Advanced Computer Architecture
ELEC3219 (2017/18)

MIPS Architecture

Dr. Terrence Mak
tmak@ecs.soton.ac.uk
What Is MIPS

• MIPS is a reduced instruction set computer (RISC) instruction set architecture (ISA) developed by MIPS Technologies
  – Originally an acronym for Microprocessor without Interlocked Pipeline Stages
  – The early MIPS architectures were 32-bit, with 64-bit versions added later.
  – The current revisions are MIPS32 (for 32-bit implementations) and MIPS64 (for 64-bit implementations)
Instruction Set Architecture

- Instruction Set Architecture (ISA)
  - Usually defines a “family” of microprocessors
    - Examples: Intel x86 (IA32), Sun SPARC, DEC Alpha, IBM/360, IBM PowerPC, M68K, DEC VAX
  - Formally, it defines the interface between a user and a microprocessor

- ISA includes:
  - Instruction set
  - Rules for using instructions
    - Mnemonics, functionality, addressing modes
  - Instruction encoding

- ISA is a form of abstraction
  - Low-level details of microprocessor are “invisible” to user
Instruction Set Architecture

• Many processor implementation details are revealed through ISA
• Example:
  – Motorola 6800 / Intel 8085 (1970s)
    • 1-address architecture: ADDA <addr>
    • (A) = (A) + (addr)
  
  – Intel x86 (1980s)
    • 2-address architecture: ADD EAX, EBX
    • (A) = (A) + (B)
  
  – MIPS (1990s)
    • 3-address architecture: ADD $2, $3, $4
    • ($2) = ($3) + ($4)
R-I-J instructions

• **R instructions** are used when all the data values used by the instruction are located in registers.

• **I instructions** are used when the instruction must operate on an immediate value and a register value.

• **J instructions** are used when a jump needs to be performed.
MIPS instruction format – 32 bit long

- All MIPS instructions are 32 bits long. The three instruction formats:

<table>
<thead>
<tr>
<th>Field</th>
<th>R-type</th>
<th>I-type</th>
<th>J-type</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>6 bits</td>
<td>6 bits</td>
<td>6 bits</td>
</tr>
<tr>
<td>rs</td>
<td>5 bits</td>
<td>5 bits</td>
<td></td>
</tr>
<tr>
<td>rt</td>
<td>5 bits</td>
<td>5 bits</td>
<td></td>
</tr>
<tr>
<td>rd</td>
<td>5 bits</td>
<td></td>
<td></td>
</tr>
<tr>
<td>shamt</td>
<td>6 bits</td>
<td></td>
<td></td>
</tr>
<tr>
<td>funct</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- The different fields are:
  - op: operation of the instruction
  - rs, rt, rd: the source and destination register specifiers
  - shamt: shift amount
  - funct: selects the variant of the operation in the “op” field
  - address / immediate: address offset or immediate value
  - target address: target address of the jump instruction
# Let's look at a MIPS subset (R)

- **ADD and SUB**
  - `add rd, rs, rt`
  - `sub rd, rs, rt`

- **OR Immediate**
  - `ori rt, rs, imm16`

- **LOAD and STORE Word**
  - `lw rt, rs, imm16`
  - `sw rt, rs, imm16`

- **BRANCH**
  - `beq rs, rt, imm16`

### Instruction Formats

- **ADD and SUB**
  - **Op**
  - **Rs**
  - **Rt**
  - **Rd**
  - **Shamt**
  - **Funct**

- **OR Immediate**
  - **Op**
  - **Rs**
  - **Rt**
  - **Immediate**

- **LOAD and STORE Word**
  - **Op**
  - **Rs**
  - **Rt**
  - **Immediate**

- **BRANCH**
  - **Op**
  - **Rs**
  - **Rt**
  - **Immediate**
Types of instructions

• Arithmetic
  – Integer arithmetic/logic & floating point instructions
    • ADD, SUB, MULT
    • OR, AND, NOR, NAND
    • FADD, FMUL, FDIV

• Memory transfer instructions
  – Loads and Stores

• Control instructions
  – Jump
  – Conditional Branch
  – Call & Return
MIPS Arithmetic

- Most instructions have 3 operands
- Operand order is fixed (destination first)

Example:

C code: \[ A = B + C \]

MIPS code: `add $s0, $s1, $s2`

($s0, $s1 and $s2 are associated with variables by compiler, and each correspond to one of the 32 registers r0-r31)
MIPS Arithmetic

C code:  
\[
A = B + C + D; \\
E = F - A;
\]

MIPS code:  
```
add $t0, $s1, $s2  
add $s0, $t0, $s3  
sub $s4, $s5, $s0
```
Registers vs memory

• 32 registers in MIPS
• Compiler associates variables with registers
  – Compiler tries to keep as many variables in registers as possible
• Compiler may run out of registers => spilling
Memory

• Viewed as a large, single-dimension array, with an address
• A memory address is an index into the array
• “Byte addressing” means that successive addresses are one byte apart
Memory

• Bytes are nice, but most data items use larger “words”
• For MIPS, a word is 32 bits or 4 bytes.

- $2^{32}$ bytes with byte addresses from 0 to $2^{32}-1$
- $2^{30}$ words with byte addresses 0, 4, 8, ... $2^{32}-4$
MIPS memory transfer instructions

**Load and Store**

Example:


MIPS code:

\[
\begin{align*}
\text{lw} & \quad $t0, 32($s3) \\
\text{add} & \quad $t0, $s2, $t0 \\
\text{sw} & \quad $t0, 32($s3)
\end{align*}
\]

- Store word operation has no destination (reg) operand
- Remember arithmetic operands are registers, not memory!
MIPS control instructions

• Decision making instructions
  – alter the control flow,
  – i.e., change the "next" instruction to be executed

• MIPS conditional branch instructions:

  BNE $t0, $t1, Label
  BEQ $t0, $t1, Label

• Example: if (i==j) h = i + j;

  BNE $s0, $s1, Label
  ADD $s3, $s0, $s1
  Label: ....
MIPS control instructions

• MIPS unconditional branch instructions:
  j label

• Example:

```assembly
if (i!=j)    beq $s4, $s5, Lab1
  h=i+j;     add $s3, $s4, $s5
else
  j Lab2
  h=i-j;Lab1: sub $s3, $s4, $s5
Lab2: ...
```
So far (including J-type instr):

**Instruction** | **Meaning**
--- | ---
`add $s1,$s2,$s3` | $s1 = $s2 + $s3$
`sub $s1,$s2,$s3` | $s1 = $s2 - $s3$
`lw $s1,100($s2)` | $s1 = \text{Memory}[$s2+100$]
`sw $s1,100($s2)` | \text{Memory}[$s2+100$] = $s1$
`bne $s4,$s5,L` | Next instr. is at Label if $s4 \neq $s5
`beq $s4,$s5,L` | Next instr. is at Label if $s4 = $s5
`j Label` | Next instr. is at Label

**Formats:**

<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>op</td>
<td>rs</td>
<td>rt</td>
<td>16 bit address</td>
<td></td>
<td></td>
</tr>
<tr>
<td>op</td>
<td></td>
<td></td>
<td>26 bit address</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
MIPS Addressing Modes

• Immediate addressing
  – Operand is help as constant (literal) in instruction word
  – Example: **ADDI** $2, $3, 64

• Register addressing
  – MIPS addresses register operands using 5-bit field
  – Example: **ADD** $2, $3, $4

• PC-relative addressing
  – relative to next instruction (“PC relative”)
  – held in 16-bit offset in I-type
  – Example: **BEQ** $2, $3, 12
MIPS Addressing Modes

• Base addressing to addresses load/store locations
  – base register + 16-bit signed offset (byte addressed)
    • Example: `lw $2, 128($3)`
  – 16-bit direct address (base register is 0)
    • Example: `lw $2, 4092($0)`
  – indirect (offset is 0)
    • Example: `lw $2, 0($4)`

• Pseudo-direct addressing
  – MIPS addresses jump targets as register content or 26-bit
  – Example: `jr $31`
  `j 128`
1. Immediate addressing

```
op  rs  rt  Immediate
```

2. Register addressing

```
op  rs  rt  rd  ...  funct
```

3. Base addressing

```
op  rs  rt  Address
```

4. PC-relative addressing

```
op  rs  rt  Address
```

5. Pseudodirect addressing

```
op  Address
```

---

**MIPS addressing mode summary**
## MIPS compiler/assembler conventions

<table>
<thead>
<tr>
<th>Name</th>
<th>Register number</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>$zero</td>
<td>0</td>
<td>the constant value 0</td>
</tr>
<tr>
<td>$v0-$v1</td>
<td>2-3</td>
<td>values for results and expression evaluation</td>
</tr>
<tr>
<td>$a0-$a3</td>
<td>4-7</td>
<td>arguments</td>
</tr>
<tr>
<td>$t0-$t7</td>
<td>8-15</td>
<td>temporaries</td>
</tr>
<tr>
<td>$s0-$s7</td>
<td>16-23</td>
<td>saved (by callee)</td>
</tr>
<tr>
<td>$t8-$t9</td>
<td>24-25</td>
<td>more temporaries</td>
</tr>
<tr>
<td>$gp</td>
<td>28</td>
<td>global pointer</td>
</tr>
<tr>
<td>$sp</td>
<td>29</td>
<td>stack pointer</td>
</tr>
<tr>
<td>$fp</td>
<td>30</td>
<td>frame pointer</td>
</tr>
<tr>
<td>$ra</td>
<td>31</td>
<td>return address</td>
</tr>
<tr>
<td>Category</td>
<td>Instruction</td>
<td>Example</td>
</tr>
<tr>
<td>------------------</td>
<td>-------------</td>
<td>--------------------------</td>
</tr>
<tr>
<td><strong>Arithmetic</strong></td>
<td>add</td>
<td>add $s1, $s2, $s3</td>
</tr>
<tr>
<td></td>
<td>subtract</td>
<td>sub $s1, $s2, $s3</td>
</tr>
<tr>
<td></td>
<td>add immediate</td>
<td>addi $s1, $s2, 100</td>
</tr>
<tr>
<td></td>
<td>load word</td>
<td>lw $s1, 100($s2)</td>
</tr>
<tr>
<td></td>
<td>store word</td>
<td>sw $s1, 100($s2)</td>
</tr>
<tr>
<td><strong>Data transfer</strong></td>
<td>load byte</td>
<td>lb $s1, 100($s2)</td>
</tr>
<tr>
<td></td>
<td>store byte</td>
<td>sb $s1, 100($s2)</td>
</tr>
<tr>
<td></td>
<td>load upper immediate</td>
<td>lui $s1, 100</td>
</tr>
<tr>
<td><strong>Conditional</strong></td>
<td>branch on equal</td>
<td>beq $s1, $s2, 25</td>
</tr>
<tr>
<td></td>
<td>branch on not equal</td>
<td>bne $s1, $s2, 25</td>
</tr>
<tr>
<td></td>
<td>set on less than</td>
<td>slt $s1, $s2, $s3</td>
</tr>
<tr>
<td></td>
<td>set less than immediate</td>
<td>slti $s1, $s2, 100</td>
</tr>
<tr>
<td><strong>Jump</strong></td>
<td>jump</td>
<td>j 2500</td>
</tr>
<tr>
<td></td>
<td>jump register</td>
<td>jr $ra</td>
</tr>
<tr>
<td></td>
<td>jump and link</td>
<td>jal 2500</td>
</tr>
</tbody>
</table>
MIPS Pipelining
Recall Pipelining

- Split execution of each instruction into several, balanced “stages”
  - Ideally leads to 100% hardware utilization and ideal IPC = 1
- Each stage is a block of combinational logic
- Latency of each stage fits within 1 clock cycle
- Insert registers between each pipeline stage to hold intermediate results
- Execute each of these steps in parallel for a sequence of instructions
MIPS pipeline stages

• Fetch (F)
  – read next instruction from memory, increment address counter
  – assume 1 cycle to access memory
• Decode (D)
  – read register operands, resolve instruction in control signals, compute branch target
• Execute (E)
  – execute arithmetic/resolve branches
• Memory (M)
  – perform load/store accesses to memory, take branches
  – assume 1 cycle to access memory
• Write back (W)
  – write arithmetic results to register file
What do we need to add to actually split the datapath into stages?
MIPS pipelined datapath

But which value do we write back? (see next slide)
MIPS corrected datapath

• The problem with the previous implementation:
  – What happens when we writeback to the register file. What instruction supplies the write register value (destination register)?
  – Solution: We must forward (preserve) the destination register value.
END