CSE 31
Computer Organization

Lecture 8 – MIPS: Conditionals

Announcement

- Lab #3 this week
  - Due in one week
- HW #2 (at CatCourses)
  - Written homework, NOT from zyBooks
  - Type your answers or scan and submit trough CatCourses
  - Due Monday (9/24)
  - Sample exam online
- Project #1 out this Friday
  - Due Monday (10/22)
    - Don’t start late, you won’t have time!
- Reading assignment
  - Chapter 2.1 – 2.9 of zyBooks (Reading Assignment #2)
    - Make sure to do the Participation Activities
    - Due Wednesday (9/26)

Announcement

- Midterm exam Wednesday (10/3, postponed)
  - Lectures 1 – 7
  - HW #1 and #2
  - Closed book
  - 1 sheet of note (8.5” x 11”)

Assembly Instructions

- In assembly language, each statement (called an Instruction), executes exactly one of a short list of simple commands
- Unlike in C (and most other High Level Languages), each line of assembly code contains at most 1 instruction
- Instructions are related to operations (=, +, -, *, /) in C or Java
- Ok, ready for MIPS?

MIPS Addition and Subtraction (1/4)

- Syntax of Instructions:

  Format: 1,2,3,4

  where:

  1) operation by name

  2) operand getting result ("destination")

  3) 1st operand for operation ("source1")

  4) 2nd operand for operation ("source2")

- Syntax is rigid:
  - 1 operator, 3 operands
  - Why?
    - Keep Hardware simple via regularity

Addition and Subtraction of Integers (1/3)

- Addition in Assembly
  - Example: `add $s0,$s1,$s2` (in MIPS)
    Equivalent to: `a = b + c` (in C)

  where MIPS registers `$s0,$s1,$s2` are associated with C variables `a, b, c`
- Subtraction in Assembly
  - Example: `sub $s3,$s4,$s5` (in MIPS)
    Equivalent to: `d = e - f` (in C)

  where MIPS registers `$s3,$s4,$s5` are associated with C variables `d, e, f`

Addition and Subtraction of Integers (2/3)

- How do the following C statement work in MIPS?

  a = b + c + d - e;

- Break into multiple instructions

```
add $t0, $s1, $s2 # temp = b + c
add $t0, $t0, $s3 # temp = temp + d
sub $s0, $t0, $s4 # a = temp - e
```

  - Notice: A single line of C may break up into several lines of MIPS.
  - Notice: Everything after the hash mark on each line is ignored (comments)

Addition and Subtraction of Integers (3/3)

- How do we do this?

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

- Use intermediate temporary register

```
add $t0,$s1,$s2    # temp = g + h
add $t1,$s3,$s4    # temp = i + j
sub $s0,$t0,$t1    # f=(g+h)-(i+j)
```

Immediates

- **Immediates** are numerical constants.
- They appear often in code, so there are special instructions for them.
- Add Immediate:

  `addi` `$s0,$s1,10` (in MIPS)

  `f = g + 10` (in C)

  where MIPS registers `$s0,$s1` are associated with C variables `f, g`
- Syntax similar to `add` instruction, except that last operand is a number instead of a register.

Register Zero

- One particular immediate:
  - The number zero (0), appears very often in code.
- So we define register zero (`$0` or `$zero`) to always have the value 0

  `add $s0,$s1,$zero` (in MIPS)

  `f = g` (in C)

  where MIPS registers `$s0,$s1` are associated with C variables `f, g`
- defined in hardware, so an instruction

  `add $zero,$zero,$s0`

  will not do anything!

Immediates

- There is no Subtract Immediate in MIPS: Why?
- Limit types of operations that can be done to absolute minimum
  - if an operation can be decomposed into a simpler operation, don't include it
  - `addi …, -X` = `subi …, X` => so no `subi`
- `addi $s0,$s1,-10` (in MIPS)

  `f = g - 10` (in C)

  where MIPS registers `$s0,$s1` are associated with C variables `f, g`

Quiz

1) Since there are only 8 local (`$s`) and 8 temp (`$t`) variables, we can't write MIPS for C exprs that contain > 16 vars.

2) If `p` (stored in `$s0`) is a pointer to an array of `int`s, then `p++;` would be `addi $s0 $s0 1`

	12
a)	FF
b)	FT
c)	TF
d)	TT
e)	dunno

Quiz

1) Since there are only 8 local (`$s`) and 8 temp (`$t`) variables, we can't write MIPS for C exprs that contain > 16 vars.

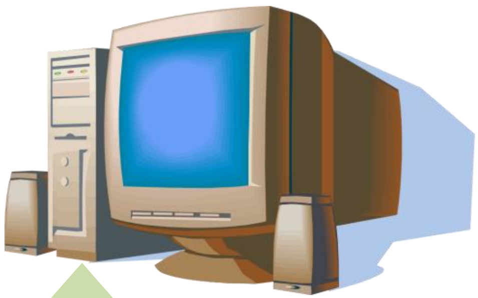
2) If `p` (stored in `$s0`) is a pointer to an array of `int`s, then `p++;` would be `addi $s0 $s0 1`

```
    12
a)  FF
b)  FT
c)  TF
d)  TT
e) dunno
```

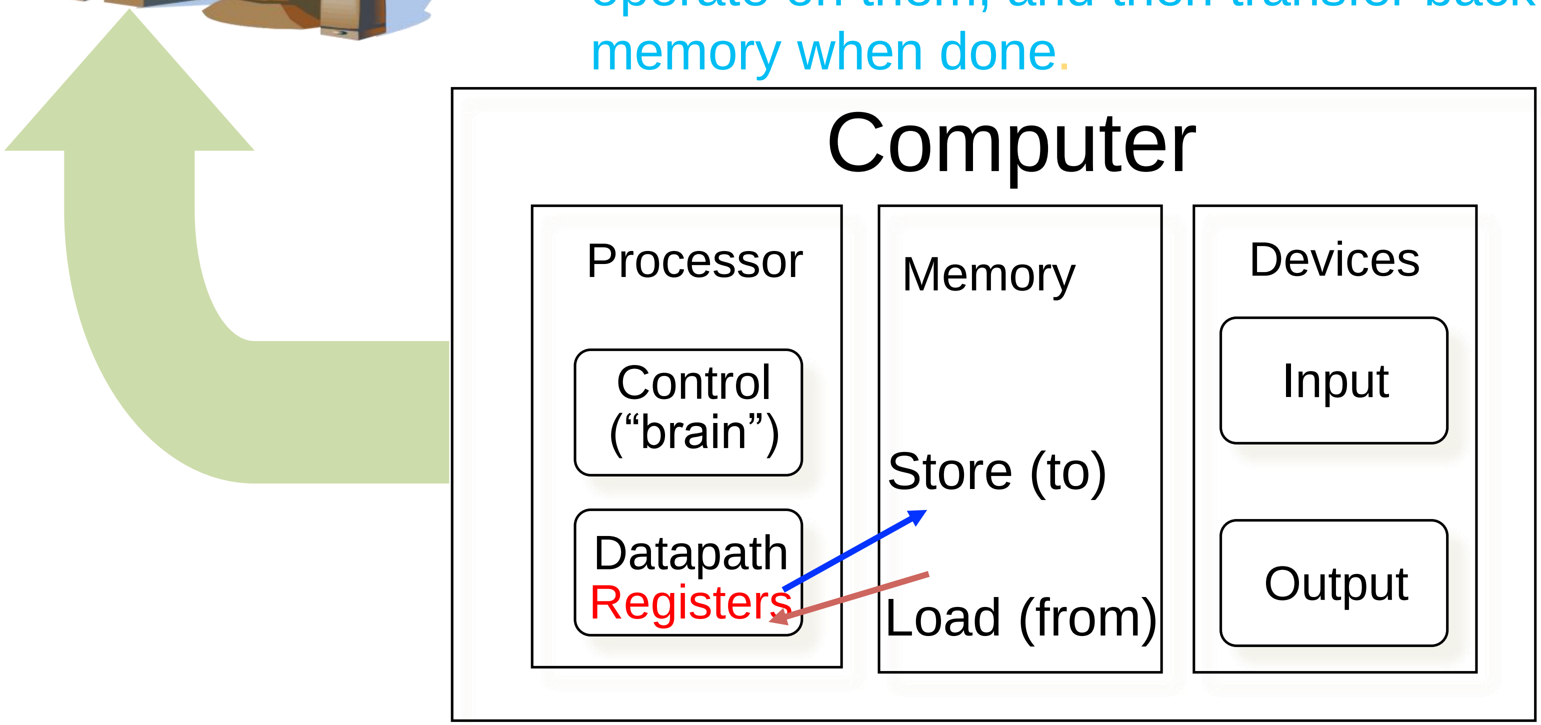
Assembly Operands: Memory

- C variables map onto registers; what about large data structures like arrays?
- 1 of 5 components of a computer: memory contains such data structures
- But MIPS arithmetic instructions only operate on registers, never directly on memory.
- Data transfer instructions transfer data between registers and memory:
  - Memory to register
  - Register to memory

Anatomy: 5 components of any Computer



Registers are in the datapath of the processor; if operands are in memory, we must transfer them to the processor to operate on them, and then transfer back to memory when done.



These are "data transfer" instructions…

Data Transfer: Memory to Reg (1/4)

- To transfer a word of data, we need to specify two things:
  - Register: specify this by # ($0 - $31) or symbolic name ($s0,…,$t0,…)
  - 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.
- Remember: “Load FROM memory”

Data Transfer: Memory to Reg (2/4)

- To specify a memory address to load from, specify two things:
  - A register containing a pointer to memory
  - A numerical offset (in bytes), how far away from the address
- The desired memory address is the sum of these two values.
- Example: `8($t0)`
  - specifies the memory address pointed to by the value in `$t0`, plus 8 bytes

Data Transfer: Memory to Reg (3/4)

- Load Instruction Syntax:
  Format: 1,2,3(4)
  - where
    1) operation name
    2) register that will receive value
    3) numerical offset in bytes
    4) register containing pointer to memory

- MIPS Instruction Name:
  - `lw` (meaning **Load Word**, so 32 bits (one word) are loaded at a time)

Data Transfer: Memory to Reg (4/4)

Data flow

Example: `lw $t0,12($s0)`

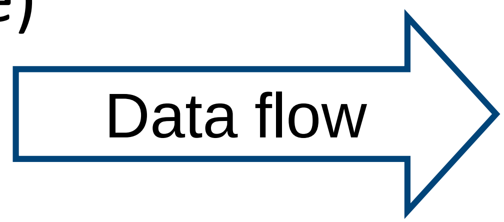
This instruction will take the pointer stored in $s0, add 12 bytes to it, and then load the value from the memory pointed to by this calculated sum into register $t0

- 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 (note offset must be a constant known at assembly time)

Data Transfer: Reg to Memory

- Also want to store from register into memory
  - Store instruction syntax is identical to Load's
- MIPS Instruction Name:

  `sw` (meaning Store Word, so 32 bits or one word is stored at a time)



- 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 that memory address
- Remember: "Store INTO memory"

Pointers vs. Values

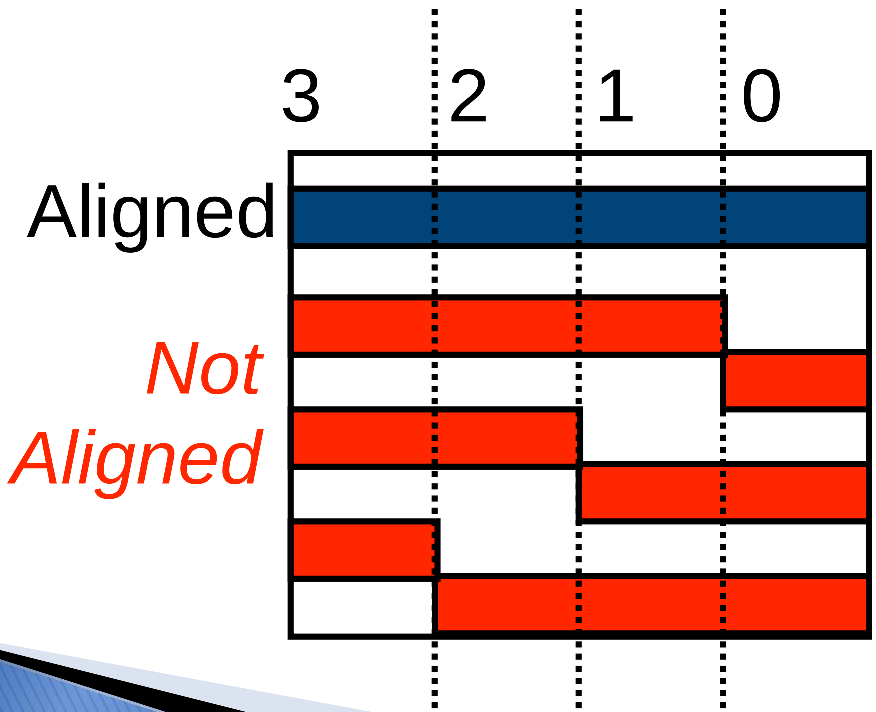
- Key Concept: A register can hold any 32-bit value. That value can be a (signed) int, an unsigned int, a pointer (memory addr), and so on
  - E.g., If you write: `add $t2,$t1,$t0 # c = b + A;` then `$t0` and `$t1` better contain values that can be added
  - E.g., If you write:

    ```
    lw $t2, 0($t0) # c = A[0];
    add $t2, $t2, $t1 #c=A[0]+b
    ```

    then `$t0` better contains a pointer
- Don't mix these up!

More Notes about Memory: Alignment

- MIPS requires that all words start at byte addresses that are multiples of 4 bytes
- Called Alignment: objects fall on address that is multiple of their size



Last hex digit of address is:

0, 4, 8, or C_{hex}

1, 5, 9, or D_{hex}

2, 6, A, or E_{hex}

3, 7, B, or F_{hex}

Notes about Memory

- Pitfall: Forgetting that sequential word addresses in machines with byte addressing do not differ by 1.
  - Many assembly language programmers have 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.
  - Also, 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)

Role of Registers vs. Memory

- What if more variables than registers?
  - Compiler tries to keep most frequently used variable in registers
  - Less common variables in memory: <u>spilling</u>
- 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

Compilation with Memory

- What offset in `lw` to select `A[5]` in C?
  - 4x5=20 to select `A[5]`: byte vs. word
- Compile by hand using registers:

  `g = h + A[5];`

  `g`: `$s1`, `h`: `$s2`, `$s3`: base address of `A`
- 1st transfer from memory to register:

```
lw $t0,20($s3)  # $t0 gets A[5]
```

  - Add 20 to `$s3` to select `A[5]`, put into `$t0`
- Next add it to h and place in `g`

```
add $s1,$s2,$t0   # $s1 = h+A[5]
```

Quiz

We want to translate `*x = *y` into MIPS
(`x`, `y` ptrs stored in: `$s0` `$s1`)

```
1: add $s0,    $s1, zero
2: add $s1,    $s0, zero
3: lw  $s0,  0($s1)
4: lw  $s1,  0($s0)
5: lw  $t0,  0($s1)
6: sw  $t0,  0($s0)
7: lw  $s0,  0($t0)
8: sw  $s1,  0($t0)
```

a) 1 or 2
b) 3 or 4
c) 5 → 6
d) 6 → 5
e) 7 → 8

Quiz

We want to translate `*x = *y` into MIPS
(`x`, `y` ptrs stored in: `$s0 $s1`)

```
1: add $s0, $s1, zero
2: add $s1, $s0, zero
3: lw  $s0, 0($s1)
4: lw  $s1, 0($s0)
5: lw  $t0, 0($s1)
6: sw  $t0, 0($s0)
7: lw  $s0, 0($t0)
8: sw  $s1, 0($t0)
```

a) 1 or 2
b) 3 or 4
c) 5 → 6
d) 6 → 5
e) 7 → 8

So Far...

- All instructions so far only manipulate data…we've built a calculator of sorts.

- In order to build a computer, we need ability to make decisions…

- C (and MIPS) provide labels to support "goto" jumps to places in code.
  - C: Horrible style;
  - MIPS: Necessary!

C Decisions: `if` Statements

- 2 kinds of if statements in C

  `if` *(condition) clause*

  `if` *(condition) clause1* `else` *clause2*

- Rearrange 2nd if into following:

  `if` *(condition)* `goto L1;`

  clause2;

  `goto L2;`

  `L1:` *clause1;*

  `L2:`

- Not as elegant as if-else, but same meaning

MIPS Decision Instructions

- Decision instruction in MIPS:

  `beq   register1, register2, L1`

  `beq` is "Branch if (registers are) equal"

  Same meaning as (using C):

  ```
  if  (register1==register2) goto L1
  ```

- Complementary MIPS decision instruction

  `bne   register1, register2, L1`

  `bne` is "Branch if (registers are) not equal"

  Same meaning as (using C):

  ```
  if  (register1!=register2) goto L1
  ```

- Called <u>conditional branches</u>

MIPS Goto Instruction

- In addition to conditional branches, MIPS has an unconditional branch:

  `j label`

- Called a Jump Instruction: jump (or branch) directly to the given label without needing to satisfy any condition
- Same meaning as (using C): `goto label`
- Technically, it's the same effect as:

  `beq $0,$0,label`

  since it always satisfies the condition.

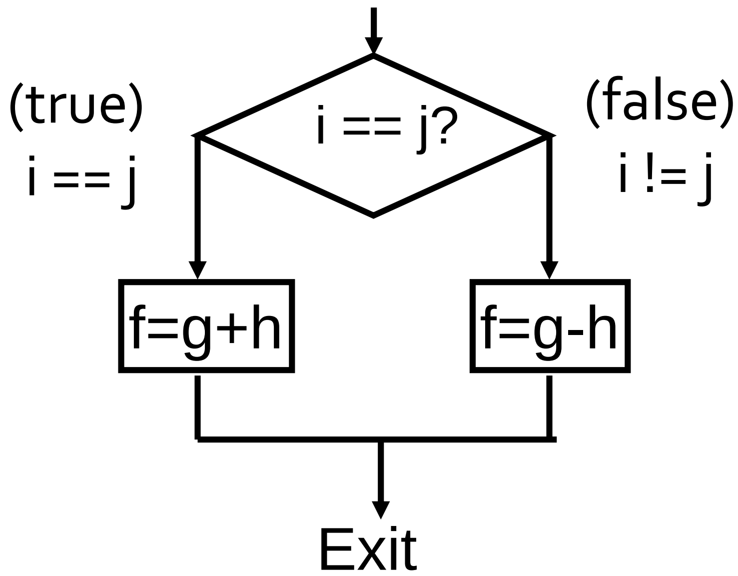
Compiling C `if` into MIPS (1/2)

- Compile by hand

```
if (i == j) f=g+h;
else f=g-h;
```

- Use this mapping:

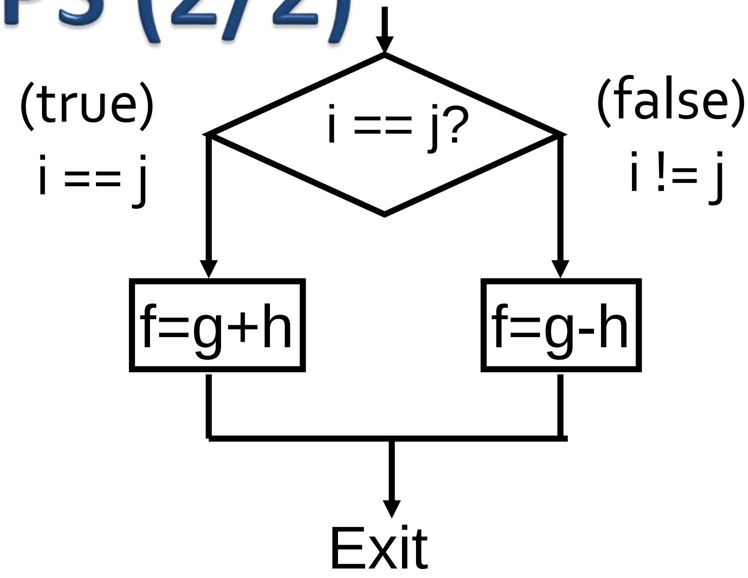
```
f: $s0
g: $s1
h: $s2
i: $s3
j: $s4
```



Compiling C `if` into MIPS (2/2)

- Compile by hand

```
if (i == j) f=g+h;
else f=g-h;
```



f: $s0, g: $s1, h: $s2, i: $s3, j: $s4

- Final compiled MIPS code:

```
      beq $s3,$s4,True   # branch i==j
      sub $s0,$s1,$s2    # f=g-h(false)
      j   Fin            # goto Fin
True: add $s0,$s1,$s2    # f=g+h (true)
Fin:
```

Note: Compiler automatically creates labels to handle decisions (branches). Generally not found in HLL code.

Loading, Storing bytes 1/2

- In addition to word data transfers (**`lw`**, **`sw`**), MIPS has byte data transfers:
  - load byte: **`lb`**
  - store byte: **`sb`**
- Same format as `lw`, `sw`
- E.g., **`lb $s0, 3($s1)`**
  - contents of memory location with address = sum of "3" + contents of register ***s1*** is copied to the **low byte position** of register ***s0***.

Loading, Storing bytes 2/2

- What to do with other 24 bits in the 32 bit register?
  - lb: sign extends to fill upper 24 bits

xxxx xxxx xxxx xxxx xxxx xxxx x zzz zzzz

…is copied to "sign-extend"

byte loaded

This bit

- Normally don't want to sign extend chars
- MIPS instruction that doesn't sign extend when loading bytes:
  - load byte unsigned: **lbu**

Overflow in Arithmetic (1/2)

- Reminder: Overflow occurs when there is a mistake in arithmetic due to the limited precision in computers.
- Example (4-bit unsigned numbers):

15	1111
+ 3	+ 0011
18	10010

  - But we don’t have room for 5-bit solution, so the solution would be **0010**, which is **+2**, and wrong.

Overflow in Arithmetic (2/2)

- Some languages detect overflow (Ada), some don't (C)
- MIPS solution is 2 kinds of arithmetic instructs:
  - These cause overflow to be detected
    - add (**add**)
    - add immediate (**addi**)
    - subtract (**sub**)
  - These do not cause overflow detection
    - add unsigned (**addu**)
    - add immediate unsigned (**addiu**)
    - subtract unsigned (**subu**)
- Compiler selects appropriate arithmetic
  - MIPS C compilers produce **addu, addiu, subu**

Two “Logic” Instructions

- Here are 2 more new instructions
- Shift Left: `sll $s1,$s2,2 #s1=s2<<2`
  - Store in `$s1` the value from `$s2` shifted 2 bits to the left (they fall off end), inserting 0’s on right; << in C.
  - Before: `0000 0002`$_{hex}$
    `0000 0000 0000 0000 0000 0000 0000 0010`$_{two}$
  - After: `0000 0008`$_{hex}$
    `0000 0000 0000 0000 0000 0000 0000 1000`$_{two}$
  - What arithmetic effect does shift left have?
    - $n \times 2^i$
- Shift Right: `srl` is opposite shift; **>>**

Loops in C/Assembly (1/3)

- Simple loop in C; **A[]** is an array of `int`

```
do {    g = g + A[i];
       i = i + j;
} while (i != h);
```

How to write this in MIPS using what we have learned so far?

- Rewrite this as:

```
Loop:   g = g + A[i];
        i = i + j;
        if (i != h) goto Loop;
```

- Use this mapping:

```
  g,   h,   i,   j, base of A
$s1, $s2, $s3, $s4, $s5
```

Loops in C/Assembly (2/3)

- Final compiled MIPS code:

Why???

```
Loop: sll  $t1,$s3,2      # $t1= 4*I
      addu $t1,$t1,$s5    # $t1=addr A+4i
      lw   $t1,0($t1)     # $t1=A[i]
      addu $s1,$s1,$t1    # g=g+A[i]
      addu $s3,$s3,$s4    # i=i+j
      bne  $s3,$s2,Loop   # goto Loop
                          # if i!=h
```

- Original code:

```
Loop:  g = g + A[i];
       i = i + j;
       if (i != h) goto Loop;
```

```
  g,   h,   i,   j, base of A
 $s1, $s2, $s3, $s4, $s5
```

Loops in C/Assembly (3/3)

- There are three types of loops in C:
  - `while`
  - `do… while`
  - `for`
- Each can be rewritten as either of the other two, so the method used in the previous example can be applied to these loops as well.
- Key Concept: Though there are multiple ways of writing a loop in MIPS, the key to decision-making is ***conditional branch***