Assembly Language
Chapter 2

• MIPS
  – arithmetic instructions
  – control instructions
• Translating Instructions to binary
• MIPS
  – memory instructions
• Advanced data structures

Learning a new ISA
Learn the syntax, semantics of:

• __________ operations
• __________ operations
• __________ operations

MIPS Registers – 32 registers

<table>
<thead>
<tr>
<th>Name</th>
<th>Reg Number</th>
<th>Usage</th>
<th>Preserved across call?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Szero</td>
<td>0</td>
<td>The constant 0</td>
<td>Yes</td>
</tr>
<tr>
<td>S1-S$V1</td>
<td>2-3</td>
<td>Function results</td>
<td>No</td>
</tr>
<tr>
<td>S$0-S$3</td>
<td>4-7</td>
<td>Arguments</td>
<td>No</td>
</tr>
<tr>
<td>S$0-S$7</td>
<td>8-15</td>
<td>Temporaries</td>
<td>No</td>
</tr>
<tr>
<td>S$0-S$9</td>
<td>16-23</td>
<td>Saved</td>
<td>Yes</td>
</tr>
<tr>
<td>S$10-S$19</td>
<td>24-25</td>
<td>More temporaries</td>
<td>No</td>
</tr>
<tr>
<td>S$18-S$27</td>
<td>28</td>
<td>Global pointer</td>
<td>Yes</td>
</tr>
<tr>
<td>S$18-S$27</td>
<td>29</td>
<td>Stack pointer</td>
<td>Yes</td>
</tr>
<tr>
<td>S$18-S$27</td>
<td>30</td>
<td>Frame pointer</td>
<td>Yes</td>
</tr>
<tr>
<td>S$0-S$8</td>
<td>31</td>
<td>Return address</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Saved vs Temporary regs

• Saved:
  – Register __________ value after function call
  – Easier to program
• Temporary
  – Register __________ value after function call
  – More efficient

Design Principle 1
Simplicity Favors Regularity

Arithmetic “R-Format”

• Two input registers
• One output register

<table>
<thead>
<tr>
<th>Operation</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
<th># meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>add $2,$3,$5</td>
<td>1</td>
<td>5</td>
<td>2</td>
<td>0</td>
<td>32</td>
<td>$2 \leftarrow $3 + $5</td>
</tr>
<tr>
<td>sub $2,$3,$5</td>
<td>1</td>
<td>5</td>
<td>2</td>
<td>0</td>
<td>34</td>
<td>$2 \leftarrow $3 - $5</td>
</tr>
<tr>
<td>addu $2,$3,$5</td>
<td>1</td>
<td>5</td>
<td>2</td>
<td>0</td>
<td>33</td>
<td>$2 \leftarrow $3 + $5</td>
</tr>
</tbody>
</table>
| slt $2, $3, $5| 1  | 5  | 2  | 0     | 42    | $2 \leftarrow 1 \
|            |    |    |    |       |       | if ($3 < $5) $2 \leftarrow $2 |
Arithmetic “I-format”

- One input register
- One hard-coded constant
- One output register

<table>
<thead>
<tr>
<th>Operation</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
<th># comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>addi</td>
<td>2</td>
<td>2</td>
<td>8</td>
<td></td>
<td></td>
<td>$2 &lt;- $3 + 8</td>
</tr>
<tr>
<td>andi</td>
<td>1</td>
<td>2</td>
<td>10</td>
<td></td>
<td></td>
<td>$2 &lt;- $3 &amp; 10</td>
</tr>
<tr>
<td>slti</td>
<td>1</td>
<td>2</td>
<td>7</td>
<td></td>
<td></td>
<td>if ($3 &lt; 7) $2 &lt;- 1 else $2 &lt;- 0</td>
</tr>
</tbody>
</table>

Design Principle 2: Small is faster

- Variables must be loaded into registers before use
- How many registers should ISA provide?
- More:
  - Fewer:
    - How many bits necessary for each operand?
      - 32 regs: _____ 64 regs: _____ N regs: _____

Conditional & Unconditional Branches

- goto loop
- if (i < 100) goto loop

<table>
<thead>
<tr>
<th>Operation</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
<th># comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>beq</td>
<td>3</td>
<td>2</td>
<td>8</td>
<td></td>
<td></td>
<td>if ($3 == $2) goto loop</td>
</tr>
<tr>
<td>bne</td>
<td>3</td>
<td>2</td>
<td>8</td>
<td></td>
<td></td>
<td>if ($3 != $2) goto loop</td>
</tr>
<tr>
<td>jr</td>
<td>3</td>
<td>1</td>
<td>8</td>
<td></td>
<td></td>
<td>goto $3</td>
</tr>
<tr>
<td>j</td>
<td>loop</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>goto loop</td>
</tr>
</tbody>
</table>

MIPS Example 1

if (x == y)
    equal = 1;
else
    equal = 0;

assume x is in $s0, y is in $s1, and equal is in $v0.

MIPS Example 2

sum = 0;
for(i=0; i < 100; i++)
    sum += i;

translated into more detailed C (with gotos)
    sum = 0;
    i = 0;
loop: if (       ) goto _____;
    sum += i;
    goto _____;
end:

MIPS Example 2 In Assembly

sum=0;
for(i=0; i < 100; i++)
    sum += i;
loop: if (       ) goto _____;
    sum += i;
    goto _____;
end:
Translating into machine code

<table>
<thead>
<tr>
<th>Assembly instr</th>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>shamt</th>
<th>funct</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>add $t0, $0, 0</td>
<td>add</td>
<td>t0</td>
<td></td>
<td></td>
<td></td>
<td>i = 0</td>
</tr>
<tr>
<td>add $t0, $0, 100</td>
<td>add</td>
<td>t0</td>
<td></td>
<td></td>
<td></td>
<td># t0 &lt;= (i &lt; 100 ?)</td>
</tr>
<tr>
<td>beq $t0, $0, end</td>
<td>beq</td>
<td>t0</td>
<td></td>
<td></td>
<td></td>
<td># if (i &gt;= 100) goto end</td>
</tr>
<tr>
<td>la $t0, A</td>
<td>la</td>
<td>t0</td>
<td></td>
<td></td>
<td></td>
<td># t0 &lt;= (i &lt; 100 ?)</td>
</tr>
<tr>
<td>sw $t2, 0($t1)</td>
<td>sw</td>
<td>t2</td>
<td></td>
<td></td>
<td></td>
<td># t0 &lt;= M[A+i] = A[i]</td>
</tr>
<tr>
<td>add $s0, $0, 1</td>
<td>add</td>
<td>s0</td>
<td></td>
<td></td>
<td></td>
<td># i++</td>
</tr>
<tr>
<td>jump loop</td>
<td>j</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td># goto loop</td>
</tr>
</tbody>
</table>

Why memory operations?
Problem:
- (collection of regs) must be small for ____________
- Register number is _______________ into instruction
- Register number not always known at compile-time (______________) 

Why memory operations?
Solution:
- Build a ___________ that will store memory that does not fit in ____________.
- Allow registers to hold the _______________ that data resides.
- Many ________________ are supported by different ISA’s.

Registers vs Memory
- Registers are _______________ memory
- Everything residing in a register has a real home ________________
- Variables live in memory and are brought into registers ________________
- When that register is needed for something else, the item ________________ ________________

Memory Setup in C/Java
- int X;
- What does this do? What does the memory look like?

Load/Store Instructions
- Displacement addressing mode
- Register indirect is Displacement with 0 offset
- lw = load word (4 bytes), lb = load byte (1 byte)

<table>
<thead>
<tr>
<th>Operation</th>
<th>rs</th>
<th>rt</th>
<th>shamt</th>
<th>funct</th>
<th># comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw $2, 32($3)</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td># $2 &lt;= M[32 +$3]</td>
</tr>
<tr>
<td>sw $2, 16($3)</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td># M[16 +$3] &lt;= $2</td>
</tr>
<tr>
<td>lb $2, 8($3)</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td># $2 &lt;= M[0 +$3]</td>
</tr>
<tr>
<td>sb $2, 3($3)</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td># M[3 +$3] &lt;= $2</td>
</tr>
</tbody>
</table>
### Possible Memory Addressing Modes (MIPS supports few)

<table>
<thead>
<tr>
<th>Mode</th>
<th>Example</th>
<th>Meaning</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>Displacement</td>
<td>( \text{Lw} , R4, 100(R1) )</td>
<td>( A = M[R1] + 100 )</td>
<td>Array with constant offset</td>
</tr>
<tr>
<td>Register indirect</td>
<td>( \text{Lw} , R4, (R1) )</td>
<td>( A = M[R1] )</td>
<td>You have pointer</td>
</tr>
<tr>
<td>Indexed</td>
<td>( \text{Lw} , R4, (R1+R2); )</td>
<td>( A = M[R1] + R2 )</td>
<td>Indexing arrays of large elements</td>
</tr>
<tr>
<td>Direct or absolute</td>
<td>( \text{Lw} , R4, (100) )</td>
<td>( A = M[100] )</td>
<td>Global, static data</td>
</tr>
<tr>
<td>Memory indirect</td>
<td>( \text{Lw} , (R1+R2) )</td>
<td>Double-pointer</td>
<td></td>
</tr>
<tr>
<td>Autodecrement</td>
<td>( \text{Lw} , R1, (R2) - )</td>
<td>( A = M[R1] - R2 )</td>
<td>Stepping through array in loop</td>
</tr>
<tr>
<td>Autoincrement</td>
<td>( \text{Lw} , R1, (R2) - )</td>
<td>( A = M[R1] + R2 )</td>
<td>Stepping through array in loop</td>
</tr>
<tr>
<td>Scaled</td>
<td>( \text{Lw} , R4, 100(R1); )</td>
<td>( A = M[100 + R2 + \text{offset}] )</td>
<td>Indexing arrays of large elements</td>
</tr>
</tbody>
</table>

### Variable Types
- **Local variables**
- **Global variables declared and allocated in**
- **Heap variables created with**

### Declaring, Allocating & Initializing Global Variables

**C:**
```c
int GlobalA = 3;
int main(int argc, char *argv[]) {
    ...
}
```

**Java:**
```java
public class MyClass {
    public static int GlobalA = 3;
}
```

**MIPS:**
```
.data
GlobalA:    .word 0x03
.text
main: ...
```

### Declaring, Allocating & Initializing Local Variables

**C:**
```c
int main(int argc, char *argv[]) {
    int *LocalA = (int*)malloc(4);
    *LocalA = 5;
    ...
}
```

**Java:**
```java
public class MyClass {
    public static int main(String[] argv) {
        int LocalA = 5;
        ...
    }
}
```

**MIPS:**
```
add $sp, $sp, -(24 + x + 4)
addi $a0, $0, 4
jal malloc
sw $v0, 0($sp)
```

### Declaring, Allocating & Initializing Heap Variables

**C:**
```c
int main(int argc, char *argv[]) {
    int *LocalA = (int*)malloc(4);
    *LocalA = 5;
    ...
}
```

**Java:**
```java
int main(int argc, char *argv[]) {
    int LocalA = 5;
    ...
}
```

**MIPS:**
```
add $sp, $sp, -(24 + x + 4)
add $v0, $sp, -(24 + x + 4)
sw $v0, 0($sp)
```

### How do I get &X (address of X)?

- **Global (Static) Vars**
- **Local (Stack) Vars**
- **Dynamic (Heap) Vars**

**C:**
```c
int GlobalA;
int main(int argc, char *argv[]) {
    int StackA;
    int *HeapA = new int;
    ...
}
```
MIPS Memory Notes

- `la $s0, variable` is a pseudoinstruction – assembler replaces it with
  - __________
  - __________
- Register __________ - not enough registers, must save a value in memory
- Alignment – Integers must have addresses that are evenly divisible by __________

MIPS Example 3

Assumptions:
- A is a global var
- B is a local var at 16+$sp

A = B + A;

B = 2 * A;

Memory Setup in C/Java

- C++: int *intarray = new int[10];
- Java: int[] intarray = new int[10];
- What does this do? What does the memory look like?
- Where is intarray[5] located?
- Where is intarray[i] located?

Declaring, Allocating & Initializing Local Arrays

```c
int main(int argc, char *argv[])
{
    int LocalA = 5;
    int LocalB[] = {1,2,3};
}
```

Declaring, Allocating & Initializing Global Arrays in HLL

```c
public class MyClass{
    public static int GlobalA = 3;
    public static int GlobalB[] = {0x20040002, 0x20080001, 0x8b502a, 0x10840000, 0x80fffa, 0x10010200, 0x00000000};
    public static int GlobalFFT[] = {0x20040003, 0x20840000, 0x80fffa, 0x10010200, 0x00000000};
}
```

Declaring, Allocating & Initializing Global Arrays in MIPS

```plaintext
.data
GlobalA:
.word 0x03;
GlobalB:
.word 0x20040002 0x20080001 0x8b502a 0x10840000 0x80fffa
.word 0x10010200 0x00000000
.text
main:
```

```
```
```
Declaring, Allocating & Initializing Heap Arrays in HLL

```c
int main(int argc, char *argv[]) {
    int *LocalA = (int *)malloc(4);
    LocalA = 5;
    LocalA[0] = 1;
    LocalA[1] = 2;
    LocalA[2] = 3;
}
```

```
public class MyClass {
    public static void main(int argc, String argv) {
        int LocalB[] = new int[3];
        LocalB[0] = 1;
        LocalB[1] = 2;
        LocalB[2] = 3;
    }
}
```

MIPS Example 5

```
Translate from C code
int A[100]; // ints are 4 bytes in Java/C
char B[100]; // chars are 1 byte in C
void main() {
    char c = B[50];
}
```

Assumptions:
- A&B are global,
- c is in the stack,
- 6 bytes from $sp

MIPS Example 6

```
Translate
int A[100]:
int i;
...
x = A[i];
```

Assumptions:
- &A[0] is in $s0,
- x is in $t0,
- i is in $s1

Objects
- Members variables are _______________
  from the beginning of the object
- Member functions have hidden _____________

Linked List
```
class Link {
    public: (since we don’t know how to do method calls yet)
        Link *next;
        void *data;
    public:
    Link() {next=NULL; data=NULL;}
    inline void SetData(void *d) {data = d;}
    inline void *GetData() {return data;}
    inline void SetNext(Link *n) {next = n;}
    inline Link *GetNext() {return next;}
};
```

If “this” pointer (address of current Link) is stored in $a0, what is the memory location of the current object’s “data” variable?
Assembly code for Link

For all methods, assume "this" pointer is in $a0, first argument is in $a1
Place the return value in $v0.

SetData
  // this.data = d
  Which is the source?
  Which is the destination?
  Is this.data in memory or in a register?
  Is this.data in a register or in memory?

GetData
  return this.data
  Which is the source?
  Which is the destination?
  Is this.data in memory or a register?
  Is this.data in a register or in memory?

Linked Lists

• The memory is not ______________ — it is scattered about.
• Allocated dynamically, so we must call __________
• allocator gives you the address of the beginning of the allocated memory in ____

Linked List

class LinkedList {  
  private:
  Link *head;
  Link *tail;
  public:
  LinkedList();
  ~LinkedList();
  void InsertHead(void *data);
  void InsertTail(void *data);
  void *RemoveHead();
};

C++ syntax

Code for

void InsertTail(void *data)
  "this" Linked List is in $a0, data is in $a1

Java syntax

Voluntary Assignment

• Create a snippet of Java, C, or C++ code that you are unsure how to translate into mips.
• E-mail it to me before class Thursday
• I will choose from among them to do more examples