9/29: Lecture Topics

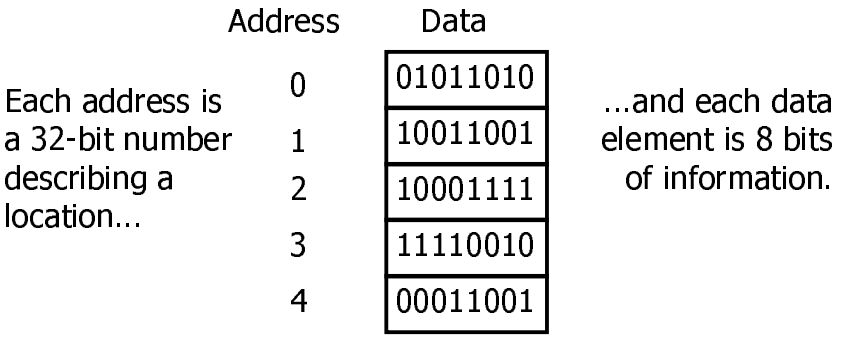
- Memory
  - Addressing (naming)
  - Address space sizing
- Data transfer instructions
  - load/store
    - on arrays
    - on arrays with variable indices
- Conditional branch instructions

Administrative notes

- Don't forget: homework due Monday
- Readings: before or after lecture?
- Lecture slides on the web/printing
- Are you on the mailing list yet?

Memory and Addressing

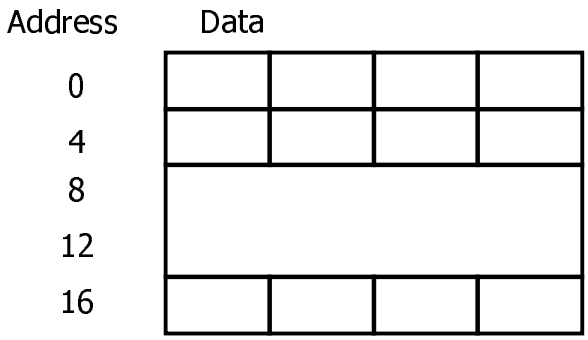
- Only so many registers, but memory holds millions of data elements
- Think of memory as a big array



MIPS Information Units

- Data types and size:
  - byte (eight bits)
  - half-word (two bytes)
  - word (four bytes)
  - float (four bytes, single precision)
  - double (eight bytes, double precision)
- Memory is byte addressable
- Data types start at an address divisible by the data type size *(alignment)*

Word Alignment



- Why do you think so many architectures use alignment?

How Big Can Memory Be?

- How big is an address?
  - 32 bits
- How many addresses are there?
  - 2^{32} = about 4 billion
- How much data lives at each address?
  - 1 byte
- How much memory can this computer have?
  - 4 billion x 1 byte = 4 GB

Some Perspective

- Today's attitude:
  - 32 bits is definitely not enough for long
  - We hope 64 bits will be enough for a while
- The attitude a while back:
  - 16 bits is enough for all time!
  - Who could possibly use up 64K of memory?

Loads and Stores

- Data transfer instructions let us
  - *load* data from memory to registers
  - *store* data from registers into memory
- In MIPS, all data must be loaded before use and stored after use
  - *Load-store architecture*
- `lw` instruction fetches a word
- `sw` instruction stores a word

Alternatives to Load-Store (p. 191)

- *register-register* is another name for load-store (MIPS)
- *accumulator* architectures had only one register (EDSAC)
- *register-memory* architectures allow one operand to be in memory (IBM 360, Intel 80386)
- *memory-memory* architectures allow both operands to be in memory (DEC VAX)

Load Syntax

- first argument: where to put the data
- second argument: offset
- third argument: address

Desired result:	MIPS assembly:
Load the word at the address in $s0 into register $t0	`lw $t0, 0($s0)`
Load the word at the address in $s0 plus 4 into $t0	`lw $t0, 4($s0)`

A Related Concept: `move`

- `lw` and `sw` are for moving data between memory and registers
- `move` is for copying data from one register to another
- common usage: zero a register

```
move $t0, $0
```

"Complex" Example Revisited

Desired result:

f = (g + h) - (i + j)

f, g, h, i, j are in registers $s0, $s1, $s2, $s3, $s4, respectively

MIPS assembly:

```
add $s0, $s1, $s2
sub $s0, $s0, $s3
sub $s0, $s0, $s4
```

"Complex" Example with L/S

Desired result:

f = (g + h) - (i + j)

load g, h, i, j, compute, then store f

MIPS assembly:

```
add $s0, $s1, $s2
sub $s0, $s0, $s3
sub $s0, $s0, $s4
```

Address	Data
200	f
204	g
208	h
212	i
216	j

Arrays in Assembly Programs

- One reason for `lw` syntax is arrays
  - keep the base address in a register
  - `lw` and `sw` on offsets into the array
- Example:

```
int A[10];      # addr of A in $t0
A[0] = 0;       sw $0, 0($t0)
A[1] = 0;       sw $0, 4($t0)
```

Variable Array Indexing

- Very common use of arrays: walk through the array by incrementing a variable
- `lw` and `sw` instructions allow offset to be specified:
  - as a constant
  - as the contents of a register

```
lw $t0, 4($t2)     vs.   lw $t0, $t1($t2)
```

Array Example

Idea: translate

```
g = h + A[i]
```

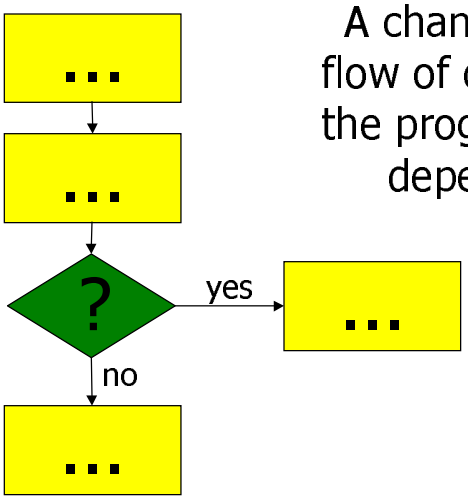
into assembly, assuming

```
Base address of A is in $s3
g, h, i in $s1, $s2, $s4
```

Operation Types

- Remember these?
- ✓Arithmetic
  - add, sub, mult, div
- ✓Data Transfer
  - lw, sw, lb, sb
- Conditional Branch
  - beq, bne
- Unconditional Jump
  - j, jr, jal

Conditional Branch



A change of the flow of control of the program that depends on a condition

Control flow in C

- Conditional branch constructs:
  - while, do while loops
  - for loops
- Unconditional jump constructs:
  - goto
  - switch statements

Branching Instructions

- `beq`
- `bne`
- other real instructions: `b`, `bgez`, `bgtz`, more...
- other pseudoinstructions: `beqz`, `bge`, `bgt`, `ble`, `blt`, more...

Pseudoinstructions

- One-to-one mapping between assembly and machine language not strictly true
- Assembler can do some of the work for you
- Example: `bgt` is a real instruction; `blt` is a pseudoinstruction

- `move` is a pseudoinstruction

Labels in MIPS

- MIPS allows text tags
  - a name for a place to branch to

```
loop:  add $t0, $t0, $t0    #
       bne $t0, $t1, loop   #
       sw  $t2, 0($t3)      #
```

- Under the covers, the assembler converts the text tag into an address

Building a while loop in MIPS

Idea: translate

```
i=0;
while(i<100) {
  i++;
}
```

into assembly.

How about a for loop?

- One answer: translate from for to while, then use while loop conventions

```
for(i=0; i<N; i++) {
  do_stuff(i);
}
```

```
i=0;
while(i<N) {
  do_stuff(i);
  i++;
}
```

- This is typically how compilers handle loops

Example C Program

```
int array[100];

void main() {
    int i;
    for(i=0; i<100; i++)
        array[i] = i;
}
```

An Assembly Version

```
main:   move $t0, $0        #
        la   $t1, array     #
start:  bge  $t0, 100, exit #
        sw   $t0, 0($t1)    #
        addi $t0, $t0, 1    #
        addi $t1, $t1, 4    #
        j start             #
exit:   j $ra               #
```