Assembler Arithmetic an ess

Overview

- Variables in Assembly
- Addition and Subtraction in Assembly
- Memory Access in A

Below Your Program

- High-level language program (in C)

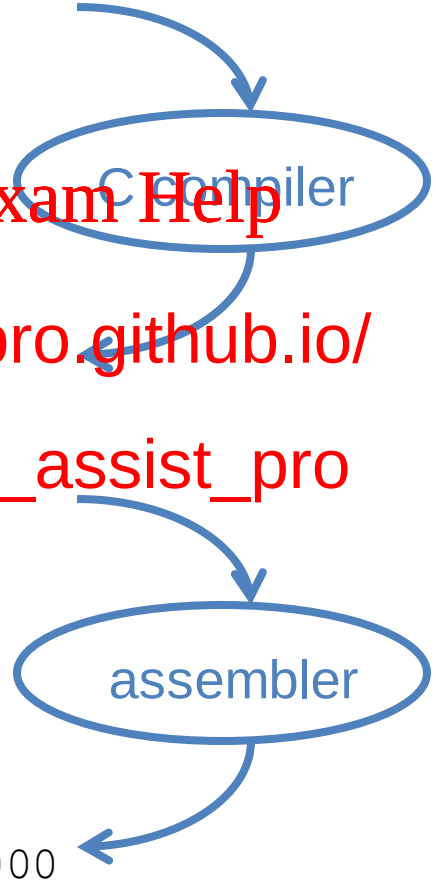
```
swap (int v[], int k) {
    int temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}
```

- Assembly language pr

```
swap:   sll     $2, $4,2
        add     $2, $4,$2
        lw      $15, 0($2)
        lw      $16, 4($2)
        sw      $16, 0($2)
        sw      $15, 4($2)
        jr      $31
```

- Machine (object) code (for MIPS)

```
000000 00000 00101 00010000010000000
000000 00100 00010 00010000000100000
 . . .
```



Operators / Operands in High-level Languages

Operators: +, -, *, /, % ;

- 7/4==1, 7%4==3

Operands:

- Variables: fahr, celsius
- Constants: 0, 1000, -17, 15.4

Statement: Variable = Expression ;

- celsius = 5*(fahr-32)/9;
- a = b+c+d-e;

Assembly Design: Key Concepts

- Assembly language is directly supported in hardware
- It is kept very simple!
  - Limit on the type of
  - Limit the set of **oper** imum

The MIPS Instruction Set

- Microprocessor without Interlocked Pipelined Stages (MIPS)
  ple in this course (Quickguide)

MARS: Free MIPS Simulator

- Download the _____
- Run the software java –jar pMARS.jar

How do I learn MIPS assembly?

- Try it out with MARS!

Assembly Variables: Registers

C and Java

- Operands are **variables and constants**
- Declare as many as you

MIPS

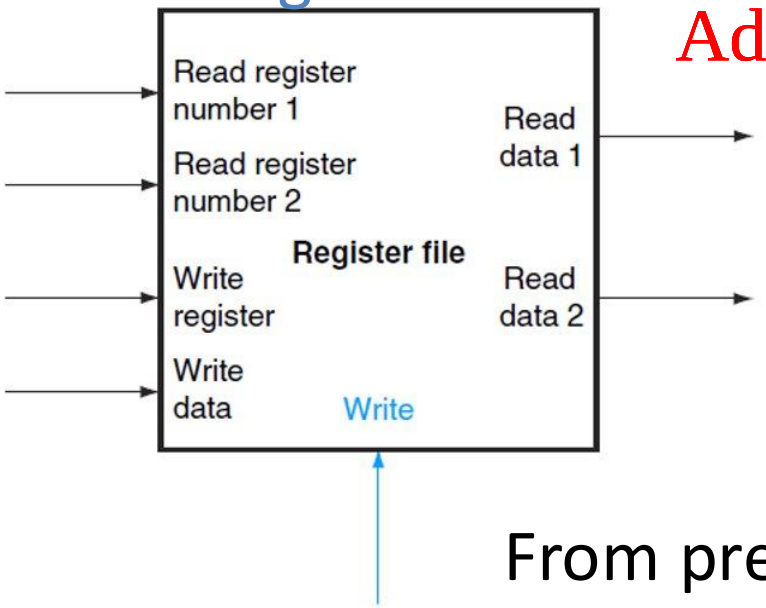
- Variables are replaced by **registers**
- tions can only
  rmed on these!
- number built
  directly into the hardware

Why? Keep the Hardware Simple!

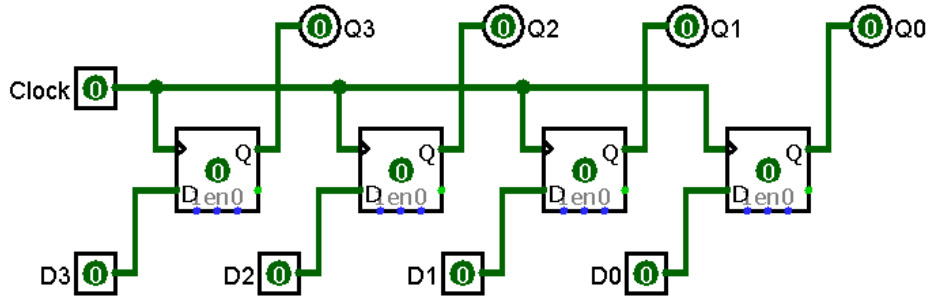
Assembly Variables: Registers

- MIPS has a register file of 32 registers
- Why 32? Smaller is faster
- Each MIPS register is 32 bits = 4 bytes = a word

Register File

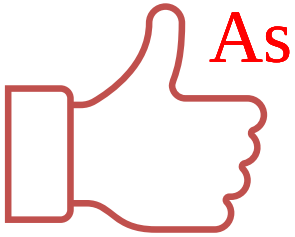


General) Register



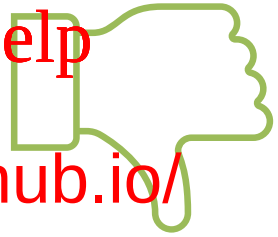
From previous lecture on "Register and Memory"

Assembly Variables: Registers



Good

Register file is small and inside of the core, so they are very fast



Bad

Since registers are implemented in the hardware, there are a predetermined number of them

MIPS code must be very carefully put together to efficiently use registers

Assembly Variables: Registers

- Registers are numbered from 0 to 31

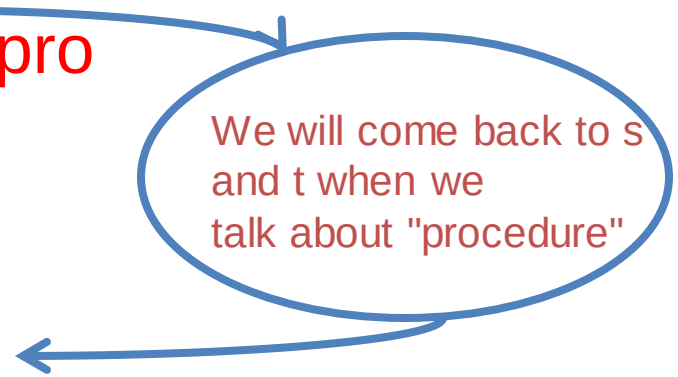
  `$0, $1, $2, … $30, $31`

- Each register also has a **name** to make it easier to code:

  `$16 - $23` ➔ (s correspond to saved temporar)

  `$8 - $15` ➔ `$t0 - $t7`

  (t correspond to temporary variables)



In general, **use register names** to make your code more readable

Assembly Variables: Registers

$1, $26, $27 are reserved for assembler and operation system

Comments

Assembly code is hard to read!

Another way to **make your code more readable**: comments!

C and Java

```
/* comment can span many lines */
// comment, to the end of a line
```


MIPS

```
# from hash mark to end of line is a comment and will be ignored
```

Assembly Instructions

C and Java

Each statement could r
multiple operations

```
a = b + c - d ;
```

Is equivalent to two small operations

```
a = b + c ;
a = a – d ;
```

MIPS

tement
n instruction), executes
ne of a short list of
simple commands

Addition and Subtraction

- **Syntax of Instructions:**

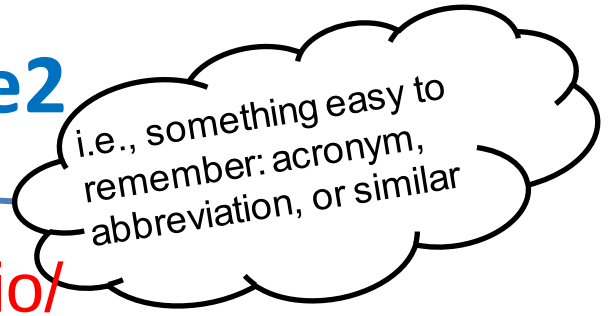
  Operation Destination, Source1, Source2

  Operation: by name (

  Destination: operand

  Source1: 1st operand for operation

  Source2: 2nd operand for operation



- Syntax is rigid:
  - Most of them use 1 operator + 3 operands ***(commas are optional)***
  - Why? Keep Hardware simple via regularity

Addition and Subtraction

Try with Mars

Addition

```
// C and Java
a = b + c ;
```

```
# MIPS
add $s1 $s2
```

registers $s0,$s iables a, b, c

Subtraction

```
// C and Java
d = e - f;
```

```
# MIPS
sub $s3,$s4,$s5
```

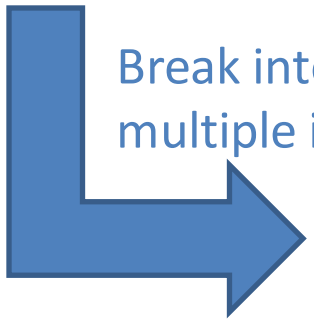
registers $s3,$s4,$s5 are associated with variables d, e, f

Addition and Subtraction

Each Instruction, executes exactly one simple commands

C and Java

```
a = b + c + d -
```

MIPS

```
                s0, $s1, $s2
                # b + c
                , $s0, $s3
        # a = a + d

        sub $s0, $s0, $s4
        # a = a - e
```

A single line of C may break up into several lines of MIPS.

Immediates

- **Immediates** are numerical constants.
- Special instructions for immediates: addi
- Syntax is similar to a pt that ***last*** argument is a number (decima stead of a register.


```
// C and Java
f = g + 10 ;
```

```
addi $s0 $s1 10
addi $s0 $s1 -10
```

- There is no subi (use a negative immediate instead)

Register Zero

- MIPS defines **register zero** (`$0` or `$zero`) ***always*** be 0.
- The number zero appears very often in code.
- Use this register, it'


```
add   $6 $0 $5                5 to $6
addi  $6 $0 77          # copy 77 to $6
```

- Register zero cannot be overwritten

```
addi  $0 $0 5           # will do nothing
```

Register Zero

- What if you want to negate a number?

```
sub $6,$0,$5
```


Data ctions

Data Transfer Instructions

- **MIPS arithmetic instructions only operate on registers**
- What about large data structures like arrays? **Memory**!
  - Add two numbers in
    - Load values from me
    - Store result from register to memor
- Use **Data transfer instructions** to transfer data between registers and memory. We need to specify
  - Register: specify this by number (0 - 31)
  - Memory address: more difficult

Memory Address

Memory is a linear array of byte

the memory has its own ss

We can access the content by supplying the memory address

The processor can read or write the content of the memory

Memory Address

- **Memory Address Syntax: Offset(AddrReg)**
  - **AddrReg: A register which contains a pointer to a memory location**
  - **Offset: A numerical offset in bytes** (optional)


```
8($t0)
# specifies the memory add          0 plus 8 bytes
```

- We might access a location with an offset from a base pointer
- The resulting memory address is the sum of these two values

Memory Address

```
// An array of 8 integers in C/Java
int arr[8]={56,26,88,45,-45,77,98,13} ;
```



```
# Assume $s0 has the address 0x1000
```



```
0($s0)   # 0x1000, to access arr[0]
4($s0)   # 0x1004, to access arr[1]
```

4-bit address example

Address	Content
	…
0x1000	56
0x1004	26
0x1008	88
0x100C	45
0x1010	-45
0x1014	77
0x1018	98
0x101C	13
	…

Data Transfer: Memory to Register

- **Load Instruction Syntax: lw DstReg, Offset(AddrReg)**
  - **lw**: Load a **Word**
  - **DstReg**: register tha
  - **Offset**: numerical of
  - **AddrReg**: register containing poin


```
lw $t0, 8($s0)
# load one word from memory at
address stored in $s0 with an offset 8 and
store the content in $t0
```

Address	Content
	…
0x1000	56
0x1004	26
0x1008	88
0x100C	45
0x1010	-45
0x1014	77
0x1018	98
0x101C	13
	…

Data Transfer: Register to Memory

- **Store instruction syntax: sw DataReg, Offset(AddrReg)**
  - sw: Store a **word**
  - **DstReg**: register
  - **Offset**: numerica
  - **AddrReg**: register containing


```
sw $t0, 4($s0)
# Store one word (32 bits) to memory
address $s0 + 4
```

Address	Content
	…
0x1000	56
0x1004	26
0x1008	88
0x100C	45
0x1010	-45
0x1014	77
0x1018	98
0x101C	13
	…

Byte vs. word

- Machines address memory as **bytes**
- **Both lw and sw access one word at a time**
- The sum of the base address and the offset ***must be a*** be word aligned)


```
sw $t0, 0
sw $t0, 4($s0)
sw $t0, 8($s0)
        .
        .
```

Address	Content
	…
0x1000	56
0x1004	26
0x1008	88
0x100C	45
0x1010	-45
0x1014	77
0x1018	98
0x101C	13
	…

Byte vs. word

Try with Mars

```
// C and Java
A[12] = h + A[8] ;
```

Index 8 requires offset of 32
Index 12 reugires offset of 48


```
# MIPS
# assume h is stored in $s0 and th      dress of A is in $s1
lw   $s2 32($s1)   # load A[8] to $s2
add $s3 $s0, $s2 # $s3 = $s0 + $s2
sw   $s3 48($s1)   # store result to A[12]
```

Register vs. Memory

- MIPS arithmetic instructions can read 2 registers, operate on them, and write 1 per instruction

write 1 operand per instruction, and no operation

Why not keep all variables in memory?

- Smaller is faster

What if more variables than registers?

- Compiler tries to keep most frequently used variable in registers
- Writing less common to memory: ***spilling***

Pointers vs. Values

- **A register can hold *any* 32-bit value.**
  - a (signed) `int`,
  - an `unsigned int`
  - a pointer (memory a
  - etc.

```
lw $t2, 0($t0)        # $t0 must contain?
```

```
add $t3, $t4, $t5  # what can you say about $t4 and $t5?
```

Review and Information

Registers:

- The variables in assembly
- Saved Temporary Variables, Temporary Variables, Register Zero

Instructions:

- Addition and Subtraction: add, addi, sub

- Data Transfer: lw, sw

References

- Textbook: 2.1, 2.2, 2.3, A.10
- MARS Tutorial