Operands in Computer HW

Andrea Janes

Content

- **Review: Instruction types**
- Types of operands
- Managing operands in memory

Instructions: Groups

- Arithmetic operations
  - Addition, Subtraction,...
- Information flow
  - Load from memory
  - Store in memory
- Logic operations
  - Logic and / or
  - Negation
  - Shift
- Branch operation

Instructions: Types

- **Instructions with different numbers of operands**
  - **1 Operand:**
    - **Jump # address**
    - **Jump $ register number**
  - **2 Operands:**
    - **Multiply $2, $3**
    - **Multiply $2 and $3 and store the result in ($2,$3)**
  - **3 Operands:**
    - **Add a, b, c   # a = b + c**
    - **Add a, a, b   # a = a + b**
    - **Sub a, b ,c   # a = b - c**

Example (already seen)

- **Add the four variables b, c, d, and e and place the result in variable a.**

  a = b + c + d + e
- **Add operation, first parameter is the destination, second + third are the source**
- **Sequence:**
  - **add a, b, c** **# a = b + c**
  - **add a, a, d** **# a = b + c + d**
  - **add a, a, e** **# a = b + c + d + e**

Content

- *Review: Instruction types*
- **Types of operands**
- Managing operands in memory

Operands

- Variables
  - Variables are mapped to registers
  - Registers can be accessed directly from the program
- Registers
  - Fixed, identical length (typical: 32 bits)
  - Fixed number (typical 32)
- MIPS
  - 32 registers with 32 bits
  - 30 usable, 2 with special meaning

Operands in the MIPS assembly

- Registers
- Immediate
- Memory

Example of allocation of MIPS registers

GNU C register allocation

$zero	0	constant 0
$at	1	assembler
$v0 ... $v1	2-3	result value registers
$a0 ... $a3	4-7	arguments
$t0 ... $t7	8-15	temporary variables
$s0 ... $s7	16-23	saved
$t8 ... $t9	24-25	temporary variables
$k0 ... $k1	26-27	operating system
$gp	28	global pointer
$sp	29	stack pointer
$fp	30	frame pointer
$ra	31	return address

Immediates

- Already seen
- 16 bits in MIPS

Memory

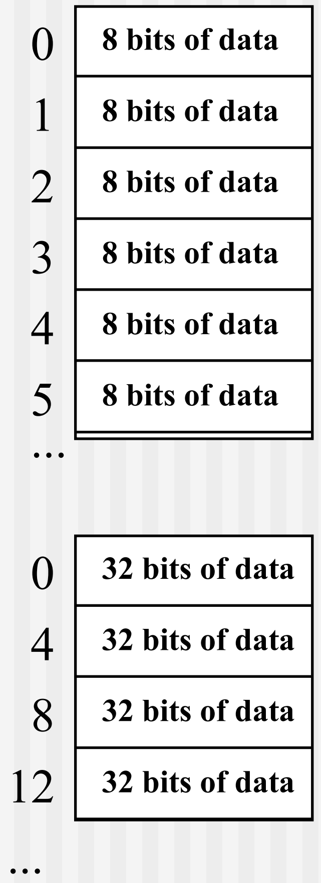
- Memory is used as an indexable array
- Notice the equivalence
  - RAM vs. Array
  - Disk vs. (Linked) lists

Content

- *Review: Instruction types*
- *Types of operands*
- **Managing operands in memory**

Data transfer: memory

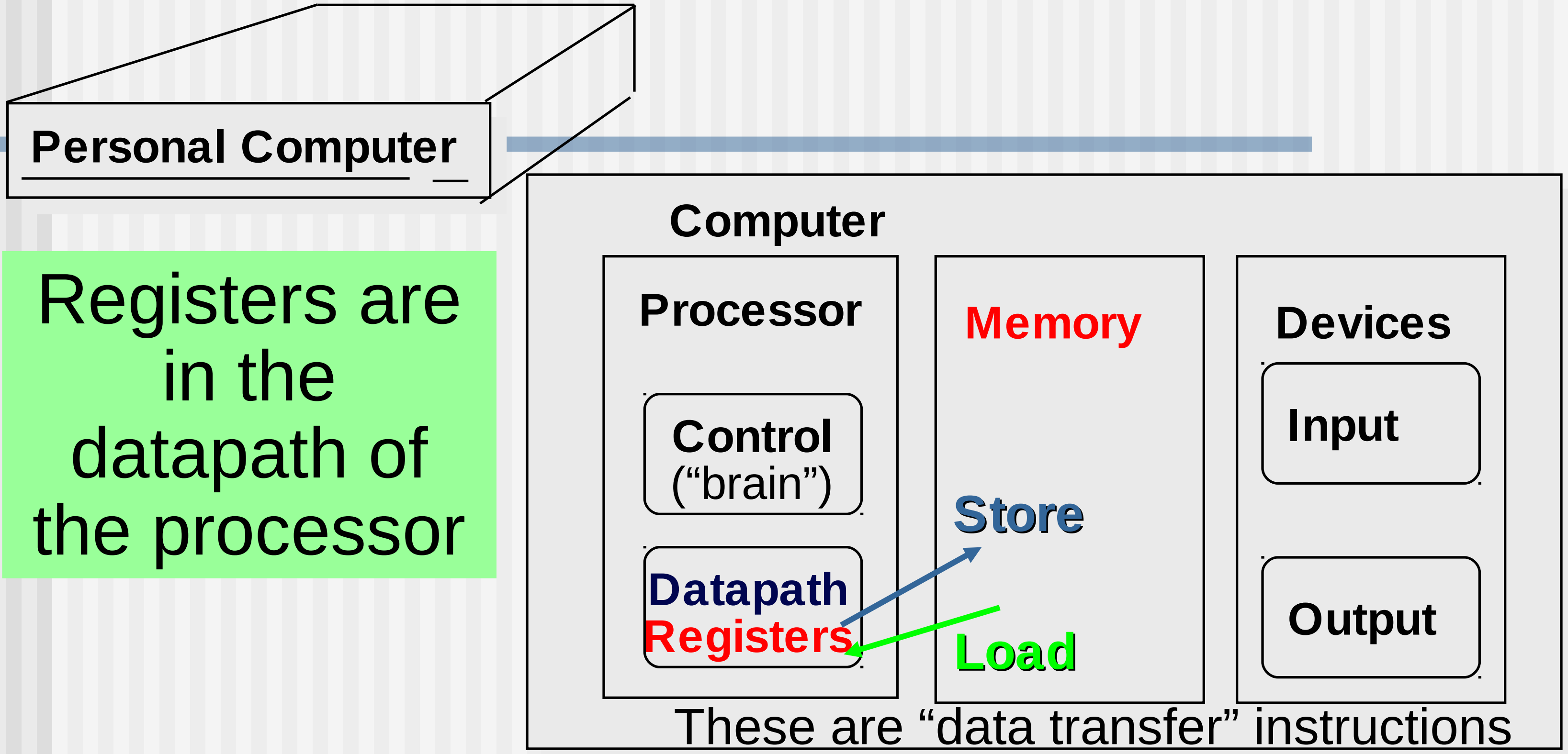
- Viewed as a large, single-dimension array, with an address.
- A memory address is an index into the array
- "Byte addressing" means that the index points to a byte of memory.
- „Word addressing" means that the index points to a word (MIPS:32 bit) of memory.



Data transfer: memory

- Get data: "load word:" moving data from memory to register
  - lw $s0, 8($s1)
- Saving data: "store word:" moving data from register to memory
  - sw $s0, 16($s1)
- Only this instruction type accesses memory: load/store architecture

Anatomy Again



Registers are in the datapath of the processor

If operands are in memory, we must:

1. transfer them to the processor to operate on them,
2. and then transfer back to memory when done

Data Transfer: Mem2Reg (1/4)

- To transfer a word of data, we need to specify two things:
  - Register: specify this by number (0 - 31)
  - Memory address: more difficult
    - Think of memory as a single one-dimensional array, so we can address it simply by supplying a pointer to a memory address.
    - Other times, we want to be able to offset from this pointer.

Data Transfer: Mem2Reg (2/4)

- To specify a memory address to copy from, specify two things:
  - A register which contains a pointer to memory
  - A numerical offset (in **bytes**)
- The desired memory address is the sum of these two values.
- Example: **8($t0)**
  - specifies the byte memory address pointed to by the value in $t0, plus 8 bytes

Data Transfer: Mem2Reg (3/4)

- Load Instruction Syntax:

  1 2,3(4)

  where

  1) operation (instruction) name
  2) register that will receive value
  3) numerical offset in bytes
  4) register containing pointer to memory

- Instruction Name:
  - **lw $t0,8($s0)**
  - (lw = Load Word, so load 32 bits or one word from memory at byte address $s0 + 8)

Data Transfer: Mem2Reg (4/4)

- Example: **lw $t0,12($s0)**
  - This instruction will take the pointer in $s0, add 12 bytes to it, and then load the value from the memory pointed to by this calculated sum into register $t0
- Notes:
  - $s0 is called the base register
  - 12 is called the offset
  - offset is generally used in accessing elements of array or structure: base reg points to beginning of array or structure

Data Transfer: Reg2Mem (1/2)

- Also want to store value from a register into memory
- Store instruction syntax is identical to Load instruction syntax
- Instruction Name:
  - **sw $t0,8($s0)**
  - (sw means Store Word, so 32 bits or one word are stored to memory at byte address $s0 + 8)

Data Transfer: Reg2Mem (2/2)

- Example: sw $t0,12($s0)

  This instruction will take the pointer in $s0, add 12 bytes to it, and then store the value from register $t0 into the memory address pointed to by the calculated sum

Role of Registers vs. Memory

- What if more variables than registers?
  - Compiler tries to keep most frequently used variable in registers
  - Writing less common to memory: spilling
  - Where? On stack
- Why not keep all variables in memory?
  - Smaller is faster: registers are faster than memory
  - Registers more versatile:
    - MIPS arithmetic instructions can read 2, operate on them, and write 1 per instruction
    - MIPS data transfer only read or write 1 operand per instruction, and no operation

Reference vs. Values

- Key Concept: A register can hold any 32-bit value. That value can be a (signed) int, an unsigned int, a reference (memory address, aka pointer), etc.
- If you write lw $t2,0($t0)
  then $t0 better contain a pointer ☺
- What if you write add $t2,$t1,$t0 then what $t0 and $t1 must contain?

Freie Universität Bozen
Libera Università di Bolzano
Free University of Bozen · Bolzano

Addressing: Byte vs. Word

- Every word in memory has an **address**, similar to an index in an array
- Early computers numbered words like Java numbers elements of an array:
  - `Memory[0], Memory[1], Memory[2], …`

Called the "**address**" of a word

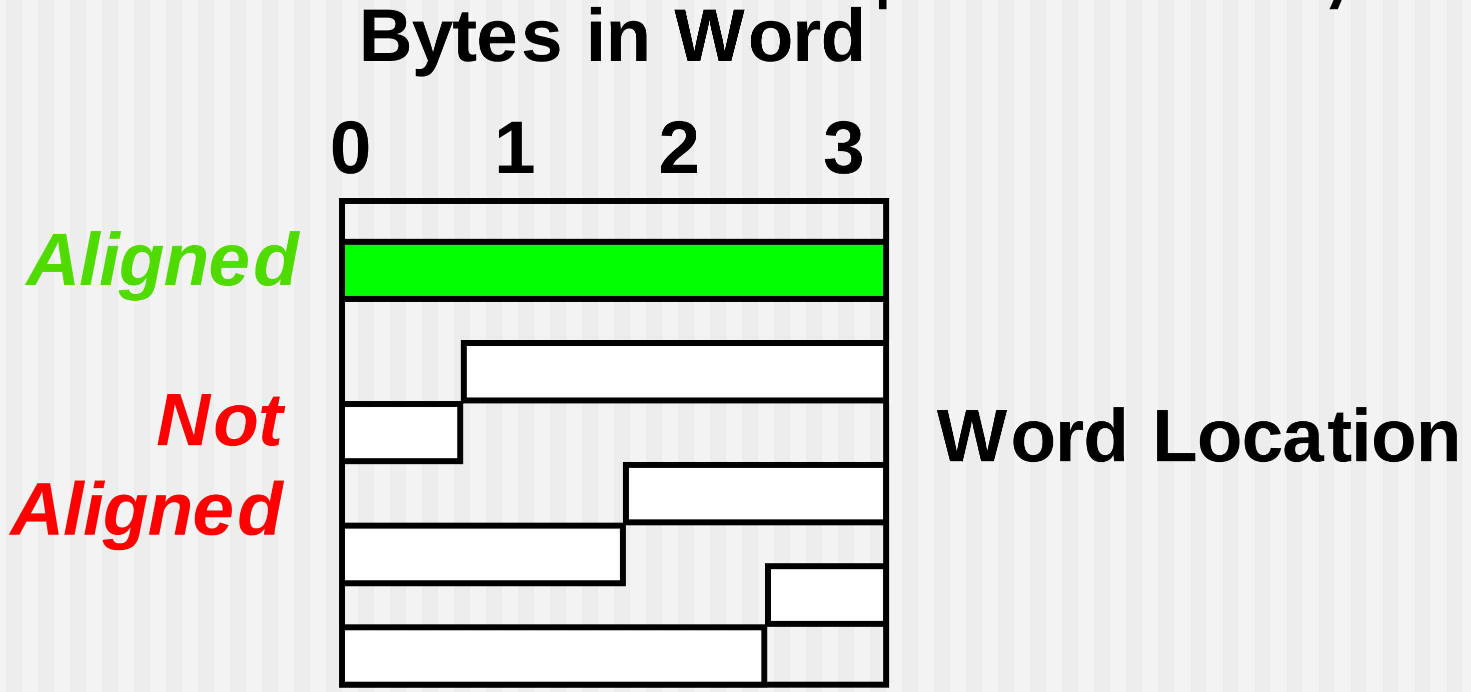
Byte vs. Word

- Computers needed to access (usually) 8-bit **bytes** as well as **words** (usually 4 bytes/word)
  - For strings; byte data transfers later
- Today machines address memory as bytes, hence word addresses differ by 4
  - `Memory[0], Memory[4], Memory[8], …`

Freie Universität Bozen
Libera Università di Bolzano
Free University of Bozen · Bolzano

Notes about Memory Alignment (if time)

- MIPS requires that all words start at addresses that are multiples of 4 bytes



Alignment

- This is called Alignment: objects must fall on address that is multiple of their size.
- See why when get to caches, pipelining

Freie Universität Bozen
Libera Università di Bolzano
Free University of Bozen · Bolzano

Example of memory access

```
void main(void){
  int A[2],*ptr;
  ptr=A;
  A[0]=5;
  A[1]=10;
  ptr = ptr + 1;
  *ptr = *ptr + 1;
}
```

```
                                   # nothing
add   $s0,$0,$s1                   # $s0 = $s1
addi  $t0, $0,  5                  # $t0 = 5
sw    $t0, 0($s1)                  # M[$s0]=$t0
addi  $t0, $0,  10                 # $t0 = 10
sw    $t0, _ ($s1)                 # M[$s0+?]=$t0
addi  $s0, $s0, _                  # $s0 += ?
lw    $t0, 0($s0)                  # $t0 = M[$s0]
addi  $t0, $t0, _                  # $t0 += ?
sw    $t0, 0($s0)                  # M[$s0]=$t0
```

What goes in the 3 boxes?

Compilation issues

- What offset in lw to select A[8] in Java?
  - 4x8=32 to select A[8]: byte v. word
- Compile by hand using registers:
  g = h + A[8];
  - g: $s1, h: $s2, $s3:base address of A
- 1st transfer from memory to register:

```
lw $t0,32($s3)      # $t0 gets A[8]
```

- Add 32 to $s3 to select A[8], put into $t0
- Next add it to h and place in g

```
add $s1,$s2,$t0   # $s1 = h+A[8]
```

Notes about Memory

- Pitfall: Forgetting that sequential word addresses in machines with byte addressing do not differ by 1.
  - Many an assembly language programmer has toiled over errors made by assuming that the address of the next word can be found by incrementing the address in a register by 1 instead of by the word size in bytes.
  - So remember that for both lw and sw, the sum of the base address and the offset must be a multiple of 4 (to be word aligned)

"And in Conclusion…" (1/2)

- In MIPS Assembly Language:
  - Registers replace C variables
  - One Instruction (simple operation) per line
  - Simpler is Better
  - Smaller is Faster
- Memory is byte-addressable, but lw and sw access one word at a time.
- A pointer (used by lw and sw) is just a memory address, so we can add to it or subtract from it (using offset).

“And in Conclusion…”(2/2)

- New Instructions:
  - lw, sw

In summary

- **If we deal with g = h + A [8]**
- **Assumption**
  - **A is an array of 100 words**
  - **Start address of the arrays is in $s3**
  - **g, h are associated to $s1, $2**
- **Code for the operation**
  - **lw $t0, 32($s3)**
  - **add $s1, $s2, $t0**
- **Notice the inverse use of the parentheses**

More complex example

- If we deal with g = h + A [i]
- Assumption
  - A: array of 100 words with start address in $s3
  - g, h, i are associated to $s1, $s2, $s4
- Code for calculating address of A[i]:
  - `add     $t1, $s4, $s4          # t1 = 2 * i`
  - `add     $t1, $t1, $t1          # t1 = 4 * i`
  - `add     $t1, $t1, $s3          # t1 = address of A[i]`
- Code for the core operation
  - `lw      $t0, 0($t1)`
  - `add     $s1, $s2, $t0`