MIPS assembly programming:

Handouts in back!

Today's lecture

- Exam 3 Structure
- Review the Datapath
  - Trace a couple of instructions
- Assembly programming
  - Register names
  - How is it implemented?
- Branches
  - Loops
  - If/then/else
  - How implemented?

Exam 3 Structure

- OH WOW! In 6 weeks you have learned how to build a computer!

- For exam 3, you will add components and control signals to the datapath to implement a new instruction for MIPS
  - Create an instruction that creates world peace

What you need for exams 4 & 5

- You must become “fluent” in MIPS assembly:
  - Translate from C to MIPS and MIPS to C

- Example problem from a 233 mid-term:

Question 3: Write a recursive function (30 points)

Here is a function pow that takes two arguments (n and m, both 32-bit numbers) and returns n^m (i.e., n raised to the m^{th} power).

```
int
pow(int n, int m) {
   if (m == 1)
      return n;
   return n * pow(n, m-1);
}
```

Translate this into a MIPS assembly language function.

We give MIPS registers meaningful names to help when writing software

- In hardware, all the registers are equivalent:
  - Except register $0, which is always zero

$zero

- For temporary values, we'll use the $t registers

$t0-$t9

- If you have no reason for picking another register, then you should probably be using a $t register.

Replace register numbers with names

$\$t0 = (\$t1 + \$t2) \times (\$t3 - \$t4)$

$\$\cancel{8} = (\$\cancel{9} + \$\cancel{10}) \times (\$\cancel{11} - \$\cancel{12})$

```
add  $t0, $t1, $t2 # $t0 contains $t1 + $t2
sub  $t5, $t3, $t4 # Temporary value $t5 = $t3 - $t4
mul  $t0, $t0, $t5 # $t0 contains the final product
```

How do we perform calculations on data in main memory?

```
char A[4] = {1, 2, 3, 4};
int result;
void main(){
   result = A[0] + A[1] + A[2] + A[3];
}
```

Computing on data in main memory generally requires load->compute->store

- Steps
  1. Load the data from memory into the register file.
  2. Do the computation, leaving the result in a register.
  3. Store that value back to memory if needed.

Global data is allocated in the .data segment

- Allocated to memory addresses at compile time.
- Amount of space allocated is based on variable type.

```
.data    // indicates the beginning of data segment
.word    // allocates space for 4-byte variable
.byte    // allocates space for 1-byte variable
.asciiz  // allocates space for an ASCII string
.space   // allocates a defined amount of space.
```

Use either byte or word operations based on datatype

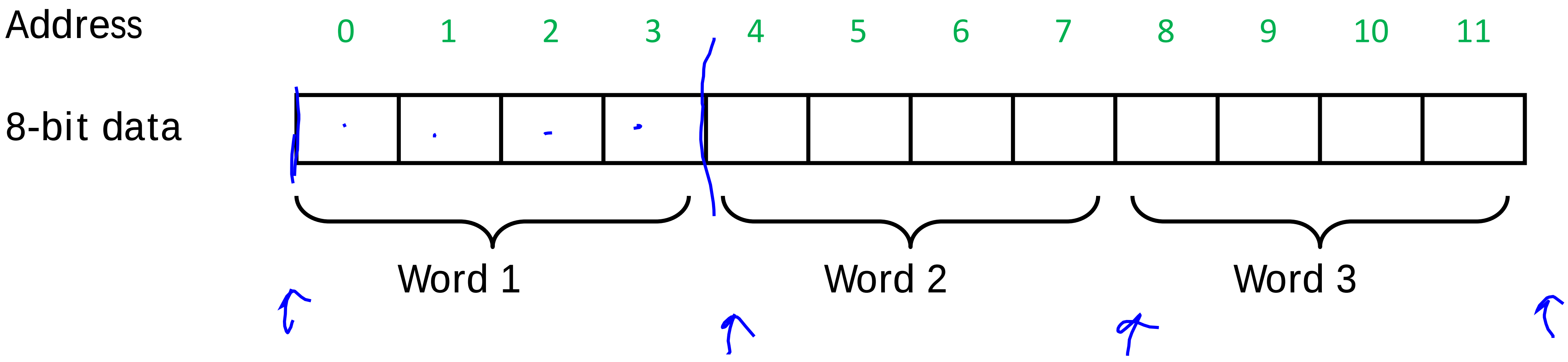
`lb` and `sb`

- Transfer 1 byte of data between regs and mem
- Datatypes: char
- Note: Use least significant bits from registers

`lw` and `sw`

- Transfer 1 word (4 bytes) of data between regs and mem
- Datatypes: integers, float, addresses/pointers
- Note: must be word-aligned

Word alignment: 32-bit words must start at an address that is divisible by 4.



- Unaligned memory accesses result in a bus error, which you may have unfortunately seen before.

An array of words

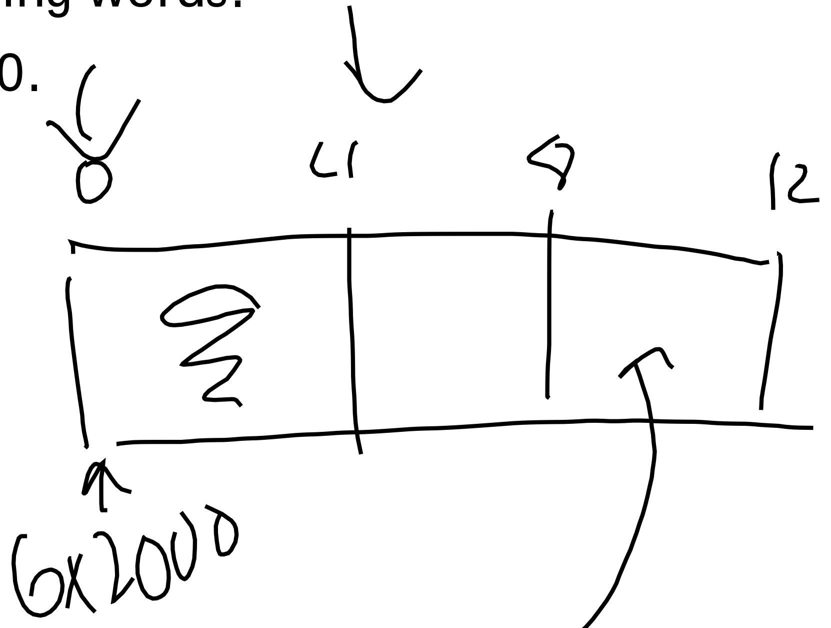
- Remember to be careful with memory addresses when accessing words.
- For instance, assume an array of words begins at address 2000.
  - The first array element is at address 2000.
  - The second word is at address *2004*, not 2001.
- Revisiting the earlier example, if $a0 contains 2000, then

```
lw $t0, 0($a0)
```

accesses the 0th word of the array, but

```
lw $t0, 8($a0)
```

would access the *2nd* word of the array, at address 2008.



Pseudo-instructions give programmers useful instructions that are not part of the MIPS architecture

Pseudo instructions		Real instructions
`li    $a0, 20# Load immediate 20 into $a0`	Assemble into	`addi   $a0, $0, 20`
`move  $a1, $t0        # Copy $t0 into $a1`		`add    $a1, $t0, $0`

- A complete list of instructions is given in [Appendix A] of the text.

Coding Example

```
char A[4] = {1,2,3,4};
int result;

void main(){
   result = A[0] + A[1] + A[2] + A[3];
}
```

Assemblers provide 4 pseudo-branches to make our lives easier

```
blt  $t0, $t1, L1  # Branch if $t0 < $t1
ble  $t0, $t1, L2  # Branch if $t0 <= $t1
bgt  $t0, $t1, L3  # Branch if $t0 > $t1
bge  $t0, $t1, L4  # Branch if $t0 >= $t1
```

There are also immediate versions of these branches, where the second source is a constant instead of a register.

Pseudo-branches assemble down to `slt` and either `beq` or `bne`

```
blt $a0, $a1, Label
```

Assembles into

```
slt  $at, $a0, $a1  # $at = 1 if $a0 < $a1
bne  $at, $0, Label # Branch if $at != 0
```

$at is the “assembler temporary” register ($1)

if-then-else statements require branches and jumps

- If there is an else clause, it is the target of the conditional branch
- And the then clause needs a jump over the else clause

```
if (v0 < 0)
    v0 --;
else
    v0 ++;
v1 = v0;
```

```
    bge  $v0, $0, E
    subi $v0, $v0, 1
    j    L
E:  addi $v0, $v0, 1
L:  move $v1, $v0
```