Chapter 2

MIPS instructions

Registers

Storage locations for information inside the CPU

- 32 Registers, $0 ... $31
- Register $0 is always 0
- Required for arithmetic and logic operations
- Access time = Clock frequency of processor

Name	Register Number	Usage	Preserve on Call
$zero	0	constant 0	n.a.
$v0 - $v1	2 - 3	values for result and expression evaluation	no
$a0 - $a3	4 - 7	arguments	no
$t0 - $t7	8 - 15	temporaries	no
$s0 - $s7	16 - 23	saved	yes
$t8 - $t9	24 - 25	more temporaries	no
$gp	28	global pointer	yes
$sp	29	stack pointer	yes
$fp	30	frame pointer	yes
$ra	31	return address	yes

Registers as Operand (Register Addressing)

- Addition and Subtraction (Integers)
- Syntax of Instructions:
  - 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

Addition and Subtraction

Register as operand

- Addition in Assembly
  - Example: add $s0,$s1,$s2 (in MIPS)
- Equivalent to: a = b + c (in C)
  - where registers $s0,$s1,$s2 are associated with variables a, b, c

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

Addition and Subtraction

Register as operand

- How do the following statement?
  - a = b + c + d - e
- Assume
  - a is in s0
  - b is in s1
  - c is in s2
  - d is in s3
  - e is in s4

Addition and Subtraction

Register as operand

- Break into multiple instructions

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

Addition and Subtraction

Register as operand

- How do we do this?
  - f = (g + h) - (i + j)
- Assume
  - f is in s0
  - g is in s1
  - h is in s2
  - i is in s3
  - j is in s4

Addition and Subtraction

Register as operand

- Use intermediate temporary register

```
add $s0,$s1,$s2    # f = g + h
add $t0,$s3,$s4    # t0 = i + j
                   # need to save i+j, but can't use f, so use t0
sub $s0,$s0,$t0    # f=(g+h)-(i+j)
```

Addition and Subtraction

Immediate operand (constant) : Immediate Addressing

- Immediates are numerical constants.
- They appear often in code, so there are special instructions for them.
- Add Immediate:
  - addi $s0,$s1,10
  - f = g + 10
  - where registers $s0,$s1 are associated with variables f, g
- Syntax similar to add instruction, except that last argument is a number instead of a register.

Addition and Subtraction

No Immediate Subtraction

- 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 $s0,$s1,-10
  - f = g - 10
  - where registers $s0,$s1 are associated with variables f, g
- addi …, -X = subi …, X
  - so no subi

Addition and Subtraction

Register as operand

- How do we do this?
  - $f = (g + 2) - (h - 7)$
- Assume
  - f is in s0
  - g is in s1

Addition and Subtraction

Register as operand

- Again use temporary variable

```
addi $s0,$s1,2      # f = g + 2
addi $t0,$s2,-7     # t0 = h - 7
                    # need to use t0 as temporary
sub $s0,$s0,$t0     # f=(g+h)-(i+j)
```

Register Zero

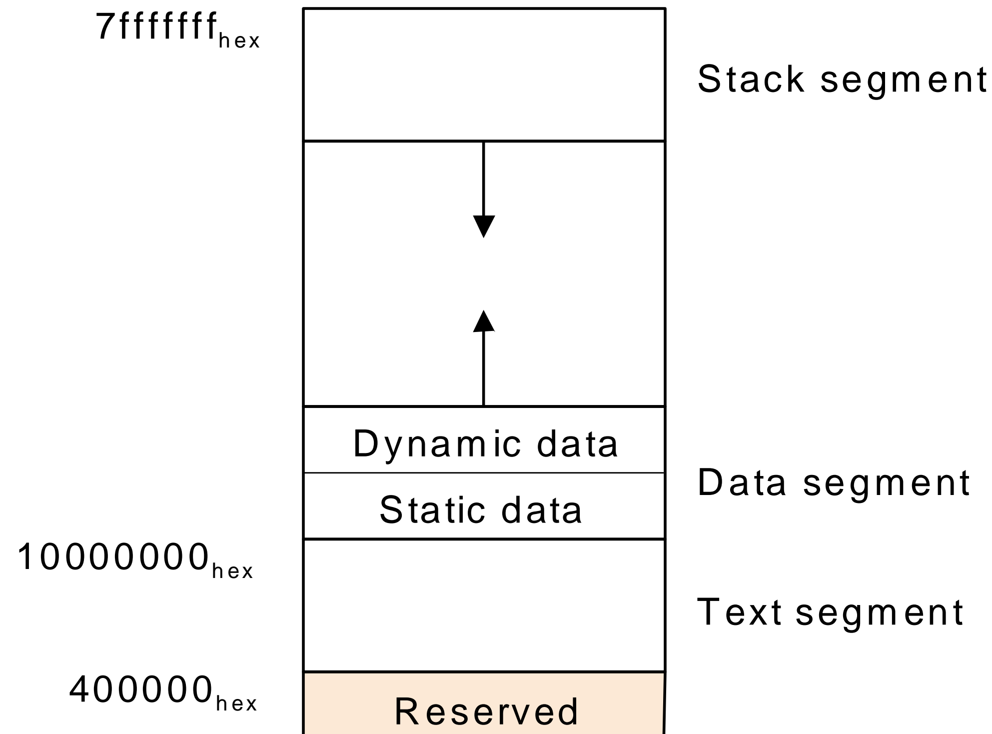
- One particular immediate, the number zero (0), appears very often in code.
- So we define register zero ($0 or $zero) to always have the value 0; eg
  - add $s0,$s1,$zero
  - f = g
  - where registers $s0,$s1 are associated with variables f, g
- defined in hardware, so an instruction
  - addi $0,$0,5
  - will not do anything!

Memory Operations

- 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

Memory Structure

- Divided into segments
  - Data
  - Text
  - Stack



Data Transfer: Memory to Reg

Base Addressing

- 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.
- 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:
    - Assume $t0 contains 0x1200
    - 8($t0)
    - specifies the memory address pointed to by the value in $t0, plus 8 bytes : 0x1200 + 8 = 0x1208

Data Transfer: Memory to Reg

Base Addressing

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

Data Transfer: Memory to Reg

Base Addressing

- 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: base reg points to beginning of array

Data Transfer: Reg to Memory

Base Addressing

- Also want to store value from a register into memory
- Store instruction syntax is identical to Load instruction syntax
- Instruction Name:
  - sw (meaning Store Word, so 32 bits or one word are loaded 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 the memory address pointed to by the calculated sum

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 address), etc.
- If you write
  - add $t2,$t1,$t0
  - then $t0 and $t1 better contain values
- If you write
  - lw $t2,0($t0)
  - then $t0 better contain a pointer
- Don’t mix these up!

Addressing: Byte vs word

- Every word in memory has an address, similar to an index in an array
- Early computers numbered words like C numbers elements of an array:
  - Memory[0], Memory[1], Memory[2], …
  - The number inside the [x] is called the "address" of a word
- Computers needed to access 8-bit bytes as well as words (4 bytes/word)
- Today machines address memory as bytes, hence word addresses differ by 4
  - Memory[0], Memory[4], Memory[8], …

Compilation with Memory

- What offset in lw to select A[8]?
- 4x8=32 to select A[8]: byte vs 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]

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
- 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

Load Immediate

Immediate Addressing

- Load an immediate value into a register
  - add $sp, $zero, $zero   # make sure it is zero
  - addi $sp, $sp, 4   # put 4 into $sp

  - Limited to 16 bit constant
- Load a large immediate value (32 bit) into a register
  - lui $s0, 61 # load upper 16 bits : 0000 0000 0011 1101
  -                 # $s0 : 0000 0000 0011 1101 0000 0000 0000 0000
  - addi $s0,$s0, 2304 # add lower 16 bits with 2304
  -                 # $s0 : 0000 0000 0011 1101 0000 1001 0000 0000

Decission

- Decisions: if, if-else
- Decisions: Multiple conditions and consequences
- Inequality

How a branch works

- PC - Program counter register
- Here is a sequence of instructions. The "load" and "add" represent typical instructions. The "jump" instruction shows the address we wish to put into the PC. (The acutal MIPS instruction for this involves details that we are skipping for the moment.)

```
Address     Instruction                PC just after this
            (details omitted)        instruction has executed
                                      (at the bottom of the cycle)

........    ....    ..........        00400000
00400000    load                      00400004
00400004    add                       00400008
00400008    jump    0x00400000        00400000
```

if Statements

- 2 kinds of if statements
  - if (condition) then
    - statement
  - if (condition) then
    - statement1
    - else
    - statement2
- Rearrange 2nd if into following:
  - if (condition) goto L1
    - statement2
    - go to L2
    - L1: statement1
    - L2:
- Not as elegant but works

MIPS Decision Instructions

Conditional branches (PC Relative Addressing)

- Decision instruction in MIPS:
  - beq register1, register2, L1
  - beq is "Branch if (registers are) equal"
- Same meaning as:
  - if (register1 == register2) goto L1

- Complementary MIPS decision instruction
  - bne register1, register2, L1
  - bne is "Branch if (registers are) not equal"
- Same meaning as:
  - if (register1 != register2) goto L1

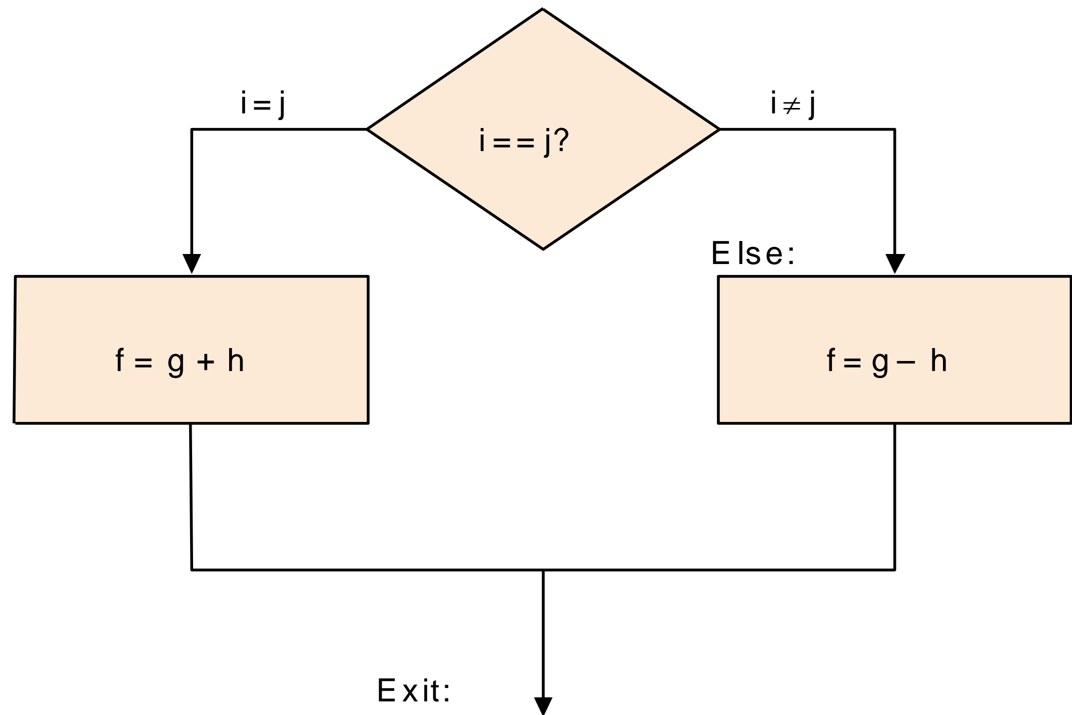
MIPS Goto Instruction

Unconditional branch (Pseudodirect Addressing)

- 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:
  - goto label
- Technically, it’s the same as:
  - beq $0,$0,label
  - since it always satisfies the condition.

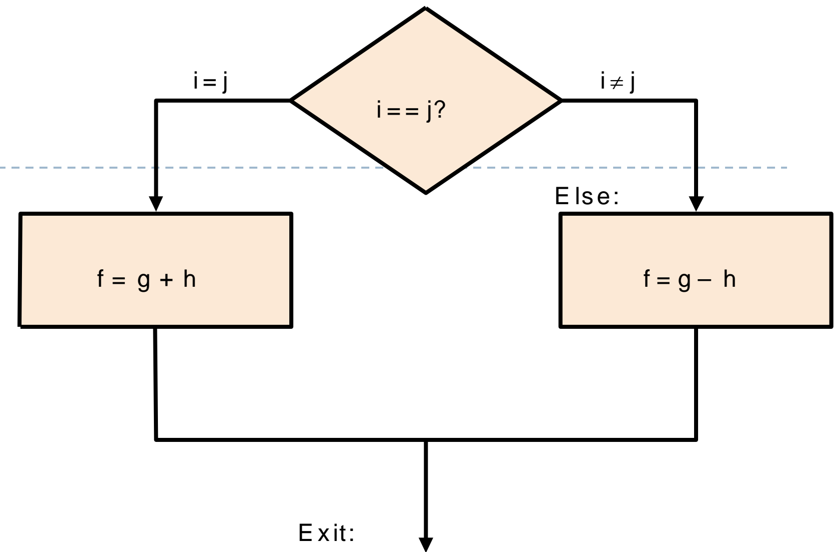
Compiling if into MIPS

- 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 if into MIPS



- Final compiled MIPS code:
  - beq $s3,$s4,True # branch i==j
  - sub $s0,$s1,$s2 # f=g-h(false)
  - j Fin # go to Fin
  - True: add $s0,$s1,$s2 # f=g+h (true)
  - Fin:
- Note: Compiler automatically creates labels to handle decisions (branches) appropriately.

Inequalities in MIPS

- Until now, we've only tested equalities (== and !=). General programs need to test < and > as well.
- Create a MIPS Inequality Instruction:
  - "Set on Less Than"
  - Syntax:
    - slt reg1,reg2,reg3
  - Meaning:
    - if (reg2 < reg3)
      - reg1 = 1;
    - else
      - reg1 = 0;
- In computereeze, "set" means "set to 1", "reset" means "set to 0".

Inequalities in MIPS

- How do we use this?
- Compile by hand:
  - if (g < h) goto Less;
- Use this mapping:
  - g: $s0, h: $s1

Inequalities in MIPS

- Final compiled MIPS code:
  - slt $t0,$s0,$s1 # $t0 = 1 if g<h
  - bne $t0,$0,Less # goto Less
  - # if $t0!=0
  - # (if (g<h)) Less:
- Branch if $t0 != 0 means (g < h)
- Register $0 always contains the value 0, so bne and beq often use it for comparison after an slt instruction.

Inequalities in MIPS

- Now, we can implement <, but how do we implement >, <= and >= ?
- Can we implement <= in one or more instructions using just slt and the branches?
- What about >?
- What about >=?

Immediates in Inequalities

- There is also an immediate version of slt to test against constants: slti
  - if (g >= 1) goto Loop

Immediates in Inequalities

- There is also an immediate version of slt o test against constants: slti
  - if (g >= 1) goto Loop

- Loop: . . .
  - slti $t0,$s0,1 # $t0 = 1 if
  - # $s0<1 (g<1)
  - beq $t0,$0,Loop # goto Loop
  - # if $t0==0
  - # (if (g>=1))

Multiple Condition and Consequences

- A chain of if-else statements, which we already know how to compile:
  - if(k==0)
    - f=i+j
  - else if(k==1)
    - f=g+h
  - else if(k==2)
    - f=g–h
  - else if(k==3)
    - f=i–j
- Use this mapping:
  - f: $s0, g: $s1, h: $s2, i: $s3, j: $s4, k: $s5

Multiple Condition and Consequences

- Final compiled MIPS code:

```
        bne $s5,$0,L1          # branch k!=0
        add $s0,$s3,$s4        #k==0 so f=i+j
        j Exit                 # end of case so Exit
L1:     addi $t0,$s5,-1        # $t0=k-1
        bne $t0,$0,L2          # branch k!=1
        add $s0,$s1,$s2        #k==1 so f=g+h
        j Exit                 # end of case so Exit
L2:     addi $t0,$s5,-2        # $t0=k-2
        bne $t0,$0,L3          # branch k!=2
        sub $s0,$s1,$s2        #k==2 so f=g-h
        j Exit                 # end of case so Exit
L3:     addi $t0,$s5,-3        # $t0=k-3
        bne $t0,$0,Exit        # branch k!=3
        sub $s0,$s3,$s4        #k==3 so f=i-j
Exit:
```

Loops (Repetition)

- Simple loop in C
  - do {
    - g = g + A[i];
    - i = i + j;

    } while (i != h);
- Rewrite this as:
  - Loop: g = g + A[i];
    - i = i + j;
    - if (i != h) goto Loop;
- Use this mapping:
  - g: $s1, h: $s2, i: $s3, j: $s4, base of A:$s5

Loops in C/Assembly

- Final compiled MIPS code:

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

Bitwise Operations

- A bitwise operation is where a logical operation is performed on the bits of each column of the operands. Here is the bitwise OR between two 8-bit patterns:

```
0110 1100    operand
0101 0110    operand
---------
0111 1110    result
```

OR Immediate Instruction

- ori d,s,const # register d <-- bitwise OR of const
  - # with the contents of register $s
  - # const is 16-bits, so
  - # 0x0000 ... const ... 0xFFFF
- The parts of the instruction must appear in the correct order, and const must be within the specified range. If the immediate operand in the assembly language shows less than 16 bits (as does 0x2 in the previous example) then the assembler expands it to the required sixteen bits. If the assembly language specifies more than sixteen bits, then the assembler writes an error message.
- The const part of the assembly language instruction can be a positive decimal or a hexadecimal constant. The assembler translates the constant into a 16-bit pattern in the machine instruction. For example, the following two assembly language instructions translate into the same machine language instruction:
  - ori $5,$4,0x10
  - ori $5,$4,16

OR operations

- Look at the instruction:
  - ori $8,$0,0x2
- Sixteen bits of immediate operand 0000 0000 0000 0010 are to be bitwise ORed with the thirty-two bits of register zero
  - 0000 0000 0000 0000 0000 0000 0000 0000
- This would not ordinarily be possible because the operands are different lengths. However, MIPS zero extends the sixteen-bit operand so the operands are the same length. Sometimes this is called padding with zeros on the left.
- zero extension

```
0000 0000 0000 0000 0000 0000 0000 0010   --zero extended
0000 0000 0000 0000 0000 0000 0000 0000   --data in register $0
---------------------------------------
0000 0000 0000 0000 0000 0000 0000 0010   --result, register $8
```

- An OR operation is done in each column. The 32-bit result is placed in register $8.

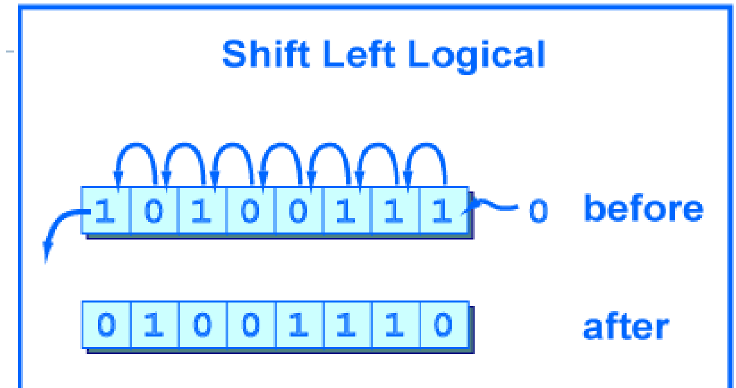
AND Immediate Instruction

- andi d,s,const # register d <-- bitwise AND of immediate const
  - # and the contents of register $s.
  - # const is a 16-bit pattern, so
  - # 0x0000 ... const ... 0xFFFF
- The andi instruction does a bitwise AND of two 32-bit patterns. At run time the 16-bit immediate operand is padded on the left with zero bits to make it a 32-bit operand.
- The three operands of the instruction must appear in the correct order, and const must be within the specified range. The immediate operand in the source instruction always specifies sixteen bits although the zeros on the left can be omitted (such as 0x2).

Exclusive Or Immediate

- xori d,s,const # register d <-- bitwise XOR of immediate const
  - # and the contents of register $s.
  - # const is a 16-bit pattern, so
  - # 0x0000 ... const ... 0xFFFF
- The three operands of the instruction must appear in the correct order, and const must be within the specified range. If the immediate operand in the assembly program is less than sixteen bits (such as 0x2) the assembler expands it to sixteen. If it is more than sixteen bits the assembler writes an error message.

Shift Left Logical



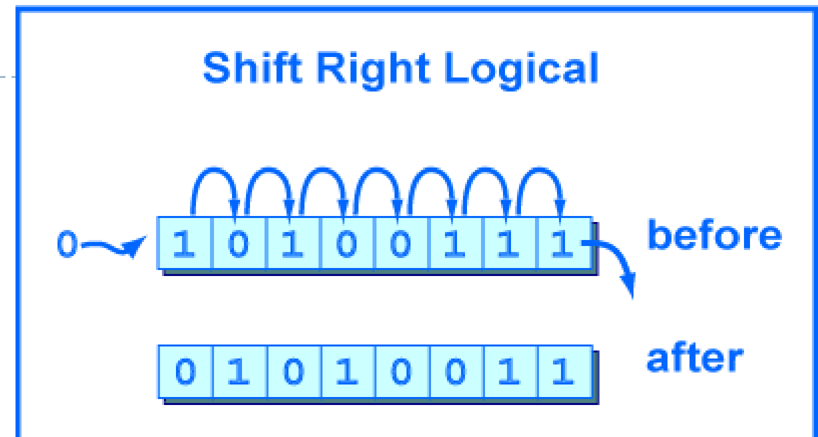
- A shift left logical of one position moves each bit to the left by one. The low-order bit (the right-most bit) is replaced by a zero and the high-order bit (the left-most bit) is discarded.
- Shifting by two positions is the same as performing a one-position shift two times. Shifting by zero positions leaves the pattern unchanged. Shifting an N-bit pattern left by N or more positions changes all of the bits to zero.
- sll d,s,shft # $d <-- the bits in $s shifted left logical
- # by shft positions,
- # where 0 <= shft < 32
- The ALU (arithmetic/logic unit) which does the operation pays no attention to what the bits mean. If the bits represent an unsigned integer, then a left shift is equivalent to multiplying the integer by two.

Shift Left Example

- Here is an 8-bit pattern (0110 1111)
- Shift it left (logical) by two.
- Code
  - ori $8, $0, 0x6F   # put bit pattern into register $8
  - sll $9, $8, 2   # shift left logical by two

Shift Right Logical

- MIPS also has a shift right logical instruction. It moves bits to the right by a number of positions less than 32. The high- order bit gets zeros and the low- order bits are discarded.



- If the bit pattern is regarded as an unsigned integer, or a positive two's comp. integer, then a right shift of one bit position performs an integer divide by two. A right shift by N positions performs an integer divide by 2N.
- The "trick" of dividing an integer by shifting should not be used in place of the MIPS arithmetic divide instruction (which will be covered in a few chapters). If you mean "divide" that is what you should write. But the trick is often used in hardware, and sometimes pops up in odd software uses, so you should know about it.
- srl d,s,shft   # $d <-- logical right shift of $s by shft positions.
-                         # shft is a 5-bit integer, 0 <= shft < 32

AND, OR, XOR and NOR Instructions

- or d,s,t   # $d <-- bitwise OR between $s with $t.
- and d,s,t   # $d <-- bitwise AND between $s with $t.
- xor d,s,t   # $d <-- bitwise XOR
-                        # between $s with $t.
- nor d,s,t   # $d <-- bitwise NOR
-                        # between $s with $t.

NOT operation

- NOT operation is done by using the NOR instruction with $0 as one of the operands:
  - nor d,s,$0   # $d <-- bitwise NOT of $s.

MOVE as OR with Zero

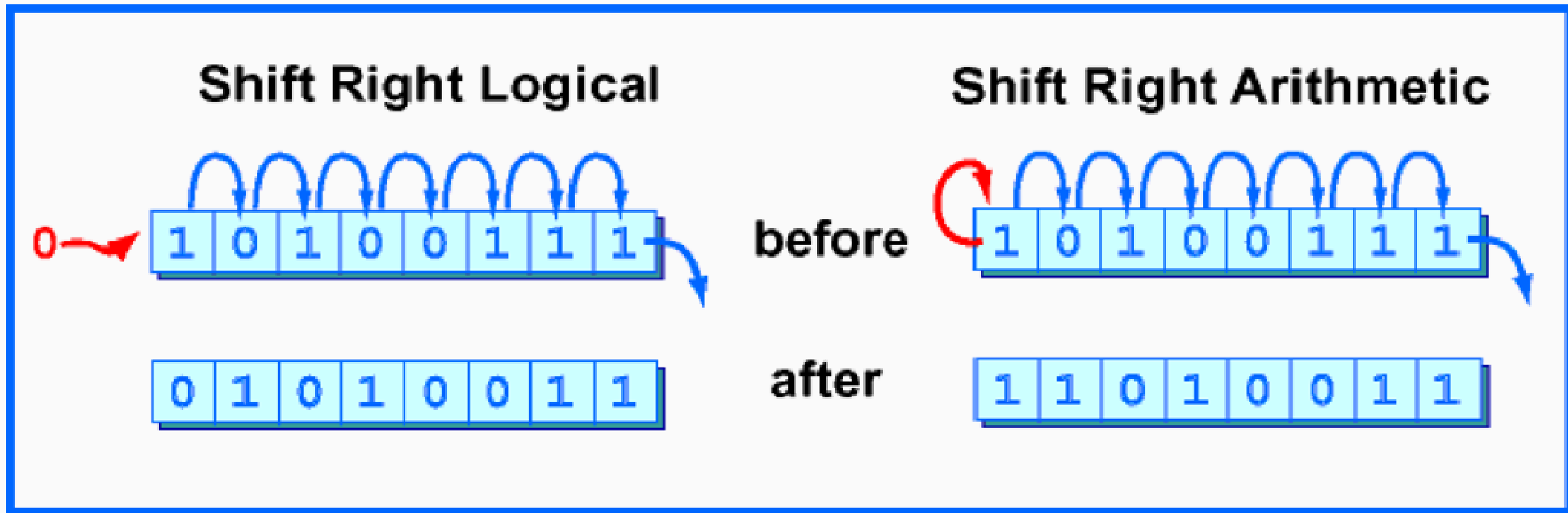
- Copying the pattern in a source register to a destination register is called a move operation, even though the source register does not change.
  - `or  d,s,$0        # $d <-- contents of $s.`

Program Logical Operations

- Start out a program with the instruction that puts a single one-bit into register one:
  - ori $1,$0,0x01
- Now, by using only shift instructions and register to register logic instructions, put the pattern 0xFFFFFFFF into register $1. Don't use another andi, ori or xori instruction. You will need to use more registers than $1. See how few instructions you can do this in. My program has 11 instructions.

Shift Right Arithmetic

- A right shift logical with two's complement negative integers does not work as division by two. The problem is that a shift right logical moves zeros into the high order bit. This is correct in some situations, but not for dividing two's complement negative integers. An arithmetic right shift replicates the sign bit as needed to fill bit positions:



The sra Instruction

- MIPS has a shift right arithmetic instruction:
  - sra   d,s,shft   # $d <-- s shifted right
  - # shft bit positions.
  - # 0 =< shft < 31

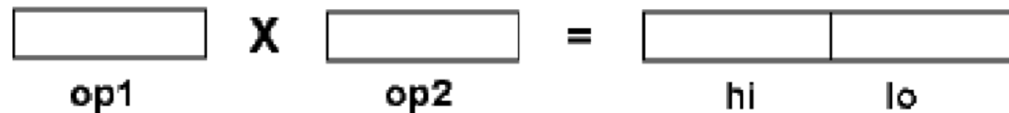
- Sometimes you need to divide by two. This instruction is faster and more convenient than the div instruction.

Multiplication

- The product of two N-place decimal integers may need 2N places. This is true for numbers expressed in any base. In particular, the product of two integers expressed with N- bit binary may need 2N bits. For example, here, two 8-bit unsigned integers are multiplied using the usual paper-and-pencil multiplication algorithm (but using binary arithmetic)
- The two 8-bit operands result in a 15-bit product. Also shown is the same product done with base 16 and base 10 notation.

```
       10110111   B7      18310
       10100010   A2      16210
       --------   --      -----
       00000000
      10110111.
     00000000..
    00000000...
   00000000....
  10110111.....
 00000000......
10110111.......
---------------   ----  -------
111001111001110   73CE  2964610
```

Multiplication Operation



- The multiply unit of MIPS contains two 32-bit registers called hi and lo. These are not general purpose registers. When two 32-bit operands are multiplied, hi and lo hold the 64 bits of the result. Bits 32 through 63 are in hi and bits 0 through 31 are in lo.
- Here are the instructions that do this. The operands are contained in general-purpose registers.
  - mult s,t # hilo <-- $s * $t. two's comp operands
  - multu s,t # hilo <-- $s * $t. unsigned operands
- Note: with add and addu, the operation carried out is the same with both instructions. The "u" means "don't trap overflow". With mult and multu, different operations are carried out. Neither instruction every causes a trap.

The mfhi and mflo Instructions

- There are two instructions that move the result of a multiplication into a general purpose register:
  - mfhi d # d <-- hi. Move From Hi
  - mflo d # d <-- lo. Move From Lo
- The hi and lo registers cannot be used with any of the other arithmetic or logic instructions. If you want to do something with a product, it must first be moved to a general purpose register. However there is a further complication on MIPS hardware:
  - Rule: Do not use a multiply or a divide instruction within two instructions after mflo or mfhi. The reason for this involves the way the MIPS pipeline works. On the SPIM simulator this rule does not matter.

Example

- Let us write a program that evaluates the formula 5 * x - 74 where the value x is in register $8. Assume that x is two's complement. Here is the program:

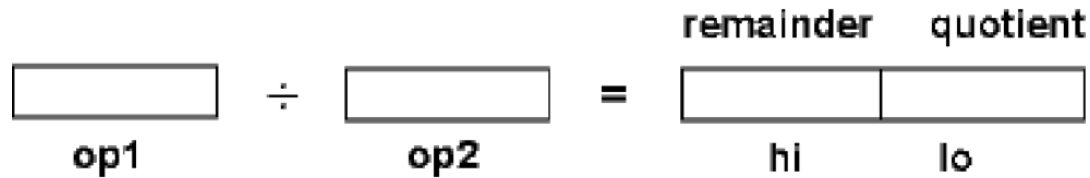
```
# newMult.asm
#
# Program to calculate 5 × x - 74
#
# Register Use:
#  $8   x
#  $9   result

        .text
        .globl  main

main:   ori     $8,   $0, 12        # put x into $8
        ori     $___, $0,  5        # put 5 into $___
        mult    $___, $___          #  ___ <--5x
        mflo    $___                # $___ = 5x
        addiu   $___, $___,-74      # $___ = 5x -74

## End of file
```

Solution to Example

- Here is the completed program. Only one additional register is needed. Register $9 is used to accumulate the result in several steps.

```
# newMult.asm
#
# Program to calculate 5 × x - 74
#
# Register Use:
#  $8   x
#  $9   result

        .text
        .globl  main

main:   ori      $8,   $0, 12         # put x into $8
        ori      $9,   $0,  5         # put 5 into $9
        mult     $9,   $8             # lo<--5x
        mflo     $9                   # $9 = 5x
        addiu    $9,   $9,-74         # $9 = 5x -74

## End of file
```

The div and the divu Instructions



- With N-digit integer division there are two results, an N-digit quotient and an N-digit remainder. With 32-bit operands there will be (in general) two 32-bit results. MIPS uses the hi and lo registers for the results:
- Here are the MIPS instructions for integer divide. The "u" means operands and results are in unsigned binary.
  - div s,t # lo <-- s div t
    - # hi <-- s mod t
    - # two's complement
  - divu s,t # lo <-- s div t
    - # hi <-- s mod t
    - # unsigned

Example

- For this example say that we wish to calculate (y + x) / (y - x). The argument x is in $8; y is in $9. The quotient is to be placed in $10 and the remainder in $11.

```
## divEg.asm
##
## Program to calculate (y + x) / (y - x)
##
## Register Use:
##  $8   x
##  $9   y
##  $10  x/y
##  $11  x%y

              .text
              .globl  main

main:         ___      $8,   $0,  8         # put x into $8
              ___      $9,   $0, 36         # put y into $9
              addu     $10,  $__, $__       # $10  <-- (y+x)
              subu     $11,  $__, $__       # $11  <-- (y-x)
              div      $__,  $__            # hilo <--(y+x)/(y-x)
              ____     $10                  # $10  <--quotient
              ____     $11                  # $11  <-- remainder

## End of file
```

Example

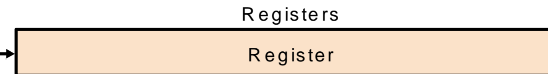
- Here is the complete program:

```
## divEg.asm
##
## Program to calculate (y + x) / (y - x)
##
## Register Use:
##  $8   x
##  $9   y
##  $10  x/y
##  $11  x%y

            .text
            .globl  main

main:    ori      $8,   $0,  8          # put x into $8
                ori      $9,   $0, 36          # put y into $9
                addu     $10,  $9, $8          # $10  <-- (y+x)
                subu     $11,  $9, $8          # $11  <-- (y-x)
                div      $10,  $11             # hilo <--(y+x)/(y-x)
                mflo     $10                   # $10  <--quotient
                mfhi     $11                   # $11  <-- remainder

## End of file
```

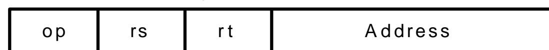
Addressing mode summary

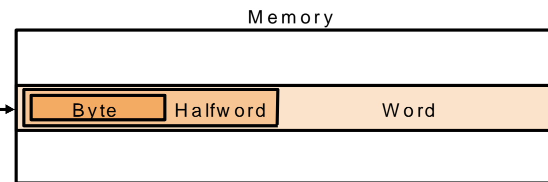
1. Immediate addressing

2. Register addressing

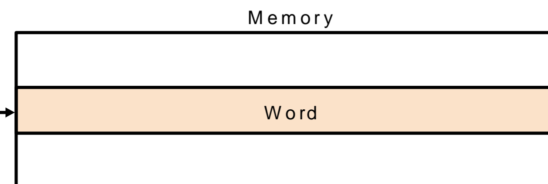


3. Base addressing

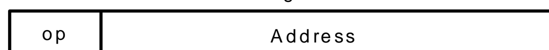


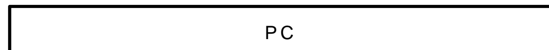


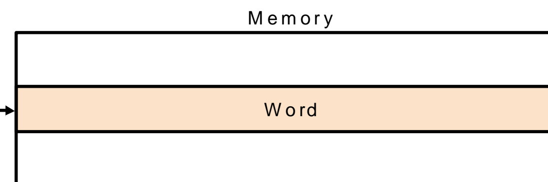
4. PC-relative addressing



5. Pseudodirect addressing







3 examples