, and, or, slt
- Control instructions: branch on equal (beq), jump (j)

**Example of a Basic MIPS Program: Hello World**

Below illustrates a basic help world program as it is presented in MIPS assembly language. There are basic instructions which use pneumonic “programmer friendly” naming conventions to reference the operation and the registers which are being used by each instruction.

Notice there are two main sections of this program, a .text and a .data section. Each of these defines where the MIPS instructions (.text) are defined and where the data (.data) for the program is defined.

```assembly
## hello.asm - print out "hello world"
# text Segment #
.text               # tells assembler program code starts here
.globl main        # defines label for execution start
main:              # execution starts here
    la $a0,str  # put string address into a0
    li $v0,4   # system call to print
    syscall   # out a string
    li $v0, 10 # Load exit syscall value
    syscall   # Exit
# data segment #
.data               # tells assembler data segment begins here
str: .asciiz "hello world\n" # declaration of a string
```

Each instruction is simply a 32-bit value. In bytecode, the above program is represented by

```
0011110000000001000100000000000001
0011010000100100000000000000000000
0010000000000100000000000000000100
0000000000000000000000000000000110
0010000000000100000000000000000101
0000000000000000000000000000000110
```

- Each line is a single MIPS instruction encoded into binary representation.
- Each instruction is stored in memory with an address associated with its position. MIPS memory is byte addressable, which means that each byte (8bits) has a unique address. MIPS instructions are each 32-bits, therefore each address must be a multiple of 4.
Abstract/Basic MIPS Single-Cycle Architecture

- The basic datapath is composed of the following main components:
  - Program counter, PC (the address of the instruction to execute)
  - Instruction memory (the .text section, program instructions)
  - Register File (32 32-bit registers, 2 read registers, 1 write register)
  - ALU (address calculation and operand arithmetic)
  - Data memory (the .data section, the data values)
  - In addition, there are two Adders used in the Fetch stage for PC=PC+4 and calculation of the branch addresses

- It is called the single cycle datapath, because each instruction is executed in 1 clock cycle of the processor.

- The register file is a unit which contains registers (connected groups of flip flops) to hold values from one clock cycle to the next. It is constructed in hardware as two portions, Read and Write:
Also, recall that we can build multiplexors for 32-bit values by combining 32 1-bit mux with the same control.

The ALU is as we designed earlier.

The instruction and data memory, take a single address and return the 32-bit value stored at this address. In addition, the data memory has the ability to write data to memory.

Before we can discuss the datapath we have to understand the format associated with each instruction in MIPS.

**RECAP: MIPS Instruction Formats**

- MIPS instructions are a fixed length of 32 bits. The 32 bits are split into fields.
- There are 3 types of instruction formats in MIPS: Register, Immediate, Jump.
**Register Format** (R-type)

- Used for arithmetic and logical AND, OR, shifts
  - Ex: add $3, $4, $5
  - Ex: and $10, $2, $6
  - Ex: sub $13, $12, $7
  - Ex: srl $10, $20, 10  #srl stands for shift right logical
- Fields have names:
  - **op**: basic operation of instruction, “opcode”
    - 91 opcodes in MIPS
    - 26 FP opcodes for single and double precision.
    - Compared to a CISC machine: 250 opcodes, 8 addressing modes
    - Compared to a JVM Machine: ~250 opcodes
  - **rs**: 1st register source operand.
  - **rt**: 2nd register source operand
  - **rd**: register destination operand, the result
  - **shamt**: shift amount (we will see how this is used later; assume it is 0 for now)
  - **funct**: function; selects the specific variant of the operation in the op field; sometimes called the function code.
    - 6 bits for the opcode means that there are 64 opcodes. However this is not enough, therefore the funct field is used. If the op code is 0 or 1, then the true instruction is determined by either the rt or funct field.
    - If 7 or 8 bits were used for the opcode, we could fit all the opcodes, but then we have to shorten all the remaining fields. This way we only use extra bits when we need them.

**Immediate Format** (I-type)

- Used for immediate instructions, data transfers, and branches.
  - Ex: lw $4, 16($2)
  - Ex: sw $12, -4($30)
  - Ex: beq $2, $4, labelA
- Allows for small operand values (signed, range $\pm 2^{15}$) to be stored in the instruction address field. This value can represent a constant value or an address
- Allowing a constant value to be in the instruction eliminates the need for a memory access.
- Fields have names:
• op: basic operation of instruction, “opcode”.
• rs: 1st register source operand.
• rt: 2nd register source operand (sw/beq) or destination operand (lw).
  • To make it easier for the hardware, the first 3 fields are the same in R and I format. However that means that the rt field changes meaning. The hardware knows which format is which based on the distinct values in the opcode field.

• **Jump Format** (J-type)
  • Used for the jump instruction.
    • Ex: j label
  • 26 bits is enough to store a word address. Multiply it by 4 to get the byte address [26 bits → 28 bits].
    • Note: that all values which are multiples of 4, word size, will always have 2 zeros in the lowest bits. Therefore it is redundant to add them to the instruction.
    • These bits are placed in the lower 28 bits of PC. The upper 4 bits of the PC are unchanged.
  • Since leftmost 4 bits of PC do not change, this kind of addressing is effective/correct if the program is placed within a block of $2^{28} = 256$ MB properly.

  • If you want to jump beyond that, then use jump register instruction (j r).
    • Place the address in a register before jumping → full 32 bits.

  • When we use 26 bit field the addressing mode is know as *pseudo-direct* (it is not ‘direct’ because 4 bits are from the existing PC value).

**MIPS Memory Addressing**
• How are the addresses of the operands specified?
  o 3 operand instructions would require 32bits*3 = 96 bits to represent the operands if stored in memory rather than registers. This is too long and not desirable.
  o Registers make things easy, only a few bits are necessary to specify a register.
  o When operand is present in register/or when destination is a register this is called *register addressing*.

**MIPS Addressing Modes**
• Depending on the instruction and the instruction format, the memory location or the address/data to be retrieved can vary.
• We have already seen the *pseudo-direct* addressing mode (in the context of the J-type instruction format used for jump instructions).
• There are four more types of addressing modes: *Register, Immediate, Base+index* and *PC-relative*. 
• **Base+index Mode**

  - When a register is specified inside a pair of parenthesis, then its contents are to be treated as an address.
  - **Ex:** `lw $t0, c($t1)` # load the word at the memory address specified in register $t1 + c into $t0.
  - The immediate, c, can be used. It is a signed 16 bit number (range $\pm2^{15}$). This is in byte quantity. Why? Because we can not reference data in memory any smaller than a byte.
  - The address in the register is known as the *base address*.

• **PC-Relative**

  - PC-relative is used to specify branch/jump instructions.
  - This is a better approach than storing the absolute address in the instruction. We don’t have enough space in our 32 bit instruction.
  - PC\_branch + 4 + (immediate value*4) is the target address, where PC\_branch is the address of the branch instruction in memory (*i.e.*, the value of the PC when we start to *Fetch-Decode-Execute* cycle for the branch instruction).
    - The immediate value is in number of words. This allows us to increase the range specified in the instruction by 4 times.
    - The immediate value is an offset, or the number of instructions before or after the current instruction.
  - Conditional branch instructions use this format.
  - **Ex:**
    
    Loop:  
    
    ```
    add $t1, $s3, $s3  
    ....  
    bne $t0, $s5, Exit  
    add $s3, $s4, $s5  
    j Loop  
    ```
    
    Exit:  ....
    
    - Exit label is used in the instruction bne. What is the offset here? 3
    - Why? From the address of bne the location Exit is 3 instructions/words further down.
    - Since the PC has been incremented in the Fetch cycle of bne to point to the next instruction, the offset is only 2 words or 8 bytes further.
    - Thus, the assembled instruction for bne should look like:
      - opcode: 5, rs: 8, rt: 21, offset: 2
    - Target address = address of Exit = (PC\_branch + 4) + 2*4 = PC\_updated + 2*4, where PC\_updated is the value of the PC if the branch instruction is taken.
    - Note: If we store the offset in bytes the range will be reduced by a factor of 4.
Fetch- Decode-Execute

When executing each instruction in the datapath the operation can be split into three phases: Fetch-Decode-Execute.

Instruction Fetch
- All instructions to be executed in the datapath must first be fetched from memory
- The fetch phase consists of 2 parts:
  o Fetch instruction from Instruction memory (PC contains current address)
  o PC = PC + 4
    - We increment the PC to point to the next instruction to execute in the program.
    - Remember that memory is byte addressable, not Word addressable. Each machine word is 32-bits, but because we want to be able to refer to bytes (for things like ASCII characters in memory). Each byte is 8-bits, and therefore each word contains 4 bytes. Memory is byte addressable, meaning that each byte has a unique address. Since instructions are only in word size, then to refer to each word, we must increment the address by 4-bytes.
- This section of the datapath does not change depending on the instruction type. We ALWAYS fetch the instruction and increment the PC.

Decode-Execute stages
- Each of these stages is different depending on the type of instruction which you are executing.
- Remember the type of instruction is specified by the opcode field in the instruction. We decode the instruction (read these bits) and determine how to control the rest of the datapath. We will discuss the control more later.
- Once the instruction is decode, we can perform all required operations for the instruction. This is known as the Execute phase.
The above datapath shows how the components are combined to perform the basic instructions (lw, sw, beq, j, R-type).

- The instruction is the 32-bit instruction read from the instruction memory (fetch).
- Pieces of the instruction (given by the instruction formats: Register, Immediate, Jump) are used to specify the values to the main components (Register File, ALU, Data Memory).
- The sign-extension (16→32 bits) is required because the ALU takes two 32-bit operands. The immediate instruction format only contains a 16-bit constant. The values must be sign extended (2’s complement format) so the ALU operations can be performed.
- The additional adder (PCBranch) is used to calculate the branch address.
- The blue signals are the control signals which specify how each component is to operate for this instruction. We will discuss the control later.

Consider first the Load/Store instructions. Break down the instructions into their pieces.

- These instructions are the only ones which access the data memory.
- They add an offset to the contents of a base register to calculate the memory address.
- For a load, we read from memory. For a store, we write to memory.
- Ex: lw $t0, 8($s0)   # Reg[$t0] = Mem[Reg[$s0] + 8]
- Ex: sw $t1, 16($t2) # Mem[Reg[$t2] + 16] = Reg[$t1]
- These instructions are in Immediate format and their operation can be split into the following steps.

*Note that load takes more “time” than store*
Load | Store
---|---
Instruction Fetch | Instruction Fetch
Register Read ($s0) | Register Read ($t1, $t2)
Calculate Address using ALU (Reg[$s0]+8) | Calculate Address using ALU (Reg[$t2]+16)
Read from memory (Mem[Reg[$s0] + 8]) | Write to memory (Mem[Reg[$t2] + 16] = Reg[$t1])
Write back to a register ($t0)

• For Load:
  o lw $t0, 8($s0)  # Reg[$t0] = Mem[Reg[$s0] + 8]
  o Immediate format:

<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>

  o The rs register ($s0) is read from the register file and the value in the register (Reg[$s0]) is available on Read data 1 (RD1).
  o To this value the immediate value (after sign extension), is added using the ALU, Reg[$s0]+8
  o This is now the address of the data in memory, the data memory is then read at this address and the value is output on the read data line of the data memory (Mem[Reg[$s0] + 8]).
  o This value is stored back in the register file (Reg[$t0]). The data to store is the write data line, and the register to store it at ($t0) on the write register line.

  o Starting at the Program Counter (PC) we trace the information we have to each unit, highlighting the pieces of the unit which are used by the instruction to obtain/calculate the information needed to complete the instruction operation.

![Diagram of memory and register file with control units]
- For Store:
  - $sw \ $t1, 16($t2) \ # \ Mem[Reg[$t2] + 16] = \ Reg[$t1]$
  - Immediate format:
    - The rs register (Reg[$t2]) and the rt register (Reg[$t1], the data to store) are read from the register file. Reg[$t2] is available on Read data 1 (RD1). Ref[$t1] on the Read data 2 (RD2).
    - To the Reg[$t2] (RD1) value the immediate value (after sign extension, SignImm), is added using the ALU Reg[$t2] + 16
    - This is the address where the data is to be stored, the value to be stored (Reg[$t1]) is on the write data (WD) line of the data memory.

- Consider ALU instructions (R-type instructions)
  - These instructions take two operands from registers and perform an ALU operation on them and store the value back in a register.
  - Ex: $add \ $t0, \ $s0 \ $s1 \ # \ Reg[$t0] = Reg[$s0] + Reg[$s1]$
  - These instructions are in Register format and their operation can be split into the following steps:

<table>
<thead>
<tr>
<th>R-type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction Fetch</td>
</tr>
<tr>
<td>Register Read (both)</td>
</tr>
<tr>
<td>Perform ALU operation</td>
</tr>
<tr>
<td>Write data to register</td>
</tr>
</tbody>
</table>

- Remember, the fetch phase is identical for all instructions.
- Many of the same components used by load and store are used. A few differences are:
  - The sign extension unit is not used. As there is no immediate value. The ALU performs operation on the RD1 and RD2 values.
- The Data memory is not required. The result value of the ALU is then written back to the Register file.
- Different values are passed to the ALU and Register File by selecting the proper MUX control signals.

Consider Branch instructions
- These instructions take two operands from registers and perform an ALU comparison operation on them, if the operation is “true” then the PC is replaced with the location of the next instruction to execute.
- Ex: beq $t0, $s0, label # if Reg[$t0] == Reg[$s0], then PC = PC + 4 + label
- The label value is really the immediate field from the instruction. The immediate value is the number of instructions to the new instruction from the beq (current instruction) - 1. The minus 1 is because at fetch we automatically do the PC +4, so when executing the branch you've already gone forward one instruction. So to branch 2 instructions forward the immediate value in the instruction should be 1. To move backward 3 instructions the immediate value should be -4.
- These instructions are in Immediate format and their operation can be split into the following steps:

<table>
<thead>
<tr>
<th>Branch</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction Fetch</td>
</tr>
<tr>
<td>Register Read (both)</td>
</tr>
<tr>
<td>Compare operands by</td>
</tr>
<tr>
<td>Performing ALU operation</td>
</tr>
<tr>
<td>If “true”, modify PC value</td>
</tr>
</tbody>
</table>

- Ex: for beq,
  - Read register operands
- Compare operands using ALU, subtract and check the Zero output (Zero is a value of 1 if the result of the ALU is 0).
- While the operands are being compared, the target address (which instruction to go to if “true”) is calculated by:
  - Sign-extend immediate value
  - Shift left 2 places (multiply by 4 to get word address)
  - Add value to PC + 4 value

  Note that for the branch operation, we must use the PCBranch adder to calculate the branch address.
  This is because the entire datapath must execute in a single clock cycle. This means that each component of the datapath can only perform 1 operation. Hardware cannot be reused for the same instruction.

**ALU Control**
- Depending on the instruction the ALU is used in different capacities:
  - Load/Store: add
  - Branch on Equal: subtract
  - R-type: depends on funct field
Because the ALU control is based on not only the opcode, but the function field (R-type), the control logic is split into 2 steps.

Based on the opcode a 2-bit control value, ALUOp, can be created. This value then combined with the function field produces the proper ALU Operation control.

<table>
<thead>
<tr>
<th>opcode</th>
<th>ALUOp</th>
<th>Operation</th>
<th>funct</th>
<th>ALU function</th>
<th>ALU control</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw</td>
<td>00</td>
<td>load word</td>
<td>XXXXXXX</td>
<td>add</td>
<td>0010</td>
</tr>
<tr>
<td>sw</td>
<td>00</td>
<td>store word</td>
<td>XXXXXXX</td>
<td>add</td>
<td>0010</td>
</tr>
<tr>
<td>beq</td>
<td>01</td>
<td>branch equal</td>
<td>XXXXXXX</td>
<td>subtract</td>
<td>0110</td>
</tr>
<tr>
<td>R-type</td>
<td>10</td>
<td>add</td>
<td>100000</td>
<td>add</td>
<td>0010</td>
</tr>
<tr>
<td></td>
<td></td>
<td>subtract</td>
<td>10010</td>
<td>subtract</td>
<td>0110</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AND</td>
<td>100100</td>
<td>AND</td>
<td>0000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>OR</td>
<td>100101</td>
<td>OR</td>
<td>0001</td>
</tr>
<tr>
<td></td>
<td></td>
<td>set-on-less-than</td>
<td>101010</td>
<td>set-on-less-than</td>
<td>0111</td>
</tr>
</tbody>
</table>

Control Unit

- The control logic is generated by the Control Unit.
- This unit is built using basic gates as we have learned since the start of the semester.
- We define each signal asserted = 1 or deasserted = 0.
- The control table for the current instruction sets as follows. The opcode is used to determine the operation to perform and mapped to a value for each control signal. The logic is split into units: Main decoder and ALU decoder. The ALU decoder uses the 2 bit ALUOp with the funct field of the R-type instructions to determine the ALU operation to perform.
Jump Instructions – separate “special” group

- Jump instructions perform an unconditional change to the PC based on the value in the instruction.
- Jump is different than a branch in the sense that we ARE NOT modifying the PC with respect to how many instructions fwd/bkwd we want to move. For jump the full 32 bit address cannot be stored in the instruction itself (need 6 bits for opcode). Therefore we can only specify 26 bits of the jump address in the instruction.
- We also know that all instructions are on word boundaries of memory (multiple of 4) and therefore the 2 least significant bits are always 00. We can use this to our advantage and not represent these 2 0’s in the instruction (waste of space, also allows for larger range of addresses to be represented/jumped to). So, if we specify the 26 bits in the instruction and then shift left by 2 (mult by 4) we have 28bits of the new address.
- We are still missing 4 bits. These must come from the current PC, where else could they come from... So the jump address is the top 4 bits of current PC+4, PC[31...28] appended to the 28bits (jump address in instruction shifted left by 2).
- This means when you use a jump instruction you can only jump to an address in the same section of memory. The upper 4 bits of the PC divide the memory into 16 different sections, and jump only allows you to move to any address within your current section. To jump to another section you need to use jump to register (which assumes the full 32-bit address to jump to is stored in a register).
- The destination address for a jump instruction is the concatenation of:
  o upper 4 bits of the current PC + 4
  o 26-bit address field in instruction
  o 00 as the 2 low-order bits (word address, not byte. Therefore multiply by 4.)

Extra control signal is required based on opcode in order to support jump

<table>
<thead>
<tr>
<th>Instr</th>
<th>Opcode</th>
<th>RegDst</th>
<th>RegWrite</th>
<th>ALUSrc</th>
<th>Mem Read</th>
<th>Mem Write</th>
<th>Memto Reg</th>
<th>Branch</th>
<th>ALUOp</th>
<th>Jump</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw</td>
<td>100011</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>sw</td>
<td>101011</td>
<td>X</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>X</td>
<td>0</td>
<td>0</td>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>R-type</td>
<td>000000</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>beq</td>
<td>000100</td>
<td>X</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>X</td>
<td>1</td>
<td>0</td>
<td>01</td>
<td>0</td>
</tr>
<tr>
<td>Jump</td>
<td>000010</td>
<td>X</td>
<td>x</td>
<td>0</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td>XX</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>