Exam 2 is Live!

MIPS Load & Stores

| handout |
Today’s lecture

- MIPS Load & Stores
  - Data Memory
  - Load and Store Instructions
  - Encoding
  - How are they implemented?
State – the central concept of computing

What happens when the register file can be only so big?

Instructions and Control Logic
We need more space!

Registers

- Fast
  - High Bandwidth
  - Synchronous
- Small (32x32 bits)
- Expensive

Main Memory

- Slow
- Low Bandwidth
  - Not always synchronous
- Large ($2^{32}$B)
- Cheap-ish
Harvard Architecture stores programs and data in *separate* memories

**Instruction memory:**
- Contains instructions to execute
- Treat as read-only

**Data memory:**
- Contains the data of the program
- Can be read/written
Data Memory is byte-addressable with \(2^{32}\) bytes

<table>
<thead>
<tr>
<th>word_we</th>
<th>byte_we</th>
<th>Operation</th>
<th>(M[\text{ADDR}])</th>
<th>(32) bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Read (Load)</td>
<td>(\text{DATA}_\text{OUT} = M[\text{ADDR}])</td>
<td>(\text{DATA}_\text{OUT}) = (M[\text{ADDR}])</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>Write (Store) byte in ADDR</td>
<td>(M[\text{ADDR}] = \text{DATA}_\text{IN}[7:0])</td>
<td>(M[\text{ADDR}] = \text{DATA}_\text{IN}[7:0])</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Write (Store) word in ADDR</td>
<td>(M[\text{ADDR}]^\dagger = \text{DATA}_\text{IN}[31:0])</td>
<td>(M[\text{ADDR}]^\dagger = \text{DATA}_\text{IN}[31:0])</td>
</tr>
</tbody>
</table>

\(^\dagger\) (ADDR must be word aligned; e.g., ADDR[1:0] == 2’b00)
We can load or store bytes or words

<table>
<thead>
<tr>
<th></th>
<th>Load</th>
<th>Store</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Word</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lb</td>
<td>$R[rt] = \text{SEXT}(M[ADDR][7:0])$</td>
<td></td>
</tr>
<tr>
<td>lbu</td>
<td>$R[rt] = \text{ZEXT}(M[ADDR][7:0])$</td>
<td></td>
</tr>
<tr>
<td><strong>Byte</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sb</td>
<td>$M[ADDR] = R[rt][7:0]$</td>
<td>$M[ADDR] = R[rt][7:0]$</td>
</tr>
</tbody>
</table>
Indexed addressing derives `ADDR` from a base “pointer” register and a constant:

```
lw  rt, rs, imm
```

$6$

Destination

```
sw  rt, rs, imm
```

```
ADDR = R[rs] + imm
```
Load word (lw) is Little Endian

$12, 0($3)

Register File

$3  0x10010000

$12  0x33 22 11 00

Data Memory

0x10010000  0x00
0x10010001  0x11
0x10010002  0x22
0x10010003  0x33

...
sw $12, 0($3)

Register File

<table>
<thead>
<tr>
<th>...</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>$3</td>
<td>0x10010000</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>$12</td>
<td>0xAABBCCDD</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Data Memory

0x10010000: D0
0x10010001: CC
0x10010002: BB
0x10010003: AA

a) 0xAA
b) 0xBB
c) 0xCC
d) 0xDD
lb $12, 0($3)

Register File

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$3</td>
<td>0x10010000</td>
</tr>
<tr>
<td>$12</td>
<td>0xFF01FFF1</td>
</tr>
</tbody>
</table>

Data Memory

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x10010000</td>
<td>xF1</td>
</tr>
<tr>
<td>0x10010001</td>
<td>xF2</td>
</tr>
<tr>
<td>0x10010002</td>
<td>xF3</td>
</tr>
<tr>
<td>0x10010003</td>
<td>xF4</td>
</tr>
</tbody>
</table>
lbu $12, 1($3)

Register File

$3  0x10010000
...
...
$12
...
...

Data Memory

offset $3 + 1

0x10010000
0x10010001
0x10010002
0x10010003
0x10010004
0x10010005
0x10010006
0x10010007
0x10010008
...
...

Where is data read from?

a) ←
b) ←
c) ←
d) ←
e) ←
`lbu $12, 1($3)`

**Register File**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>$3</td>
<td>0x10010000</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>$12</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

**Data Memory**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0x10010000</td>
<td>xf1</td>
</tr>
<tr>
<td>0x10010001</td>
<td>xf2</td>
</tr>
<tr>
<td>0x10010002</td>
<td>xf3</td>
</tr>
<tr>
<td>0x10010003</td>
<td>xf4</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

What data is loaded into the register?

- a) 0x0000F200
- b) 0x000000F2
- c) 0xFFFFF200
- d) 0xFFFFFFFFF2
sb $12, 2 ($3)

Register File

Data Memory

What data is stored into memory?

a) xAA
b) xBB
c) xCC
d) xDD
Convert C code into MIPS assembly

```c
int a = 10;
int b = 0;
void main() {
    b = a + 7;
}
```
int a = 10;
int b = 0;
void main() {
    b = a + 7;
}

.data
a: .word 10
b: .word 0

.text
main:
    la $4, a

Convert C code into MIPS assembly
Which code finishes converting C code into MIPS assembly?

```c
int a = 10;
int b = 0;
void main() {
    b = a + 7;
}
```

```
.data
a: .word 10
b: .word 0
.text
main:
    $4 = 0x10010000
    la $4, a
```

A: `lw $5, 0($4) addi $5, $5, 7 sw $5, 0($4)`

B: `lw $5, 0($4) addi $5, $5, 7 sw $5, 1($4)`

C: `lw $5, 0($4) addi $5, $5, 7 sw $5, 2($4)`

D: `lw $5, 0($4) addi $5, $5, 7 sw $5, 4($4)`
int a = 10;
ext
int b = 0;

void main() {
   b = a+7;
}

.data
   a: .word 10
   b: .word 0
.text
   main:
   la $4, a
   lw $5, 0($4)
   addi $5, $5, 7
   sw $5, 4($4)
Loads and stores use the I-type format

<table>
<thead>
<tr>
<th>lw, lb, lbu</th>
<th>base</th>
<th>dest</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>rs</td>
<td>rt</td>
<td>address</td>
</tr>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sw, sb</th>
<th>base</th>
<th>source</th>
<th>offset</th>
</tr>
</thead>
</table>
load word implemented

\[ \text{Addr} = R[rs] + \text{sign_ext}(\text{ima}) \]
load byte unsigned implemented
\[ R[rt] = Z\ 2^{32}\cdot \overline{M[Addr][3:0]}^3 \]

\[ A0r = R[rs] + \text{signext}(\text{imm}) \]

Ignore bottom 2 bits.
store implemented
Full Machine Datapath – Lab 6