CS 281

# Week 1

## Lecture 1

### Compilation Process

* C program 🡪 Compiler 🡪 Assembly language program 🡪 Assembler 🡪 Object: Machine language module 🡪 Linker 🡪 Executable: Machine language program 🡪 Loader 🡪 Memory ^

Object: Library routine (machine language)

* Source file 🡪 Assembler 🡪 Object file 🡪

Source file 🡪 Assembler 🡪 Object file 🡪 Linker 🡪 Executable file

Source file 🡪 Assembler 🡪 Object file 🡪 ^

Program library

### Below Your Program

* Unix example
* Source Files: count.c and main.c
* Corresponding assembly code: count.s and main.s
* Corresponding machine code (object code): count.o and main.o
* Library functions: libc.a
* Executable file: a.out
* Format for a.out and object code: ELF (Executable and Linking Format)

### Producing an Executable Program

* Unix example
* Compiler: count.c and main.c 🡪 count.s and main.s
* gcc –S count.c main.c
* ;dfk
* Assembler: count.s and main.s 🡪 count.o and main.o
* gcc –c count.s main.s
* as count.s -o count.o
* kdf;jdfk
* Linker/Loader: count.o main.o libc.a 🡪 a.out
* gcc main.o count.o
* ld main.o count.o -lc (additional libraries are required)

### Source Files

void main()

{

int n,s;

printf("Enter upper limit: ");

scanf("%d",&n);

s = count(n);

printf("Sum of i from 1 to %d = %d\n",n,s);

}

int count(int n)

{

int i,s;

s = 0;

for (i=1;i<=n;i++)

s = s + i;

return s;

}

### Assembly Code for MIPS (count.s)

#.file 1 "count.c"

.option pic2

.section .text

.text

.align 2

.globl count

.ent count

count:

.LFB1:

.frame $fp,48,$31 # vars= 16, regs= 2/0, args= 0, extra= 1

6

.mask 0x50000000,-8

.fmask 0x00000000,0

subu $sp,$sp,48

.LCFI0:

sd $fp,40($sp)

.LCFI1:

sd $28,32($sp)

.LCFI2:

move $fp,$sp

.LCFI3:

.set noat

lui $1,%hi(%neg(%gp\_rel(count)))

addiu $1,$1,%lo(%neg(%gp\_rel(count)))

daddu $gp,$1,$25

.set at

sw $4,16($fp)

sw $0,24($fp)

li $2,1 # 0x1

sw $2,20($fp)

.L3:

lw $2,20($fp)

lw $3,16($fp)

slt $2,$3,$2

beq $2,$0,.L6

b .L4

L6:

lw $2,24($fp)

lw $3,20($fp)

addu $2,$2,$3

sw $2,24($fp)

.L5:

lw $2,20($fp)

addu $3,$2,1

sw $3,20($fp)

b .L3

.L4:

lw $3,24($fp)

move $2,$3

b .L2

.L2:

move $sp,$fp

ld $fp,40($sp)

ld $28,32($sp)

addu $sp,$sp,48

j $31

.LFE1:

.end count

### Assembly Characteristics: Data Types

* “Integer” data of 1, 2, or 4 bytes
* Data values
* Addressed (untyped pointers)
* Floating point data of 4, 8, or 10 bytes
* No aggregate types such as arrays or structures
* Just contiguously allocate bytes in memory

### Assembly Characteristics: Operations

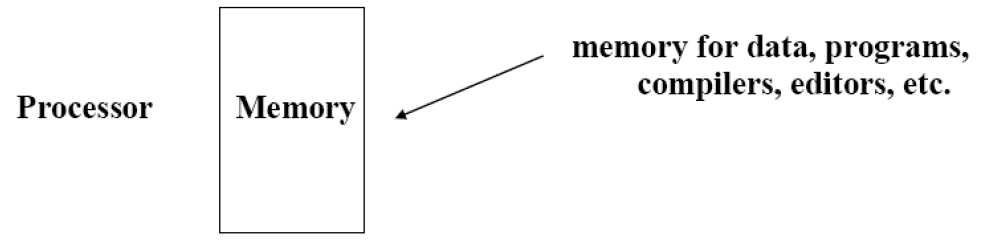
* Perform arithmetic function on register or memory data
* Transfer data between memory and register
* Load data from memory into register
* Store register data into memory
* Transfer control
* Unconditional jumps to/from procedures
* Conditional branches

### MIPS Instruction Set

* Objective: to introduce the MIPS instruction set and to show how MIPS instructions are represented in the computer
* The stored-program concept
* Instructions are represented as numbers
* Programs can be stored in memory to be read or written just like data

### Stored Program Concept

* Instructions are just a sequence of 32 bits
* Programs are stored in memory to be read or written just like data



* Fetch & Execute Cycle
* Instruction is fetched and put into a special register
* Bits in the instruction register determine the subsequent actions
* When done, fetch the next instruction and continue execution

### MIPS Arithmetic

* All arithmetic instructions have 3 operands
* Operand order is fixed (destination first)
* Example
* C code: A = B + C
* MIPS code: add $s0, $s1, $s2 (associated with variables by computer)
* Using the natural number of operands for an operation (e.g. addition) conforms to the design principle of keeping the hardware simple

### Temporary Variables

* Regularity of instruction format requires that expression get mapped to a sequence of binary operations with temporary results being stored in temporary variables
* Example
* C code: f = (g + h) - (i + j)
* MIPS code (assume f, g, h, i, j are in $s0 through $s4 respectively):

add $t0, $s1, $s2 # $t0 = g+h

add $t1, $s3, $s4 # $t1 = i+j

sub $s0, $t0, $t1 # f = $t0 - $t1

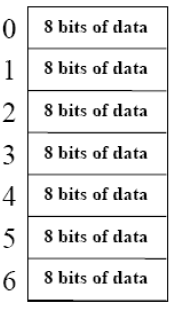
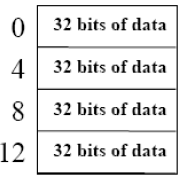
### Registers vs. Memory

* Operands for arithmetic instructions must be registers, only 32 integer registers are available
* Compiler associates variables with registers
* When too many variables are used to fit in 32 registers, the compiler must allocate temporary space in memory and then load and store temporary results to/from memory
* The compiler tries to put most frequently occurring variables in register
* The extra temporary variables must be “spilled” to memory

### Memory Operands

* Main memory used for composite data
* Arrays, structures, dynamic data
* To apply arithmetic operations
* Load values from memory into registers
* Store results from register to memory
* Memory is byte addressed
* Each address identifies an 8-bit byte
* Words are aligned in memory
* Address must be a multiple of 4
* MIPS is Big Endian
* Most-significant byte at least address of a word
* c.f. Little Endian: least-significant byte at least address

### Memory Organization

* Viewed as large, single-dimension array, where a memory address is an index to the array
* MIPS systems address memory in byte chunks
* The memory address (= index) points to a byte in memory
* This is called “Byte addressing”
* Most data items are grouped into words
* A MIPS word is 4 bytes or 32 bits
* Registers also hold 32 bits of data
* 232 bytes with byte addresses from

0 to 232 - 1

* 230 words with byte addresses

0, 4, 8, … 232 - 4

* Words are aligned (alignment restriction)
* Bytes can be accessed from left to right (big endian) or right to left (little endian)
* Big Endian: most-significant byte at least address of a word, accessed from left to right
* Little Endian: least-significant byte at least address of a word, accessed from right to left

### Load and Store

* All arithmetic instructions operate on registers
* Memory is accessed through load and store instructions
* Example C Code: A[12] = h + A[8]
* MIPS code (assume that $s3 contains the base address of A):

lw $t0, 32($s3) # 4 \* 8 = 32

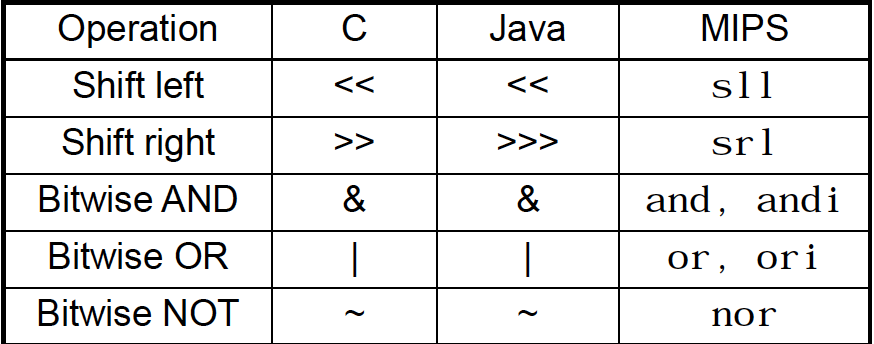
add $t0, $t0, $s2

sw $t0, 48($s3) # 4 \* 12 = 48

* Note: sw (store word instruction) has designation last
* Note: remember arithmetic operands are registers, not memory
* This is invalid: add 48($s3), $s2, 32($s3)
* Example and Meaning
* lw $s1, 100($s2): $s1 = Memory[$s2+100]
* sw $s1, 100($s2): Memory[$s2+100] = $s1

### Logical Operations

* Instructions for bitwise manipulation



* Useful for extracting and inserting groups of bits in a word

### Machine Language

* Instructions, registers, and data words are 32 bits long
* Example: add $t1, $s1, $s2
* Registers have numbers/indices: $t1=9, $s1=17, $s2=18
* Instruction format
* 000000: op
* 10001: rs
* 10010: rt
* 01001: rd
* 00000: shamt
* 100000: funct

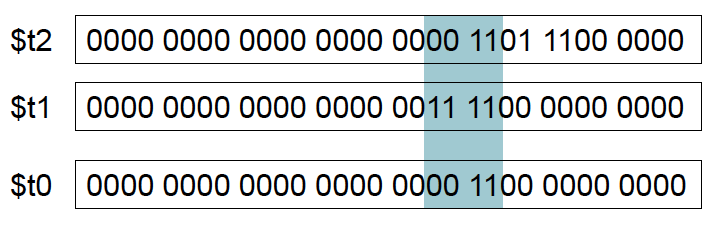
### ../../../../../../../Pictures/Screenshots/Screen%20Shot%202018-04-04%20at%Register Names

### Shift Operations

* op: 6 bits
* rs: 5 bits
* rt: 5 bits
* rd: 5 bits
* shamt: 5 bits
* funct: 6 bits
* shamt: how many positions to shift
* Shift left logical
* Shift left and fill with 0 bits
* sll by i bits multiplies by 2i
* Shift right logical
* Shift right and fill with 0 bits
* srl by i bits divides by 2i (unsigned only)

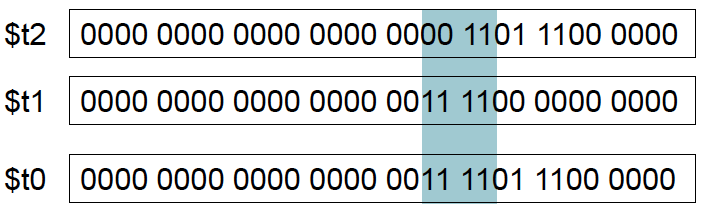
### AND Operations

* Useful to mask bits in a word
* Select some bits, clear others to 0
* and $t0, $t1, $t2:



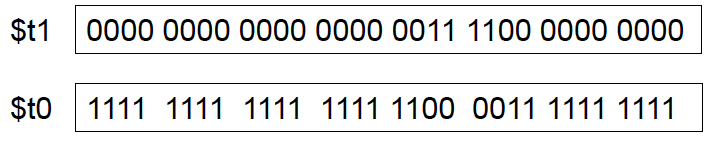
### OR Operations

* Useful to include bits in a word
* Set some bits to 1, leave others unchanged
* or $t0, $t1, $t2

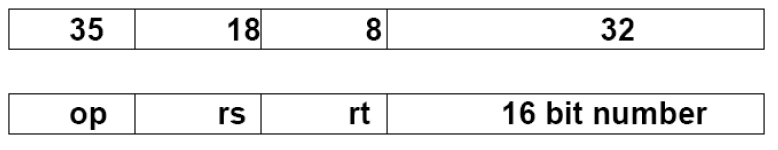


### NOT Operations

* Useful to invert bits in a word
* Changes 0 to 1, and 1 to 0
* MIPS has NOR 3-operand instruction
* a NOR b == NOT ( a OR b )
* nor $t0, $t1, $zero
* Register 0: always read as zero



### Machine Language Cont’d.

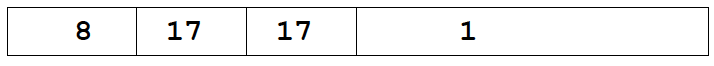
* Consider the load-word and store-word instructions
* What would the regularity principle have us do?
* New principle: Good design demands a compromise
* Introduce a new type of instruction format
* I-format type for data transfer instructions
* The other format was R-type for register (add and sub)
* Example: lw $t0, 32($s2)

### The Constant Zero

* MIPS register 0 ($zero) is the constant 0
* Cannot be overwritten
* Useful for common operations
* e.g move between registers
* add $t2, $s1, $zero

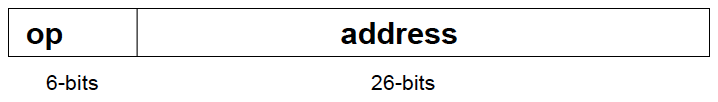
### Immediate Instructions

* Immediate mode includes small constants in instructions
* Avoid extra memory operations
* Example: addi $s1, $s1, 1

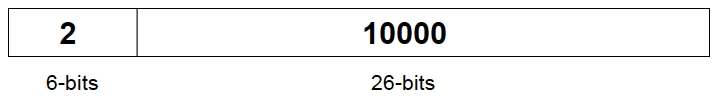


### Addressing in Jumps

* J format format (jump format – j, jal)



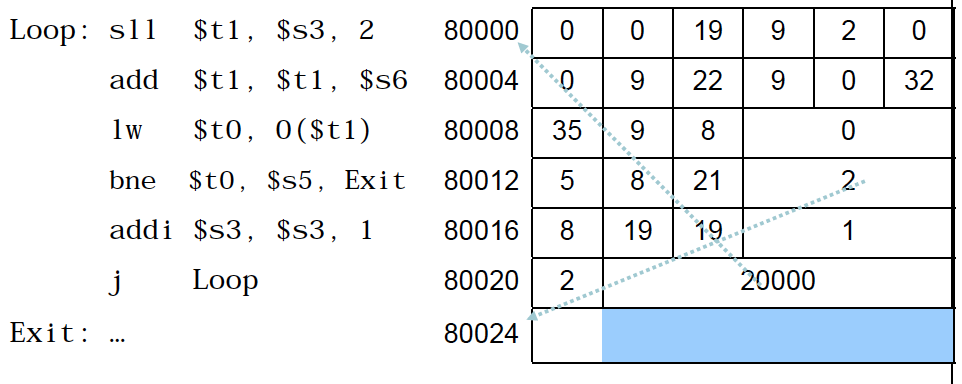
* Example: j 10000



* Addresses are 32 bits, but we only have 26 bits to place an address
* 232 / 226 = 64, which means we can only access 1/64th of memory
* The rightmost two bits are zeroes, which we can ignore
* We can shift left twice to get two more bits
* Program counter keeps track of where the next instruction is
* Gives us 28 bits
* 232 / 228 = 16, which means we can access 1/16th of memory
* The left 4 bits of the program counter are left untouched
* So, you can only jump within that 16th of memory
* To get around this, stuff the 32 bits into a register half-by-half (like we did in HW W1A Problem 5)
* Branch: Take number, multiply by 4, sign extend it, and add PC
* PC = PC + Sign Extension (d \* 4)

### Target Addressing Example

* Loop code example
* Assume Loop at location 80000



### Design Principle

* Simplicity favors regularity
* All instructions 32 bits
* All instructions have 3 operands
* Smaller is faster
* Only 32 registers
* Good design demands good compromises
* All instructions are the same length
* Limited number of instruction formats: R, I, J
* Make common cases fast
* 16-bit immediate constant
* Only two branch instructions

### Conditional Operations

* Branch to a labeled instruction if a condition is true
* Otherwise, continue sequentially
* beq rs, rt, L1
* if (rs == rt) branch to instruction labelled L1
* bne rs, rt, L1
* if (rs != rt) branch to instruction labelled L1
* j L1
* unconditional jump to instruction labelled L1

### Control

* MIPS unconditional branch instructions
* j label
* Example
* if (i!=j)

h=i+j;

else

h=i-j;

* beq $s4, $s5, Equal

add $s3, $s4, $s5

j GoOn

Equal: sub $s3, $s4, $s5

GoOn: ...

### Compiling If Statements

* C Code
* if ( i==j )

f = g+h;

else

f = g-h;

* Compiled MIPS Code
* bne $s3, $s4, Else

add $s0, $s1, $s2

j Exit

Else: sub $s0, $s1, $s2

Exit: … (assembler calculates address)

### Compiling Loop Statements

* C Code
* while (save[i] == k)

i += 1;

* i in $s3, k in $s5, address of save in $s6
* Compiled MIPS Code
* Loop: sll $t1, $s3, 2

add $t1, $t1, $s6

lw $t0, 0($t1)

bne $t0, $s5, Exit

addi $s3, $s3, 1

j Loop

Exit: …

### Branch Instruction Design

* Why not blt, bge, etc?
* Hardware for <, ≥, … slower than =, ≠
* Combining with branch involves more work per instruction, requiring a slower clock
* All instructions penalized
* beq and bne are the common case
* This is a good design compromise

### Less Than Test

* What about Branch-if-less-than?
* New instruction
* if $s1 < $s2 then

$t0 = 1

slt $t0, $s1, $s2

else

$t0 = 0

* MIPS does not include blt, bgt, ble, bge, bgz, etc. instructions because it is considered too complicated
* Assembler pseduoinstructions
* blt $s1, $s2, Label
* Note that the assembler needs a register to do this
* There are important conventions for register use

### More Conditional Operations

* Set result to 1 if a condition is true
* Otherwise, set to 0
* slt rd, rs, rt
* if (rs < rt) rd = 1; else rd = 0;
* slti rt, rs, constant
* if (rs < constant) rt = 1; else rt = 0;
* Use in combination with beq, bne
* slt $t0, $s1, $s2 # if ($s1 < $s2)
* bne $t0, $zero, L $ branch to L

### Quick Review

Instruction Meaning

add $s1,$s2,$s3 $s1 = $s2 + $s3

sub $s1,$s2,$s3 $s1 = $s2 – $s3

lw $s1,100($s2) $s1 = Memory[$s2+100]

sw $s1,100($s2) Memory[$s2+100] = $s1

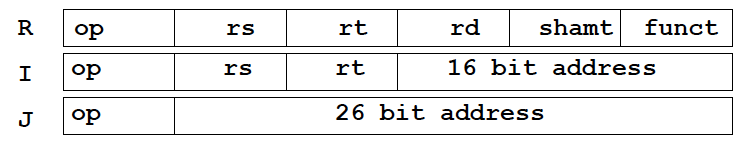
bne $s4,$s5,Label Jump to Label if $s4≠$s5

beq $s4,$s5,Label Jump to Label if $s4=$s5

j Label Next instr. is at Label

slt $t0, $s1, $s2 Set $t0 = 1 if $s1 < $s2, else set $t0 = 0

Formats



### Loops

* Compiling a while loop (assume i = $s3 and k = $s5 and that the base address of save is in $s6)
* while (save[i] == k)

i += 1;

* Loop: sll $t1, $s3, 2 # $t1 = 4 \* i

add $t1, $t1, $s6 # address of save[i]

lw $t0, 0($t1) # get save[i]

bne $t0, $s5, Exit # goto Exit if save[i] ≠ k

add $s3, $s3, 1 # i = i + 1

j Loop

Exit:

### Procedures

* In the execution of a procedure, the program must follow these steps
* Place parameters in a place where the procedure can access them
* Transfer control to the procedure
* Acquire the storage resources needed for the procedure
* Perform the desired task
* Place the result where the calling program can access it
* Return control to the point of origin

### Registers for Procedure Calling and the jal Instruction

* $a0 - $a3: four argument registers used to pass parameters
* $v0 - $v1: two value registers in which to return values
* $ra: one return address register to return to the point of origin
* jal Procedure Address: instruction to transfer control to a procedure and store the return address in $ra ($ra is set to PC + 4, address of the next instruction after procedure call)