University of Southern California

Viterbi School of Engineering

EE352
Computer Organization and Architecture

Assembly Directives
Control Flow (Branch Instructions)

References:

1) **Textbook**
2) **Mark Redekopp's slide series**

Shahin Nazarian

Spring 2010

Directives

Pseudo-instructions

ASSEMBLERS

Assembler Syntax

- In MARS and most assemblers each line of the assembly program may be one of three possible options
  - Comment
  - Instruction / Pseudo-instruction
  - Assembler Directive

Comments

- In MARS an entire line can be marked as a comment by starting it with a pound (#) character:
- Example:

```
# This line will be ignored by the assembler
        LW      $2,8($3)
        ADDI    $2,$2,1
        ...
```

Instructions

- **In MARS each instruction is written on a separate line and has the following syntax:**

  (Label:) Instruc. Op. Operands Comment

- **Example:**

  START: ADD $2,$3,$4 # R[2]=R[3] + R[4]

- **Notes:**
  - **Label is optional and is a text identifier for the address where the instruction is placed in memory. (These are normally used to identify the target of a branch or jump instruction.)**
  - **In MARS, a comment can be inserted after an instruction by using a '#' sign**
  - **A label can be on a line by itself in which case it refers to the address of the first instruction listed after it**

Labels and Instructions

- The optional label in front of an instruction evaluates to the address where the instruction starts in memory and can be used in other instructions

```
            .text
START:      LW          $4,8($10)
L1:         ADDI        $4,$4,-1
            BNE         $4,$0,L1
            J           START
```

Assembly Source File

LW	0x400000 = START
ADDI	0x400004 = L1
BNE	0x400008
J	0x40000C

Note: The BNE instruc. causes the program to branch (jump) to the instruction at the specified address if the two operands are Not Equal. The J(ump) instruction causes program execution to jump to the specified label (address)

Assembler finds what address each instruction starts at…

```
            .text
            LW      $4,8($10)
            ADDI    $4,$4,-1
            BNE     $4,$0,0x400004
            J       0x400000
```

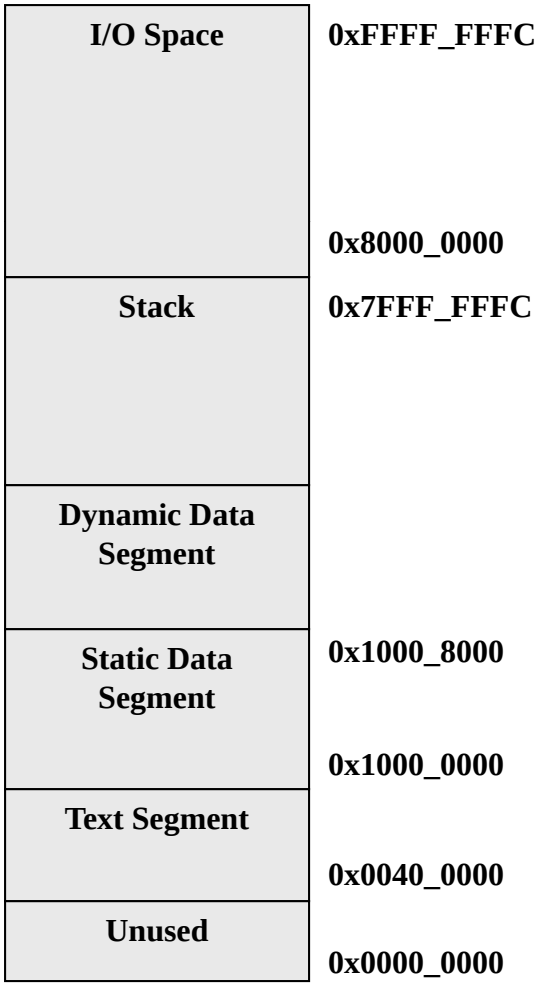
…and replaces the labels with their corresponding address

Assembler Directives

- Similar to pre-processor statements and global variable declarations in C/C++
  - Text and data segments
  - Reserving & initializing global variables and constants
  - Compiler and linker status
- Direct the assembler in how to assemble the actual instructions and how to initialize memory when the program is loaded

Text and Static Data Segments

- .text directive indicates the following instructions should be placed in the program area of memory
- .data directive indicates the following data declarations will be placed in the data memory segment



Static Data Directives

- Fills memory with specified data when program is loaded
- Format:

(*Label*:) .*type_id* *val_0,val_1,…,val_n*

- type_id = {.byte, .half, .word, .float, .double}
- Each value in the comma separated list will be stored using the indicated size
  - Example: myval: .word 1, 2, 0x0003
    - Each value 1, 2, 3 is stored as a word (i.e. 32-bits)
    - Label "myval" evaluates to the start address of the first word (i.e. of the value 1)

More Static Data Directives

- Can be used to initialize ASCII strings
- Format:

  (*Label*:) .ascii "*string*"

  (*Label*:) .asciiz "*string*"

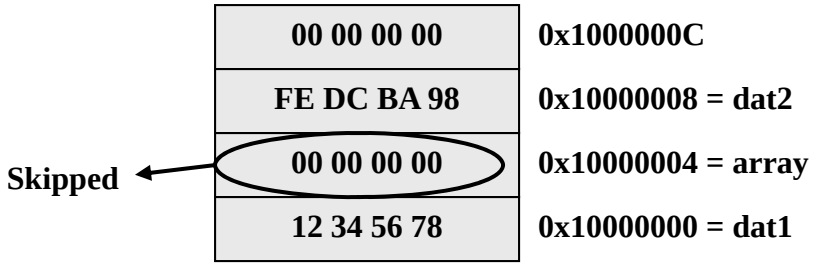
- .asciiz adds a null-termination character (0) at the end of the string while .ascii does not
- Example: string: .asciiz "Hello world\n"
  - Each character stored as a byte (including \n = Line Feed and \0 = Null (0) character)
  - Label "myval" evaluates to the start address of the first byte of the string

Reserving Memory

- **Reserves space in memory but leaves the contents unchanged**
- **Format:**

  (*Label*:) .space num_bytes

```
        .data
dat1:   .word  0x12345678
array:  .space 4
dat2:   .word  0xFEDCBA98
```



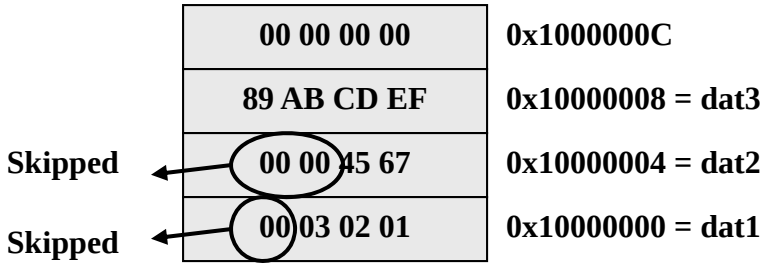
Alignment Directive

- Used to skip to the next, correctly-aligned address for the given data size
- Format:

  .align 0,1,2, or 3

- 0 = byte-, 1 = half-, 2 = word-, 3 = double-alignment

```
        .data
dat1:   .byte  1, 2, 3
        .align 1
dat2:   .half  0x4567
        .align 2
dat3:   .word  0x89ABCDEF
```



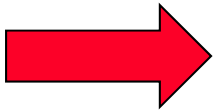
Note: The number after .align is not how many bytes to skip, it indicates what type of data will come next and thus the size to be aligned

Linker Directives

- .globl *label*
  - Allows the following label and data to be referenced from other compilation units (files)
- .extern *label size*
  - Defines an externally allocated (in another file) static data storage with *size* at address *label*

```
        .data
        .extern dat1 4
        .text
        .globl main
main:   la  $1,dat1
        ...
```

file1.s



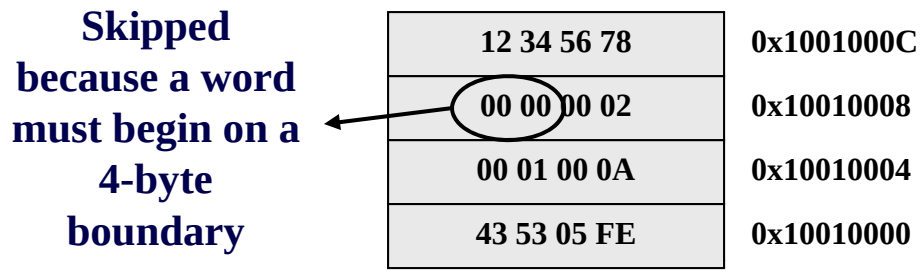
dat1 is defined in another file. Main is visible to other files

```
        .data
dat1:   .word 0x12345678
```

file2.s

.data example Examples

```
        .data
C1:     .byte       0xFE,0x05
MSG:    .asciiz     “SC\n”
DAT:    .half       1,2
        .align      2
VAR:    .word       0x12345678
```



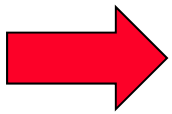
- C1 evaluates to 0x10001000
- MSG evaluates to 0x10001002 (Note: \n = Line Feed char. = 0x0A)
- DAT evaluates to 0x10001006
- VAR evaluates to 0x1000100C

C/C++ and Directives

- Directives are used to initialize or reserve space for global variables in C

```
short int count = 7;
char message[16];
int table[8] = {0,1,2,3,4,5,6,7};

void main()
{
  ...
}
```



```
          .data
count:    .half  7
message:  .space 16
          .align 2
table:    .word  0,1,2,3,4,5,6,7

          .text
          .globl main
main:     ...
```

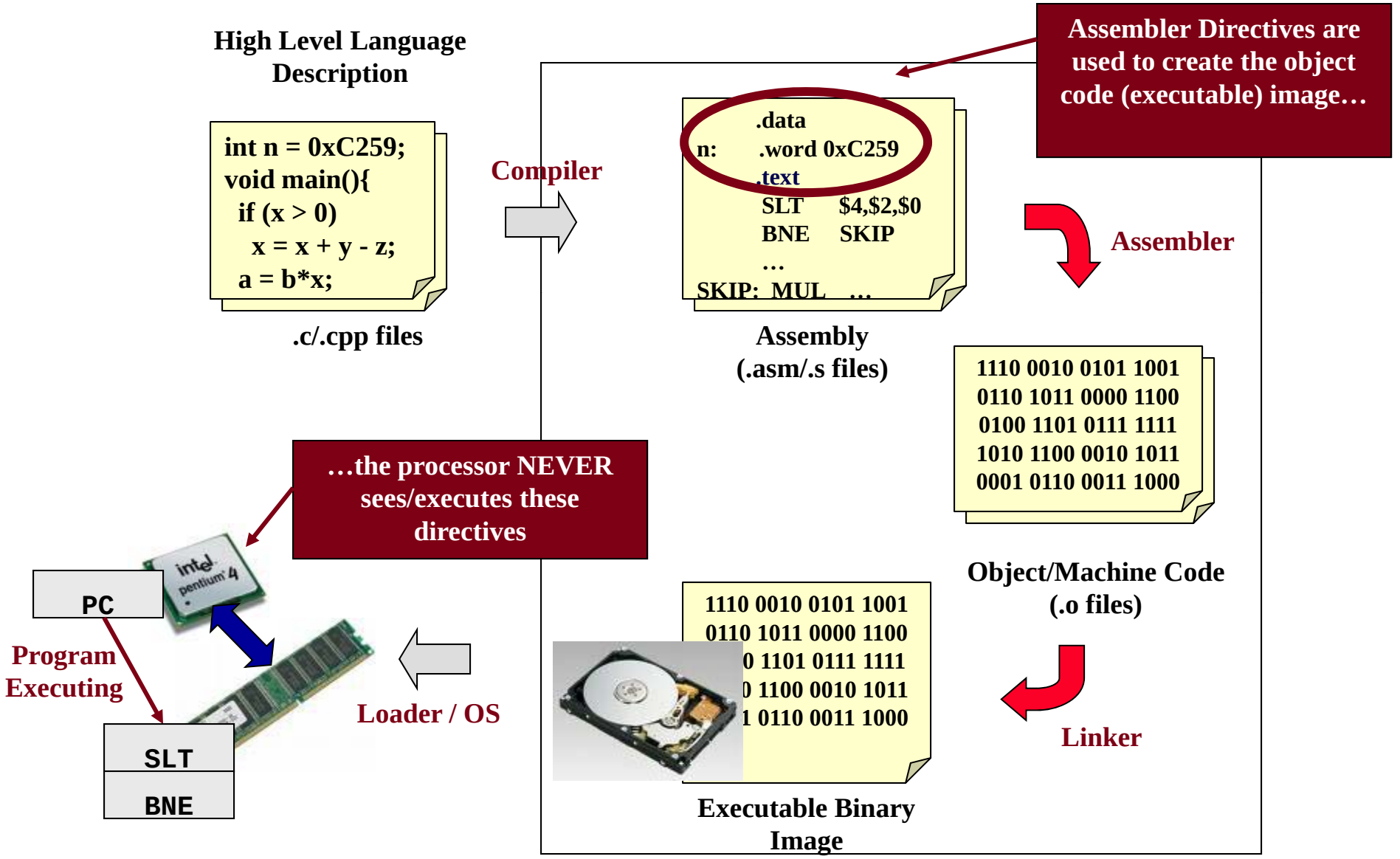
C/C++ style global declarations

Assembly equivalent

Summary & Notes

- Assembler Directives:
  - Tell the assembler how to build the program memory image
    - Where instructions and data should be placed in memory when the program is loaded
    - How to initialize certain global variables
- Recall, a compiler/assembler simply outputs a memory IMAGE of the program, which must then be loaded into memory by the OS to be executed
- Key: Directives are NOT instructions!
  - They are used by the assembler to create the memory image and then removed
  - The MIPS processor never sees these directives!

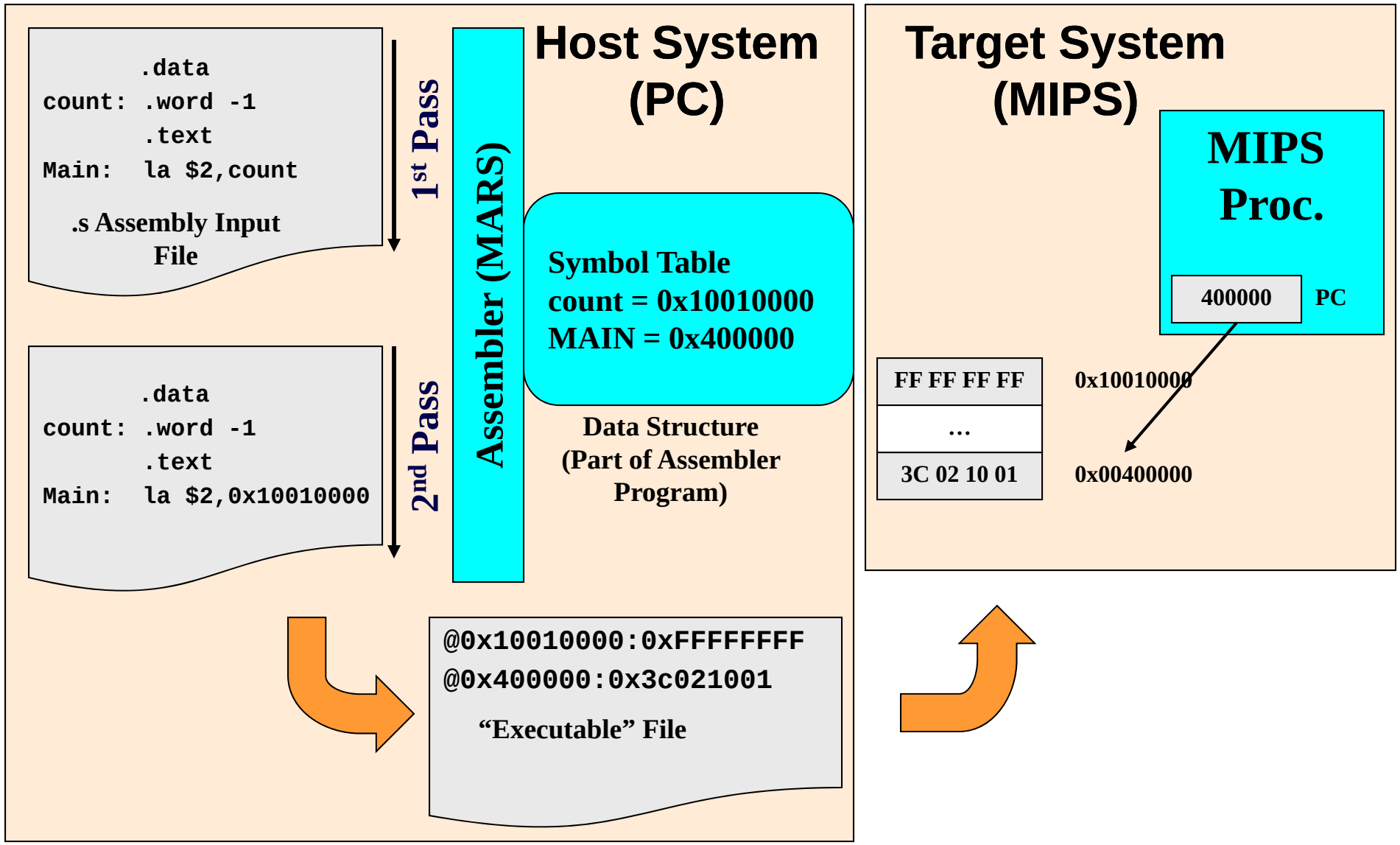
Directives in the Software Flow



Assembly Process

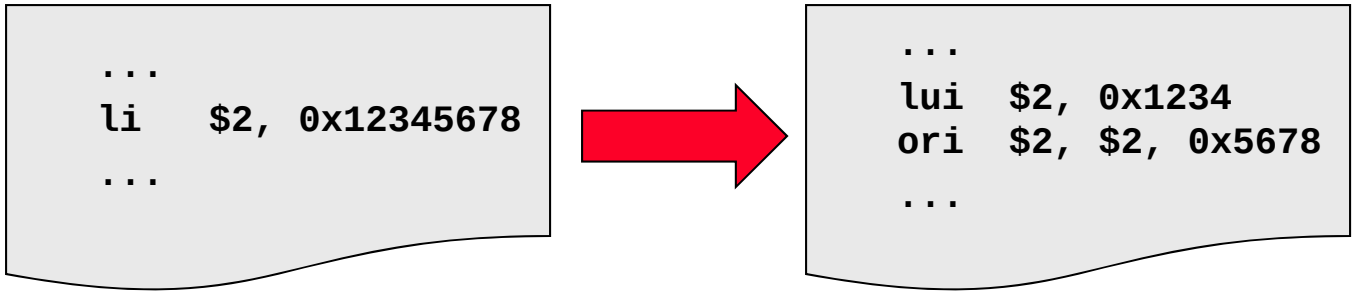
- The assembly procedure of a file requires two passes over the code
  - 1st Pass: Build a symbol table
    - Maps labels to corresponding addresses
  - 2nd Pass: Substitute corresponding values for labels and symbols and then translate to machine code

Assembly Process Diagram



Pseudo-instructions

- "Macros" translated by the assembler to instructions actually supported by the HW
- Simplifies writing code in assembly
- Example – LI (Load-immediate) pseudo-instruction translated by assembler to 2 instruction sequence (LUI & ORI)



With pseudo-instruction

After assembler…

Pseudo-instructions

Pseudo-instruction	Actual Assembly
NOT Rd,Rs	NOR Rd,Rs,$0
NEG Rd,Rs	SUB Rd,$0,Rs
LI Rt, immed. # Load Immediate	LUI Rt, {immediate[31:16], 16'b0} ORI Rt, {16'b0, immediate[15:0]}
LA Rt, label # Load Address	LUI Rt, {immediate[31:16], 16'b0} ORI Rt, {16'b0, immediate[15:0]}
BLT Rs,Rt,Label	SLT $1,Rs,Rt BNE $1,$0,Label

- Note: Pseudo-instructions are assembler-dependent
- See MARS Help for more details

Support for Pseudo-instructions

- Pseudo-instructions often expand to several instructions and there is a need for usage of a temporary register
- Assembler reserves R[1] ($1)
  - In the assembler, $1 = $at (assembler temp.)
- You can use $1 but it will be overwritten when you use certain pseudo-instructions

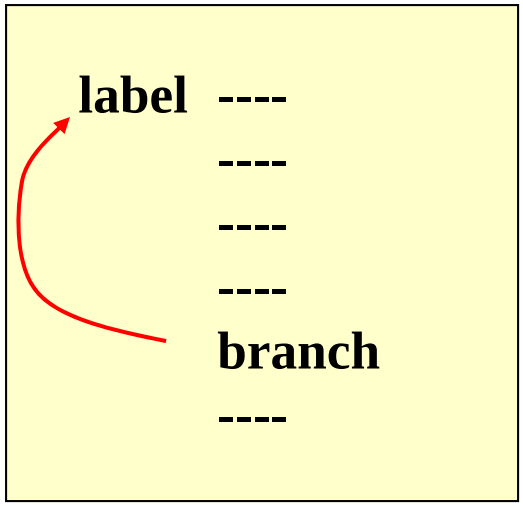
Branch Instructions

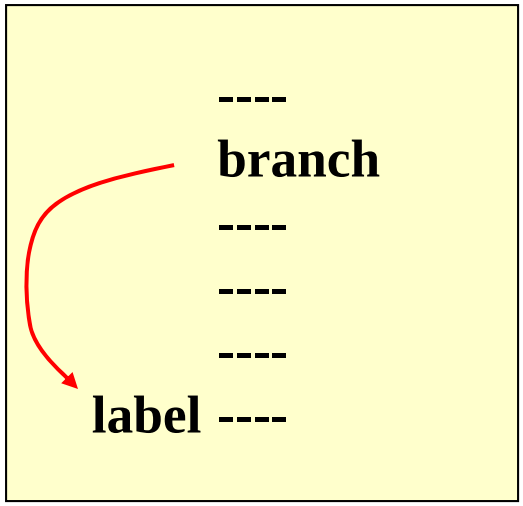
Loops and Conditionals

CONTROL FLOW

Branch Instructions

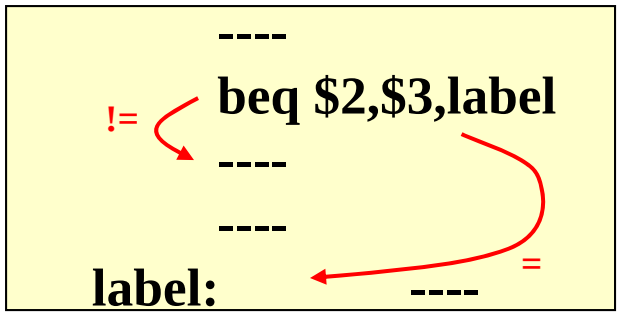
- Operation: PC = PC + displacement
- Branches allow us to jump backward or forward in our code

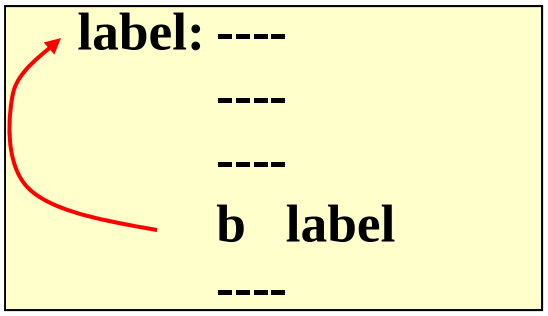




Branch Instructions

- **Conditional Branches**
  - Branches only if a particular condition is true
  - Fundamental Instrucs.: BEQ (if equal), BNE (not equal)
  - Syntax: `BNE/BEQ Rs, Rt, label`
    - Compares Rs, Rt and if EQ/NE, branch to label, else continue
- **Unconditional Branches**
  - Always branches to a new location in the code
  - Instruction: `BEQ $0,$0,label`
  - Pseudo-instruction: `B label`





Single-Operand Compare & Branches

- For <, >, etc. comparison, actual MIPS conditional branch instructions can only compare one operand with zero
- Syntax: BccZ Rt, label
  - cc = {LT, LE, GT, GE}

Branch Instruc.	Branch if…
BLTZ $2,label	$2 < 0
BLEZ $2,label	$2 ≤ 0
BGTZ $2,label	$2 > 0
BGEZ $2,label	$2 ≥ 0

Two-Operand Compare & Branches

- Two-operand comparison is accomplished using the SLT/SLTI/SLTU (Set If Less-than) instruction
  - Syntax: SLT Rd,Rs,Rt or SLT Rd,Rs,imm
    - If Rs < Rt then Rd = 1, else Rd = 0
  - Use appropriate BNE/BEQ instruction to infer relationship

Branch if…	SLT	BNE/BEQ
$2 < $3	SLT $1,$2,$3	BNE $1,$0,label
$2 ≤ $3	SLT $1,$3,$2	BEQ $1,$0,label
$2 > $3	SLT $1,$3,$2	BNE $1,$0,label
$2 ≥ $3	SLT $1,$2,$3	BEQ $1,$0,label

Branch Pseudo-Instructions

Pseudo-instruction	Description
BLT Rt, Rs, label	Branch if less-than
BLE Rt, Rs, label	Branch if less-than or equal
BGT Rt, Rs, label	Branch if greater-than
BGE Rt, Rs, label	Branch if greater-than of equal
BLTU Rt, Rs, label	Branch if less-than (unsigned)
BLT Rt, imm, label	Branch if less-than immediate

Note: Pseudoinstructions are assembler-dependent. See MARS Help for more details

Comparison with SLT

- Performing comparison with the SLT instruction is really accomplished by subtracting A-B and examining the sign of the result
  - if A-B = 0, then A=B
  - if A-B = negative #, then A<B
  - If A-B = positive # and not 0, then A>B
- Determining if the result is positive or negative requires
  - knowing what system is being used
    - signed or unsigned?
  - if overflow occurred
    - when overflow occurs the sign of the result is incorrect (i.e. p+p = n or n+n = p)

SLT/SLTU Operation

- Use SLT for signed operand and SLTU for unsigned operands
- An SLT instruction subtracts A-B and examine sign of the result and the overflow test to determine if it should set the result
- Tests to determine less-than (negative) condition
  - < Signed: (Neg. & No OV) OR (Pos. & OV)
  - < Unsigned: (Unsigned OV)
    - For unsigned subtraction, overflow is defined when the result is negative (i.e. Cout = 0)…thus we use that test

Branch Example 1

C Code

```
if A > B     (&A in $t0)
  A = A + B (&B in $t1)
else
  A = 1
```

MIPS Assembly

```
        .text
        LW     $t2,0($t0)
        LW     $t3,0($t1)
        SLT    $1,$t3,$t2
        BEQ    $1,$0,ELSE
        ADD    $t2,$t2,$t3
        B      NEXT
ELSE:   ADDI   $t2,$0,1
NEXT:   SW     $t2,0($t0)
        ----
```

Could use pseudo-inst. "BLE $4,$5,ELSE"

This branch skips over the "else" portion. This is a pseudo-instruction and is translated to BEQ $0,$0,next

Branch Example 2

C Code

```
for(i=0;i < 10;i++) ($t0=i)
  j = j + i;        ($t1=j)
```

MIPS Assembly

```
        .text
        ADDI   $t0,$0,$0
LOOP:   SLTI   $1,$t0,10
        BEQ    $1,$0,NEXT
        ADD    $t1,$t1,$t0
        ADD    $t0,$t0,1
        B      LOOP
NEXT:   ----
```

Branches if i is not less than 10

Loops back to the comparison check

Branch Example 3

C Code

```
int dat[10];

for(i=0;i < 10;i++)
  data[i] = 5;
```

M68000 Assembly

```
        .data
dat:    .space  40

        .text
        la      $t0,dat
        addi    $t1,$zero,10
        addi    $t2,$zero,5
LOOP:   sw      $t2,0($t0)
        addi    $t0,$t0,4
        addi    $t1,$t1,-1
        bnez    $t1,$zero,LOOP
NEXT:   ----
```

Branch Example 4

C Code

```
char A[] = “hello world”;
char B[50];
// strcpy(B,A);
i=0;
while(A[i] != 0){
  B[i] = A[i]; i++;
}
B[i] = 0;
```

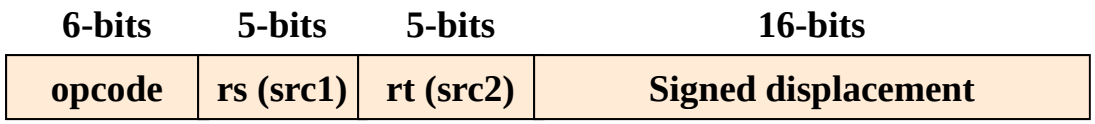
M68000 Assembly

```
        .data
A:      .asciiz “hello world”
B:      .space  50

        .text
        la      $t0,A
        la      $t1,B
LOOP:   lb      $t2,0($t0)
        beq     $t2,$zero,NEXT
        sb      $t2,0($t1)
        addi    $t0,$t0,1
        addi    $t1,$t1,1
        b       LOOP
NEXT:   sb      $t2,0($t1)
```

Branch Machine Code Format

- Branch instructions use the I-Type Format



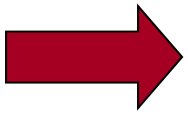
- Operation: PC = PC + {disp.,00}
- Displacement notes
  - Displacement is the value that should be added to the PC so that it now points to the desired branch location
  - Processor appends two 0's to end of disp. since all instructions are 4-byte words
    - Essentially, displacement is in units of words
    - Effective range of displacement is an 18-bit signed value = ±128KB address space (i.e. can't branch anywhere in memory…but long branches are rare and there is a mechanism to handle them)

Branch Displacement

- To calculate displacement you must know where instructions are stored in memory (relative to each other)
  - Don't worry, assembler finds displacement for you…you just use the label

```
        .text
        ADDI   $8,$0,$0
        ADDI   $7,$0,10
LOOP:   SLTI   $1,$8,10
        BEQ    $1,$0,NEXT
        ADD    $9,$9,$8
        ADD    $8,$8,1
        BEQ    $0,$0,LOOP
NEXT:   ----
```

MIPS Assembly



A	ADDI
A + 0x4	ADDI
A + 0x8	SLTI
A + 0xC	BEQ
A + 0x10	ADD
A + 0x14	ADD
A + 0x18	BEQ
A + 0x1C	----

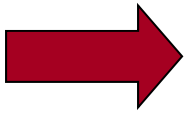
1 word for each instruction

Calculating Displacements

- Disp. = [(Addr. of Target) – (Addr. of Branch + 4)] / 4
  - Constant 4 is due to the fact that by the time the branch executes the PC will be pointing at the instruction after it (i.e. plus 4 bytes)
- Following slides will show displacement calculation for BEQ $1,$0,NEXT

```
        .text
        ADDI   $8,$0,$0
        ADDI   $7,$0,10
LOOP:   SLTI   $1,$8,10
        BEQ    $1,$0,NEXT
        ADD    $9,$9,$8
        ADD    $8,$8,1
        BEQ    $0,$0,LOOP
NEXT:   ----
```

MIPS Assembly



A	ADDI
A + 0x4	ADDI
A + 0x8	SLTI
A + 0xC	BEQ
A + 0x10	ADD
A + 0x14	ADD
A + 0x18	BEQ
A + 0x1C	----

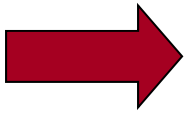
1 word for each instruction

Calculating Displacements

- Disp. = [(Addr. of Target) – (Addr. of Branch + 4)] / 4
- Disp. = (A+0x1C) – (A+0x0C+ 4) = 0x1C – 0x10 = 0x0C / 4
  = 0x03

```
        .text
        ADDI   $8,$0,$0
        ADDI   $7,$0,10
LOOP:   SLTI   $1,$8,10
        BEQ    $1,$0,NEXT
        ADD    $9,$9,$8
        ADD    $8,$8,1
        BEQ    $0,$0,LOOP
NEXT:   ----
```

MIPS Assembly



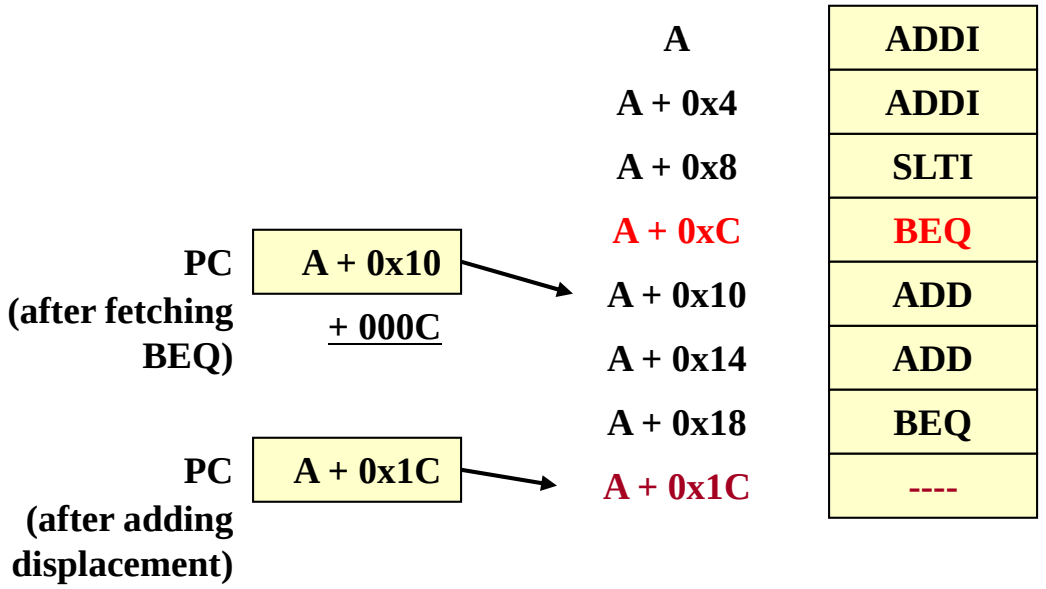
A	ADDI
A + 0x4	ADDI
A + 0x8	SLTI
A + 0xC	BEQ
A + 0x10	ADD
A + 0x14	ADD
A + 0x18	BEQ
A + 0x1C	----

1 word for each instruction

Calculating Displacements

- If the BEQ does in fact branch, it will add the displacement ({0x03, 00} = 0x000C) to the PC (A+0x10) and thus point to the instruction at (A+0x1C)

```
        .text
        ADDI   $8,$0,$0
        ADDI   $7,$0,10
LOOP:   SLTI   $1,$8,10
        BEQ    $1,$0,NEXT
        ADD    $9,$9,$8
        ADD    $8,$8,1
        BEQ    $0,$0,LOOP
NEXT:   ----
```



MIPS Assembly

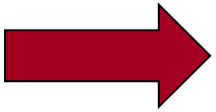
BEQ $1,$0,0x03

opcode	rs	rt	immediate
000100	00001	00000	0000 0000 0000 0011

Another Example

- Disp. = [(Addr. of Label) – (Addr. of Branch + 4)] / 4
- Disp. = (A+0x04) – (A+0x14 + 4) = 0x04 – 0x18
  = 0xFFEC / 4 = 0xFFFB

```
        .text
        ADDI   $8,$0,$0
LOOP:   SLTI   $1,$8,10
        BEQ    $1,$0,NEXT
        ADD    $9,$9,$8
        ADD    $8,$8,1
        BEQ    $0,$0,LOOP
NEXT:   ----
```



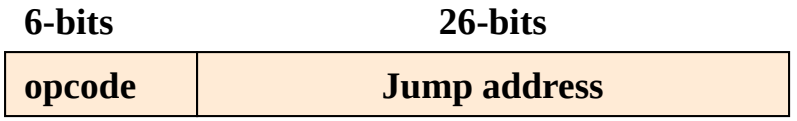
A	ADDI
A + 0x4	SLTI
A + 0x8	BEQ
A + 0xC	ADD
A + 0x10	ADD
A + 0x14	BEQ
A + 0x18	----

BEQ $0,$0,0xFFFB

opcode	rs	rt	immediate
000100	00000	00000	1111 1111 1111 1011

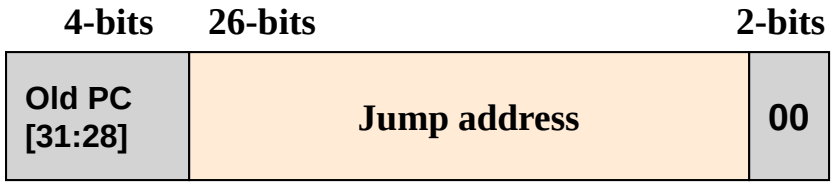
Jump Instructions

- Jumps provide method of branching beyond range of 16-bit displacement
- Syntax: J *label/address*
  - Operation: PC = address
  - Address is appended with two 0s just like branch displacement yielding a 28-bit address with upper 4-bits of PC unaffected
- New instruction format: J-Type



Sample Jump instruction

PC before execution of Jump



New PC after execution of Jump

Jump Example

- Take 28 LSB's of target address, remove LSB's (which are 0s) and store 26-bits in the jump instruction

```
        .text
        ADDI    $8,$0,$0
        SLTI    $1,$8,10
        BEQ     $1,$0,SKIP
        J       L1
SKIP:   ADD     $8,$8,1
        ...
L1:     SUB
```

PC before exec. of jump:

E0400010

PC after exec. of jump:

EC800004

1110 1100 1000 0000 0000 0000 0000 01 00

0xE0400000	ADDI
0xE0400004	SLTI
0xE0400008	BEQ
0xE040000C	BEQ
0xE0400010	J
0xE0400014	ADD

0xEC800004	SUB

opcode	Target Address / 4
0000 10	11 0010 0000 0000 0000 0000 0001

0xC8000004 / 4

Jump Register

- 'jr' instruction can be used if a full 32-bit jump is needed or variable jump address is needed
- Syntax: JR rs
  - Operation: PC = R[s]
  - R-Type machine code format
- Usage:
  - Can load rs with an immediate address
  - Can calculate rs for a variable jump (class member functions, switch statements, etc.)