Revised: 4/28/2020

Assembly Language II

Michael C. Hackett

Assistant Professor, Computer Science

Community College of Philadelphia

Lecture Topics

- Memory Reference Instructions
- Transfer of Control
  - Conditional Transfer
  - Unconditional Transfer
  - Decisions
  - Repetition
- Memory Arrays
- Functions
- Strings
  - Printing Strings
  - Processing String Data

Memory Reference Instructions

- The previous lecture dealt with (constant) data loaded into registers.
- This lecture will begin with storing and retrieving data from main memory.
- The previous lecture defined a **load** operation as the process of placing data into a register, either as a constant value or data loaded from main memory
- The process of placing data from a register into main memory is called **store** operation

Memory Reference Instructions

- To include data that is to be stored into main memory when the program begins, we need identifiers in the assembly source code to differentiate assembly instructions from that data.

- **Assembler directives** tell the assembler how to correctly translate the statements after the directive.
  - The **`.text`** directive tells the assembler to translate the subsequent statements as instructions
  - The **`.data`** directive tells the assembler to translate the subsequent statements as data stored in main memory

Memory Reference Instructions

```
.text
#Instructions go here

.data
#Data to be stored to main memory goes here
```

- The **`.data`** directive could come before **`.text`**
  - Normal convention is **`.text`** is at the beginning and **`.data`** is at the end

Memory Reference Instructions

- Values in the data segment have the following format:

```
[label:]   type        value(s)                  [#comment]
```

- The fields in brackets are optional
  - The brackets themselves, [and], are not literally in the instruction

Memory Reference Instructions

- Values in the data segment have the following format:

```
[label:]   type      value(s)              [#comment]
```

- A label can be used by an instruction to access that value
- You could think of using a label like how you would name a variable in a high-level language
  - Though they are not exactly the same thing

Memory Reference Instructions

- Values in the data segment have the following format:

```
[label:]   type       value(s)                [#comment]
```

- The type directive specifies the value's data type
  - Like how you would specify an int, or String, or boolean in Java
- The **`.word`** directive is used for integers
  - Akin to the int type in Java or C++

Memory Reference Instructions

- Values in the data segment have the following format:

```
[label:]   type        value(s)              [#comment]
```

- The values are simply entered as literal values, separated by commas

Memory Reference Instructions

```
.text
#Instructions go here

.data
x:   .word    7
y:   .word    -1
z:   .word    9, 10, 0xff
     .word    11
```

Memory Reference Instructions

- In the MARS simulator, main memory addresses begin at 10010000_{16}
- Recall that the word size for MIPS is 4-bytes
- Each memory address references one byte (8 bits) of information

Memory Reference Instructions

```
.data
x: .word   7
y: .word   -1
z: .word   9, 10, 0xff
   .word   11
```

ADDRESS	DATA
10010000	00
10010001	00
10010002	00
10010003	00
10010004	00
10010005	00
10010006	00
10010007	00
10010008	00
10010009	00
1001000A	00
1001000B	00

ADDRESS	DATA
1001000C	00
1001000D	00
1001000E	00
1001000F	00
10010010	00
10010011	00
10010012	00
10010013	00
10010014	00
10010015	00
10010016	00
10010017	00

Memory Reference Instructions

```
.data
x: .word   7
y: .word   -1
z: .word   9, 10, 0xff
   .word   11
```

ADDRESS	DATA
10010000	**00**
10010001	**00**
10010002	**00**
10010003	**07**
10010004	00
10010005	00
10010006	00
10010007	00
10010008	00
10010009	00
1001000A	00
1001000B	00

ADDRESS	DATA
1001000C	00
1001000D	00
1001000E	00
1001000F	00
10010010	00
10010011	00
10010012	00
10010013	00
10010014	00
10010015	00
10010016	00
10010017	00

Memory Reference Instructions

```
.data
x: .word   7
y: .word   -1
z: .word   9, 10, 0xff
   .word   11
```

ADDRESS	DATA
10010000	00
10010001	00
10010002	00
10010003	07
10010004	**ff**
10010005	**ff**
10010006	**ff**
10010007	**ff**
10010008	00
10010009	00
1001000A	00
1001000B	00

ADDRESS	DATA
1001000C	00
1001000D	00
1001000E	00
1001000F	00
10010010	00
10010011	00
10010012	00
10010013	00
10010014	00
10010015	00
10010016	00
10010017	00

Memory Reference Instructions

```
.data
x: .word   7
y: .word   -1
z: .word   9, 10, 0xff
   .word   11
```

ADDRESS	DATA
10010000	00
10010001	00
10010002	00
10010003	07
10010004	ff
10010005	ff
10010006	ff
10010007	ff
10010008	**00**
10010009	**00**
1001000A	**00**
1001000B	**09**

ADDRESS	DATA
1001000C	00
1001000D	00
1001000E	00
1001000F	00
10010010	00
10010011	00
10010012	00
10010013	00
10010014	00
10010015	00
10010016	00
10010017	00

Memory Reference Instructions

```
.data
x: .word   7
y: .word   -1
z: .word   9, 10, 0xff
   .word   11
```

ADDRESS	DATA
10010000	00
10010001	00
10010002	00
10010003	07
10010004	ff
10010005	ff
10010006	ff
10010007	ff
10010008	00
10010009	00
1001000A	00
1001000B	09

ADDRESS	DATA
1001000C	**00**
1001000D	**00**
1001000E	**00**
1001000F	**0a**
10010010	00
10010011	00
10010012	00
10010013	00
10010014	00
10010015	00
10010016	00
10010017	00

Memory Reference Instructions

```
.data
x: .word   7
y: .word   -1
z: .word   9, 10, 0xff
   .word   11
```

ADDRESS	DATA
10010000	00
10010001	00
10010002	00
10010003	07
10010004	ff
10010005	ff
10010006	ff
10010007	ff
10010008	00
10010009	00
1001000A	00
1001000B	09

ADDRESS	DATA
1001000C	00
1001000D	00
1001000E	00
1001000F	0a
10010010	**00**
10010011	**00**
10010012	**00**
10010013	**ff**
10010014	00
10010015	00
10010016	00
10010017	00

Memory Reference Instructions

```
.data
x: .word   7
y: .word   -1
z: .word   9, 10, 0xff
   .word   11
```

ADDRESS	DATA
10010000	00
10010001	00
10010002	00
10010003	07
10010004	ff
10010005	ff
10010006	ff
10010007	ff
10010008	00
10010009	00
1001000A	00
1001000B	09

ADDRESS	DATA
1001000C	00
1001000D	00
1001000E	00
1001000F	0a
10010010	00
10010011	00
10010012	00
10010013	ff
10010014	**00**
10010015	**00**
10010016	**00**
10010017	**0b**

Memory Reference Instructions

- The **`lw`** mnemonic **l**oads a **w**ord from main memory into a register.

```
lw   $rd, label
```

  - $rd = Destination Register

- This will load the word identified by label to the destination register

Memory Reference Instructions

```
.text
lw   $t0, x          #Loads 7 (0x07) into $t0
lw   $t1, z          #Loads 9 (0x09)into $t1
lw   $t2, z+8        #Loads 0xff (255) into $t2

.data
x:   .word    7
y:   .word    -1
z:   .word    9, 10, 0xff
     .word    11
```

Memory Reference Instructions

- This statement might look a little odd:

```
lw    $t2, z+8
```

- **z+8** indicates the word to load is eight bytes beyond the start of z

Memory Reference Instructions

`lw $t2, z`

	ADDRESS	DATA
x:	10010000	00
	10010001	00
	10010002	00
	10010003	07
y:	10010004	ff
	10010005	ff
	10010006	ff
	10010007	ff
z:	10010008	**00**
	10010009	**00**
	1001000A	**00**
	1001000B	**09**

ADDRESS	DATA
1001000C	00
1001000D	00
1001000E	00
1001000F	0a
10010010	00
10010011	00
10010012	00
10010013	ff
10010014	00
10010015	00
10010016	00
10010017	0b

Memory Reference Instructions

`lw $t2, z+4`

	ADDRESS	DATA
x:	10010000	00
	10010001	00
	10010002	00
	10010003	07
y:	10010004	ff
	10010005	ff
	10010006	ff
	10010007	ff
z:	10010008	00
	10010009	00
	1001000A	00
	1001000B	09

ADDRESS	DATA
1001000C	**00**
1001000D	**00**
1001000E	**00**
1001000F	**0a**
10010010	00
10010011	00
10010012	00
10010013	ff
10010014	00
10010015	00
10010016	00
10010017	0b

Memory Reference Instructions

`lw $t2, z+8`

	ADDRESS	DATA
x:	10010000	00
	10010001	00
	10010002	00
	10010003	07
y:	10010004	ff
	10010005	ff
	10010006	ff
	10010007	ff
z:	10010008	00
	10010009	00
	1001000A	00
	1001000B	09

ADDRESS	DATA
1001000C	00
1001000D	00
1001000E	00
1001000F	0a
10010010	**00**
10010011	**00**
10010012	**00**
10010013	**ff**
10010014	00
10010015	00
10010016	00
10010017	0b

Memory Reference Instructions

`lw $t2, z+12`

Or…

`lw $t2, x+20`

	ADDRESS	DATA
x:	10010000	00
	10010001	00
	10010002	00
	10010003	07
y:	10010004	ff
	10010005	ff
	10010006	ff
	10010007	ff
z:	10010008	00
	10010009	00
	1001000A	00
	1001000B	09

ADDRESS	DATA
1001000C	00
1001000D	00
1001000E	00
1001000F	0a
10010010	00
10010011	00
10010012	00
10010013	ff
10010014	**00**
10010015	**00**
10010016	**00**
10010017	**0b**

Memory Reference Instructions

- The **sw** mnemonic **s**tores a **w**ord from register into main memory.

```
sw   $rs, label
```

  - $rs = Source Register

- This will load the word in the source register to the word in main memory identified by the label

Memory Reference Instructions

```
#Swaps two items in memory

.text
lw   $t0, x          #Loads 7 (0x07) into $t0
lw   $t1, y          #Loads -1 (0xffffffff) into $t1
sw   $t1, x          #Stores -1 (0xffffffff) to x
sw   $t0, y          #Stores 7 (0x07) to y

.data
x:   .word     7
y:   .word     -1
z:   .word     9, 10, 0xff
     .word     11
```

Memory Reference Instructions

- The previous examples, where we used labels to refer to memory locations, is called **symbolic memory references**

- **Non-symbolic memory references** (or *explicit addressing*) is when we load a memory address to a register
  - *Not the word beginning at that address*

Memory Reference Instructions

- The **`la`** mnemonic **l**oads an **a**ddress into a register.

```
la   $rd, label
```

  - $rd = Destination Register

- This will load the address identified by label to the destination register

Memory Reference Instructions

- To load the word at that address, we provide a displacement value when using the **la** mnemonic

```
.text
la   $t0, x          #Loads the address of word into $t0
lw   $t1, 0($t0)     #Loads 7 into $t1
lw   $t2, 4($t0)     #Loads 9 into $t2
lw   $t3, 8($t0)     #Loads 11 into $t3

.data
x:   .word     7, 9, 11
```

Memory Reference Instructions

`la   $t0, x`

	ADDRESS	DATA
x:	**10010000**	00
	10010001	00
	10010002	00
	10010003	07
	10010004	00
	10010005	00
	10010006	00
	10010007	09
	10010008	00
	10010009	00
	1001000A	00
	1001000B	0b

ADDRESS	DATA
1001000C	00
1001000D	00
1001000E	00
1001000F	00
10010010	00
10010011	00
10010012	00
10010013	00
10010014	00
10010015	00
10010016	00
10010017	00

Memory Reference Instructions

```
la   $t0, x
lw   $t1, 0($t0)
```

x:

ADDRESS	DATA
10010000	**00**
10010001	**00**
10010002	**00**
10010003	**07**
10010004	00
10010005	00
10010006	00
10010007	09
10010008	00
10010009	00
1001000A	00
1001000B	0b

ADDRESS	DATA
1001000C	00
1001000D	00
1001000E	00
1001000F	00
10010010	00
10010011	00
10010012	00
10010013	00
10010014	00
10010015	00
10010016	00
10010017	00

Memory Reference Instructions

```
la   $t0, x
lw   $t2, 4($t0)
```

x:

ADDRESS	DATA
10010000	00
10010001	00
10010002	00
10010003	07
10010004	**00**
10010005	**00**
10010006	**00**
10010007	**09**
10010008	00
10010009	00
1001000A	00
1001000B	0b

ADDRESS	DATA
1001000C	00
1001000D	00
1001000E	00
1001000F	00
10010010	00
10010011	00
10010012	00
10010013	00
10010014	00
10010015	00
10010016	00
10010017	00

Memory Reference Instructions

```
la   $t0, x
lw   $t3, 8($t0)
```

	ADDRESS	DATA
x:	10010000	00
	10010001	00
	10010002	00
	10010003	07
	10010004	00
	10010005	00
	10010006	00
	10010007	09
	10010008	**00**
	10010009	**00**
	1001000A	**00**
	1001000B	**0b**

ADDRESS	DATA
1001000C	00
1001000D	00
1001000E	00
1001000F	00
10010010	00
10010011	00
10010012	00
10010013	00
10010014	00
10010015	00
10010016	00
10010017	00

Memory Reference Instructions

- Alternatively, you could use the explicit address

```
.text
la   $t0, 0x10010000    #Loads an address into $t0
lw   $t1, 0($t0)        #Loads 7 into $t1
lw   $t2, 4($t0)        #Loads 9 into $t2
lw   $t3, 8($t0)        #Loads 11 into $t3

.data
x:   .word          7, 9, 11
```

Memory Reference Instructions

```
la   $t0, 0x10010000
```

	ADDRESS	DATA
x:	**10010000**	00
	10010001	00
	10010002	00
	10010003	07
	10010004	00
	10010005	00
	10010006	00
	10010007	09
	10010008	00
	10010009	00
	1001000A	00
	1001000B	0b

ADDRESS	DATA
1001000C	00
1001000D	00
1001000E	00
1001000F	00
10010010	00
10010011	00
10010012	00
10010013	00
10010014	00
10010015	00
10010016	00
10010017	00

Memory Reference Instructions

```
la   $t0, 0x10010000
lw   $t1, 0($t0)
```

	ADDRESS	DATA
x:	10010000	**00**
	10010001	**00**
	10010002	**00**
	10010003	**07**
	10010004	00
	10010005	00
	10010006	00
	10010007	09
	10010008	00
	10010009	00
	1001000A	00
	1001000B	0b

ADDRESS	DATA
1001000C	00
1001000D	00
1001000E	00
1001000F	00
10010010	00
10010011	00
10010012	00
10010013	00
10010014	00
10010015	00
10010016	00
10010017	00

Memory Reference Instructions

```
la   $t0, 0x10010000
lw   $t2, 4($t0)
```

x:

ADDRESS	DATA
10010000	00
10010001	00
10010002	00
10010003	07
10010004	**00**
10010005	**00**
10010006	**00**
10010007	**09**
10010008	00
10010009	00
1001000A	00
1001000B	0b

ADDRESS	DATA
1001000C	00
1001000D	00
1001000E	00
1001000F	00
10010010	00
10010011	00
10010012	00
10010013	00
10010014	00
10010015	00
10010016	00
10010017	00

Memory Reference Instructions

```
la   $t0, 0x10010000
lw   $t3, 8($t0)
```

	ADDRESS	DATA
x:	10010000	00
	10010001	00
	10010002	00
	10010003	07
	10010004	00
	10010005	00
	10010006	00
	10010007	09
	10010008	**00**
	10010009	**00**
	1001000A	**00**
	1001000B	**0b**

ADDRESS	DATA
1001000C	00
1001000D	00
1001000E	00
1001000F	00
10010010	00
10010011	00
10010012	00
10010013	00
10010014	00
10010015	00
10010016	00
10010017	00

Transfer of Control

- An assembly program might want to execute instructions in a manner different from the top-down execution shown in previous examples.

- The program may want to make a decision between executing different groups of instructions
  - *Decision Structure* (or *Selection Structure*)
  - Akin to an if-else in a high-level programming language

- The program may want to execute the same instructions repeatedly
  - *Repetitive Structure* (or *Iterative Structure* or *Loop*)
  - Akin to loops in high-level programming languages

Transfer of Control

- **Transfer of control** is when an assembly program departs from the sequential execution of instructions
- A **conditional transfer** is when the transfer may or may not take place
  - Implemented with a **branch** instruction
- An **unconditional transfer** is when the transfer will always take place
  - Implemented with a **jump** instruction

Conditional Transfer

- The **beq** mnemonic **b**ranches if the values in two registers are **eq**ual.

```
beq  $rs, $rt, label
```

  - $rs = Source Register 1
  - $rt = Source Register 2
  - label = instructions identified with the specified label

- If the number in $rs is equal to $rt, then the program will transfer control to the statements identified by the label

Conditional Transfer

- The **bne** mnemonic **b**ranches if the values in two registers are **n**ot **e**qual.

```
bne  $rs, $rt, label
```

  - $rs = Source Register 1
  - $rt = Source Register 2
  - label = instructions identified with the specified label

- If the number in $rs is not equal to $rt, then the program will transfer control to the statements identified by the label

Conditional Transfer

- The **`blt`** mnemonic **b**ranches if the value in the first register is **l**ess **t**han the value in the second register.

```
blt  $rs, $rt, label
```

  - $rs = Source Register 1
  - $rt = Source Register 2
  - label = instructions identified with the specified label

- If the number in $rs is less than $rt, then the program will transfer control to the statements identified by the label

Conditional Transfer

- The **`bgt`** mnemonic **b**ranches if the value in the first register is **g**reater **t**han the value in the second register.

```
bgt  $rs, $rt, label
```

  - $rs = Source Register 1
  - $rt = Source Register 2
  - label = instructions identified with the specified label

- If the number in $rs is greater than $rt, then the program will transfer control to the statements identified by the label

Conditional Transfer

- The **`ble`** mnemonic **b**ranches if the value in the first register is **l**ess than or **e**qual the value in the second register.

```
ble  $rs, $rt, label
```

  - $rs = Source Register 1
  - $rt = Source Register 2
  - label = instructions identified with the specified label

- If the number in $rs is less than or equal to $rt, then the program will transfer control to the statements identified by the label

Conditional Transfer

- The **bge** mnemonic **b**ranches if the value in the first register is **g**reater than or **e**qual to the value in the second register.

```
bge  $rs, $rt, label
```

  - $rs = Source Register 1
  - $rt = Source Register 2
  - label = instructions identified with the specified label

- If the number in $rs is greater than or equal $rt, then the program will transfer control to the statements identified by the label

Conditional Transfer

```
.text
li   $t0, 7                #Loads 7 into $t0
li   $t1, 9                #Loads 9 into $t1
beq  $t0, $t1, test1       #Go to test1 if $t0 == $t1
blt  $t0, $t1, test2       #Go to test2 if $t0 < $t1

test1:
li $t3, 8                  #Loads 8 into $t3

test2:
li $t4, 10                 #Loads 10 into $t4
```

Conditional Transfer

```
.text
li   $t0, 7                  #Loads 7 into $t0
li   $t1, 9                  #Loads 9 into $t1
beq  $t0, $t1, test1         #Go to test1 if $t0 == $t1
blt  $t0, $t1, test2         #Go to test2 if $t0 < $t1

test1:
li $t3, 8                    #Loads 8 into $t3

test2:
li $t4, 10                   #Loads 10 into $t4
```

Conditional Transfer

```
.text
li   $t0, 9                 #Loads 9 into $t0
li   $t1, 9                 #Loads 9 into $t1
beq  $t0, $t1, test1        #Go to test1 if $t0 == $t1
blt  $t0, $t1, test2        #Go to test2 if $t0 < $t1

test1:
li $t3, 8                   #Loads 8 into $t3

test2:
li $t4, 10                  #Loads 10 into $t4
```

Conditional Transfer

```
.text
li   $t0, 11              #Loads 11 into $t0
li   $t1, 9               #Loads 9 into $t1
beq  $t0, $t1, test1      #Go to test1 if $t0 == $t1
blt  $t0, $t1, test2      #Go to test2 if $t0 < $t1

test1:
li $t3, 8                 #Loads 8 into $t3

test2:
li $t4, 10                #Loads 10 into $t4
```

Unconditional Transfer

- The `j` mnemonic **j**umps to a label.

```
j    label
```

  - label = instructions identified with the specified label

- The program will immediately transfer control to the statements identified by the label
- This is an example of a **Jump Format** Instruction (or **J-format**)
  - More details on this in a later lecture

Unconditional Transfer

```
.text
li $t0, 7           #Loads 7 into $t0
j test2             #Jumps to test2 section

test1:
li $t2, 8           #Loads 8 into $t2
j done              #Jumps to done section

test2:
li $t1, 10          #Loads 10 into $t1
j test1             #Jumps to test1 section

done:
li $t3, 9           #Loads 10 into $t1
```

\$t0	0x00
\$t1	0x00
\$t2	0x00
\$t3	0x00

Unconditional Transfer

```
.text
li $t0, 7          #Loads 7 into $t0
j test2            #Jumps to test2 section

test1:
li $t2, 8          #Loads 8 into $t2
j done             #Jumps to done section

test2:
li $t1, 10         #Loads 10 into $t1
j test1            #Jumps to test1 section

done:
li $t3, 9          #Loads 10 into $t1
```

$t0	**0x07**
$t1	0x00
$t2	0x00
$t3	0x00

Unconditional Transfer

```
.text
li $t0, 7           #Loads 7 into $t0
j test2             #Jumps to test2 section

test1:
li $t2, 8           #Loads 8 into $t2
j done              #Jumps to done section

test2:
li $t1, 10          #Loads 10 into $t1
j test1             #Jumps to test1 section

done:
li $t3, 9           #Loads 10 into $t1
```

$t0	0x07
$t1	**0x0a**
$t2	0x00
$t3	0x00

Unconditional Transfer

```
.text
li $t0, 7        #Loads 7 into $t0
j test2          #Jumps to test2 section

test1:
li $t2, 8        #Loads 8 into $t2
j done           #Jumps to done section

test2:
li $t1, 10       #Loads 10 into $t1
j test1          #Jumps to test1 section

done:
li $t3, 9        #Loads 10 into $t1
```

\$t0	0x07
\$t1	0x0a
\$t2	**0x08**
\$t3	0x00

Unconditional Transfer

```
.text
li $t0, 7          #Loads 7 into $t0
j test2            #Jumps to test2 section

test1:
li $t2, 8          #Loads 8 into $t2
j done             #Jumps to done section

test2:
li $t1, 10         #Loads 10 into $t1
j test1            #Jumps to test1 section

done:
li $t3, 9          #Loads 10 into $t1
```

$t0	0x07
$t1	0x0a
$t2	0x08
$t3	**0x09**

Decisions

- While we don’t have if-else statements in an assembly language, we can use branch and jump instructions to achieve the same decision-making processes.
- To demonstrate, we’ll convert a couple of high-level language if-else statements to assembly instructions

Decisions

- An if-else in Java/C++ that decides whether to add 1 to $t2 or $t3, depending on if $t0 is greater than $t1

```
if ($t0 > $t1) {
  $t2++;
}
else {
  $t3++;
}
```

Decisions

```
.text
li $t0, 9              #Loads 9 into $t0
li $t1, 7              #Loads 7 into $t1
bgt $t0, $t1, path1
addi $t3, $t3, 1       #Adds 1 to $t3
j done

path1:
addi $t2, $t2, 1       #Adds 1 to $t2

done:
#Finished
```

$t0	**0x09**
$t1	**0x07**
$t2	**0x01**
$t3	0x00

Decisions

```
.text
li $t0, 7               #Loads 7 into $t0
li $t1, 9               #Loads 9 into $t1
bgt $t0, $t1, path1
addi $t3, $t3, 1        #Adds 1 to $t3
j done

path1:
addi $t2, $t2, 1        #Adds 1 to $t2

done:
#Finished
```

$t0	**0x07**
$t1	**0x09**
$t2	0x00
$t3	**0x01**

Decisions

- An if-else in Java/C++ that decides whether to:
  - Add 1 to $t2 if $t0 is greater than $t1
  - Add 1 to $t3 if $t0 is less than $t1
  - Add 1 to $t4 if $t0 is equal to $t1

```
if ($t0 > $t1) {
   $t2++;
}
else if ($t0 < $t1) {
   $t3++;
}
else {
   $t4++;
}
```

Decisions

```
.text
li $t0, 9                   #Loads 9 into $t0
li $t1, 7                   #Loads 7 into $t1
bgt $t0, $t1, path1
blt $t0, $t1, path2
addi $t4, $t4, 1            #Adds 1 to $t4
j done

path1:
addi $t2, $t2, 1            #Adds 1 to $t2
j done

path2:
addi $t3, $t3, 1            #Adds 1 to $t3

done:
#Finished
```

$t0	**0x09**
$t1	**0x07**
$t2	**0x01**
$t3	0x00
$t4	0x00

Decisions

```
.text
li $t0, 7                   #Loads 7 into $t0
li $t1, 9                   #Loads 9 into $t1
bgt $t0, $t1, path1
blt $t0, $t1, path2
addi $t4, $t4, 1            #Adds 1 to $t4
j done

path1:
addi $t2, $t2, 1            #Adds 1 to $t2
j done

path2:
addi $t3, $t3, 1            #Adds 1 to $t3

done:
#Finished
```

$t0	**0x07**
$t1	**0x09**
$t2	0x00
$t3	**0x01**
$t4	0x00

Decisions

```
.text
li $t0, 9                    #Loads 9 into $t0
li $t1, 9                    #Loads 9 into $t1
bgt $t0, $t1, path1
blt $t0, $t1, path2
addi $t4, $t4, 1             #Adds 1 to $t4
j done

path1:
addi $t2, $t2, 1             #Adds 1 to $t2
j done

path2:
addi $t3, $t3, 1             #Adds 1 to $t3

done:
#Finished
```

$t0	**0x09**
$t1	**0x09**
$t2	0x00
$t3	0x01
$t4	**0x01**

Repetition

- We also don't have explicit loops (for, while, do-while) in an assembly language, but we can use branch and jump instructions to achieve the same repetitive processes.

- To demonstrate, we'll convert a couple of high-level language loops to assembly instructions

Repetition

- You’ll recall from introductory programming classes that loops are either pre-test or post-test

- A **pre-test loop** tests its termination condition prior to starting each iteration.
  - For loops and While loops are pre-test loops

- A **post-test loop** tests its termination condition after executing each iteration
  - Do-While loops are post-test loops

Repetition

- A while loop in Java/C++ that adds up the numbers 1 through 5

```
$t0 = 0
$t1 = 1
while($t1 <= 5) {
  $t0 += $t1;
  $t1++;
}
```

Repetition

```
.text
li $t0, 0                #Loads 0 into $t0
li $t1, 1                #Acts as a counter
li $t2, 5                #Used to stop the loop

loop:
bgt $t1, $t2, done       #Stops if $t1 > $t2
add $t0, $t0, $t1        #Adds $t1 to $t0
addi $t1, $t1, 1         #Adds 1 to $t1
j loop


done:
#Finished
```

$t0	0x00
$t1	0x00
$t2	0x00
$t3	0x00
$t4	0x00

Repetition

```
.text
li $t0, 0              #Loads 0 into $t0
li $t1, 1              #Acts as a counter
li $t2, 5              #Used to stop the loop

loop:
bgt $t1, $t2, done     #Stops if $t1 > $t2
add $t0, $t0, $t1      #Adds $t1 to $t0
addi $t1, $t1, 1       #Adds 1 to $t1
j loop


done:
#Finished
```

$t0	**0x00**
$t1	**0x01**
$t2	**0x05**
$t3	0x00
$t4	0x00

Repetition

```
.text
li $t0, 0                #Loads 0 into $t0
li $t1, 1                #Acts as a counter
li $t2, 5                #Used to stop the loop

loop:
bgt $t1, $t2, done       #Stops if $t1 > $t2
add $t0, $t0, $t1        #Adds $t1 to $t0
addi $t1, $t1, 1         #Adds 1 to $t1
j loop


done:
#Finished
```

\$t0	**0x01**
\$t1	**0x02**
\$t2	0x05
\$t3	0x00
\$t4	0x00

Repetition

```
.text
li $t0, 0                   #Loads 0 into $t0
li $t1, 1                   #Acts as a counter
li $t2, 5                   #Used to stop the loop

loop:
bgt $t1, $t2, done          #Stops if $t1 > $t2
add $t0, $t0, $t1           #Adds $t1 to $t0
addi $t1, $t1, 1            #Adds 1 to $t1
j loop


done:
#Finished
```

$t0	**0x03**
$t1	**0x03**
$t2	0x05
$t3	0x00
$t4	0x00

Repetition

```
.text
li $t0, 0               #Loads 0 into $t0
li $t1, 1               #Acts as a counter
li $t2, 5               #Used to stop the loop

loop:
bgt $t1, $t2, done      #Stops if $t1 > $t2
add $t0, $t0, $t1       #Adds $t1 to $t0
addi $t1, $t1, 1        #Adds 1 to $t1
j loop


done:
#Finished
```

$t0	**0x06**
$t1	**0x04**
$t2	0x05
$t3	0x00
$t4	0x00

Repetition

```
.text
li $t0, 0                #Loads 0 into $t0
li $t1, 1                #Acts as a counter
li $t2, 5                #Used to stop the loop

loop:
bgt $t1, $t2, done       #Stops if $t1 > $t2
add $t0, $t0, $t1        #Adds $t1 to $t0
addi $t1, $t1, 1         #Adds 1 to $t1
j loop


done:
#Finished
```

$t0	**0x0a**
$t1	**0x05**
$t2	0x05
$t3	0x00
$t4	0x00

Repetition

```
.text
li $t0, 0              #Loads 0 into $t0
li $t1, 1              #Acts as a counter
li $t2, 5              #Used to stop the loop

loop:
bgt $t1, $t2, done     #Stops if $t1 > $t2
add $t0, $t0, $t1      #Adds $t1 to $t0
addi $t1, $t1, 1       #Adds 1 to $t1
j loop


done:
#Finished
```

$t0	**0x0f**
$t1	**0x06**
$t2	0x05
$t3	0x00
$t4	0x00

Repetition

```
.text
li $t0, 0               #Loads 0 into $t0
li $t1, 1               #Acts as a counter
li $t2, 5               #Used to stop the loop

loop:
bgt $t1, $t2, done      #Stops if $t1 > $t2
add $t0, $t0, $t1       #Adds $t1 to $t0
addi $t1, $t1, 1        #Adds 1 to $t1
j loop


done:
#Finished
```

$t0	0x0f
$t1	0x06
$t2	0x05
$t3	0x00
$t4	0x00

Repetition

```
.text
li $t0, 0               #Loads 0 into $t0
li $t1, 1               #Acts as a counter
li $t2, 5               #Used to stop the loop

loop:
bgt $t1, $t2, done      #Stops if $t1 > $t2
add $t0, $t0, $t1       #Adds $t1 to $t0
addi $t1, $t1, 1        #Adds 1 to $t1
j loop


done:
#Finished
```

\$t0	0x0f
\$t1	0x06
\$t2	0x05
\$t3	0x00
\$t4	0x00

Repetition

- A do-while loop in Java/C++ that adds up the numbers 1 through 5

```
$t0 = 0
$t1 = 1
do {
   $t0 += $t1;
   $t1++;
} while($t1 <= 5)
```

Repetition

```
.text
li $t0, 0                  #Loads 0 into $t0
li $t1, 1                  #Acts as a counter
li $t2, 5                  #Used to stop the loop

loop:
add $t0, $t0, $t1          #Adds $t1 to $t0
addi $t1, $t1, 1           #Adds 1 to $t1
bgt $t1, $t2, done         #Stops if $t1 > $t2
j loop


done:
#Finished
```

Memory Arrays

- **Arrays** are sequences of contiguous memory

- Instead of referencing values using a subscript (like **`array[i]`** in a high-level language) we access elements using the starting address of the array and adding a displacement value

Memory Arrays

```
.text
la $t0, array           #Loads the starting address of array
lw $t1, 0($t0)          #Loads 35 to $t1
lw $t2, 4($t0)          #Loads 67 to $t2
lw $t3, 8($t0)          #Loads 42 to $t3
lw $t4, 12($t0)         #Loads -6 to $t4


.data
array: .word    35, 67, 42, -6
```

Memory Arrays

```
.text
la $t0, array           #Loads the starting address of array
lw $t1, 0($t0)          #Loads 35 to $t1
lw $t2, 4($t0)          #Loads 67 to $t2
lw $t3, 8($t0)          #Loads 42 to $t3
lw $t4, 12($t0)         #Loads -6 to $t4
sw $t4, 0($t0)          #Stores -6 to 0($t0)
sw $t3, 4($t0)          #Stores 42 to 4($t0)
sw $t2, 8($t0)          #Stores 67 to 8($t0)
sw $t1, 12($t0)         #Stores 35 to 12($t0)


.data
array: .word      35, 67, 42, -6
```

Functions

- A **subroutine** is a group of instructions that are executed when *called* or *invoked*.
  - A **function** is a subroutine that returns data when called.
  - A **procedure** is a subroutine that does not return data when called.
  - A **method** is a subroutine (function or procedure) that is part of a software object.
- All four terms are generally used interchangeably.
- We'll use the term **function** to describe any subroutine in an assembly program

Functions

- The **`jal`** mnemonic calls a function using a **j**ump **a**nd **l**ink process.

  `jal  label`

  - label = instructions identified with the specified label

- This instruction will load the address of the next instruction after the function call to register $ra
  - This is the return address, or the address where execution should resume when the function is finished
- After loading the address of the next instruction to $ra, the program jumps to the statements identified by the label

Functions

- The **`jr`** mnemonic is used to **j**ump to a **r**egister that contains the address of instruction.

```
jr   $ra
```

- This instruction is executed at the end of a function’s statements to return back to where the function was first called.
  - This address is stored in $ra when the **`jal`** instruction is used.

Functions

```
.text
li $t0, 5                 #Loads 5 into $t0
jal func
li $t2, 3                 #Loads 3 into $t2
j done

func:
li $t1, 7                 #Loads 7 into $t1
jr $ra                    #Returns


done:
#Finished
```

$t0	0x00
$t1	0x00
$t2	0x00
$t3	0x00
$t4	0x00

Functions

```
.text
li $t0, 5                #Loads 5 into $t0
jal func
li $t2, 3                #Loads 3 into $t2
j done

func:
li $t1, 7                #Loads 7 into $t1
jr $ra                   #Returns


done:
#Finished
```

$t0	**0x05**
$t1	0x00
$t2	0x00
$t3	0x00
$t4	0x00

Functions

```
.text
li $t0, 5                 #Loads 5 into $t0
jal func
li $t2, 3                 #Loads 3 into $t2
j done

func:
li $t1, 7                 #Loads 7 into $t1
jr $ra                    #Returns


done:
#Finished
```

\$t0	0x05
\$t1	**0x07**
\$t2	0x00
\$t3	0x00
\$t4	0x00

Functions

```
.text
li $t0, 5                  #Loads 5 into $t0
jal func
li $t2, 3                  #Loads 3 into $t2
j done

func:
li $t1, 7                  #Loads 7 into $t1
jr $ra                     #Returns


done:
#Finished
```

$t0	0x05
$t1	0x07
$t2	**0x03**
$t3	0x00
$t4	0x00

Functions

```
.text
li $t0, 5                 #Loads 5 into $t0
jal func
li $t2, 3                 #Loads 3 into $t2
j done

func:
li $t1, 7                 #Loads 7 into $t1
jr $ra                    #Returns


done:
#Finished
```

$t0	0x05
$t1	0x07
$t2	0x03
$t3	0x00
$t4	0x00

Functions

- To handle data passed as input to a function (think parameters/arguments) the registers $a0, $a1, $a2, $a3 are used by convention
  - If a function needs more than 4 arguments, a memory array's starting address can be loaded into any of those 4 registers
  - The memory array can contain as many values as needed.
- There isn't an explicit statement to return a value (or values) from a function.
  - The function can simply load data into registers or store data to main memory
  - Typically, $v0 and $v1 are the registers used for returning values from a function

Strings

- Strings allow us to represent non-numeric data (such as letters and symbols)

- Each character in a string is represented using 8 bits
    - This encoding is called ASCII, which is a subset of the larger and more general Unicode
    - Unicode characters are 16 bits and includes foreign alphabets
    - The first 127 characters of Unicode are the ASCII symbols

Strings

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

Strings

- A string is simply an array of 8-bit ASCII characters
  - No different than an array of numeric data, except the binary information of each byte represents an ASCII character instead of an integer
- The string’s address is also the address of the first character

Strings

- The **`.ascii`** directive tells the assembler to initialize the data memory with the characters of a string

```
.data
example1: .ascii    "Hello world!"
```

Strings

- The **`.asciiz`** directive also tells the assembler to initialize the data memory with the characters of a string, but includes a null byte (00000000) at the end
  - This *terminates* the string; In other words, indicates the end of the string

```
.data
example2: .asciiz   "Hello world!"
```

Strings

```
.data
example1: .ascii    "Hello world!"
example2: .asciiz   "Hello world!"
```

- They may look alike, but:
  - **`asciiz`** always uses an extra byte of space
  - No need to “know” the ending address of an **`asciiz`** string
    - The null byte indicates the end of the string’s data

Printing Strings

- To print an string, the string's starting address **must** be placed in register $a0

- The system call code for printing a string is 4
  - 4 **must** be placed in register $v0

- When the above two steps are complete, the **`syscall`** instruction is used to perform the function indicated by the number in $v0

Printing Strings

```
.text
la $a0, example          #Loads (symbolically) the example string's starting address to $a0
la $t0, example2         #Loads (symbolically) the example2 string's starting address to $t0
li $v0, 4                #Sets the system call code for printing a string
syscall                  #4 is in $v0, so the string (starting at the address in $a0) is printed

move $a0, $t0            #Moves the example2 string's starting address from $t0 to $a0
syscall                  #4 is in $v0, so the string (starting at the address in $a0) is printed


.data
example:  .asciiz        "Hello World!"
example2: .asciiz        "Goodbye World!"
```

Output: `Hello World!Goodbye World!`

Printing Strings

```
.text
la $a0, example         #Loads (symbolically) the example string's starting address to $a0
la $t0, example2        #Loads (symbolically) the example2 string's starting address to $t0
li $v0, 4               #Sets the system call code for printing a string
syscall                 #4 is in $v0, so the string (starting at the address in $a0) is printed

move $a0, $t0           #Moves the example2 string's starting address from $t0 to $a0
syscall                 #4 is in $v0, so the string (starting at the address in $a0) is printed


.data
example:  .asciiz       "Hello World!\n"
example2: .asciiz       "Goodbye World!"
```

Use \n to add line breaks in printed output

Output:
```
Hello World!
Goodbye World!
```

Processing String Data

- When processing string data, it is similar to how we processes the data of an array

- Two key differences:
  - Each character is a byte, not a word (4 bytes). We will increment the address register by 1 instead of 4
  - We can't use **`lw`** and **`sw`** to load and store individual characters (since they are bytes, not words).
    - **Byte instructions** are used instead.

Processing String Data

- The **`lbu`** mnemonic **l**oads a **b**yte (**u**nsigned) from main memory into the lower order 8 bits of a register.
  - The upper 24 bits of the register will contain 0's
  - With symbolic addressing:

```
lbu $rd, label
```

    - $rd = Destination Register
    - This would load the first byte/character of the string to $rd

  - With explicit addressing:

```
lbu $rd, d($a0)
```

    - d($a0) = Offset from starting address
    - This would load the d^{th} byte from the address in $a0

Processing String Data

- The **`sb`** mnemonic **s**tores a **b**yte from the lower order 8 bits of a register to main memory.
  - With symbolic addressing:

    ```
    sb   $rs, label
    ```

    - $rs = Source Register
    - This would store the byte (lower order 8 bits) in $rs to the first byte (lower order 8 bits) at the memory address
  - With explicit addressing:

    ```
    sb   $rs, d($a0)
    ```

    - d($a0) = Offset from starting address
    - This would store the byte in $rs to the d^{th} byte from the address in $a0