ISA, Machine Language, and Assembly Language

Machine language

- Bit patterns that are directly executable by computer
- Assembled into files called objects, binaries, executables
    - Looks like gibberish to humans, but it's the binary encoded instructions the CPU understands
    - May also contain data, in addition to instructions

Assembly language

- Symbolic representation of machine language
    - Instructions as mnemonic ASCII strings, e.g., ADD R2,R6,R2
    - Can't run, but mostly human readable
    - Can be hand written or produced by a compiler from source
    - In this module we'll also see Assembly files can contain ASCII labels, e.g., BRnzp LOOP
- **Assembler**: produces object file from assembly file
    - Assembler directives: non-instructions tell assembler what to do

The "Assembler"
– Bridge Between Assembly & Machine Code

ASSEMBLY Program
Text File (.asm)

↓

Assembler
Translates input assembly program to Machine Code

↓

Machine Code
Binary File (.obj)
Can be loaded into memory and executed

Basic Overview of Programming

Programming:

The process of designing/writing/testing/debugging/maintaining the “source code” of computer programs

- Process of taking the 29 assembly instructions in the ISA and solving problems with them!

The 29 Instructions of ISA are like words in the English Language

- We can author great literature, or misuse them all together
- That’s up to us!

We now “know” the language of the LC-4: ***the LC4 ISA instruction set***

- We must learn to use the language to have the CPU do useful things for us
- In the same way that AND/OR/NOT gave rise to a CPU…
- …29 simple instructions give rise to programs that can solve complicated problems!

3 Key Examples in These Slides…

1) Loops/If-Else Statements

   Covers:

   Loading Constants /Arithmetic /Compare /Branch / Basic Jump instructions / Labeling in Assembly

2) Subroutines

   Covers:

   JSR / RET / Labels / Assembly Directives: .FALIGN

3) Accessing Data Memory

   Covers:

   LDR/STR / What is a Pointer? / For Loops

PROGRAMMING CONSTRUCTS

Programming Constructs

Variables – As in algebra, place to "store" information

- Ex: 5+6=11, OR Let A=5, B=6, then A+B=C
- Variables give us freedom to change our programs

Loops – A way to "repeat" a portion of a program over and over again

```
Print Paycheck for Bob
Print Paycheck for Cindy
Print Paycheck for John
```

```
Let X=name
Loop Begin {
    Print Paycheck for X
    Update X
    All done? Yes->exit loop
} Loop End
```

Conditional Control – A way for our program to change natural flow of program based on a condition

- normally programs execute line after line

```
if (today is end of month) {
        print paychecks
} else {
        order paper
}
```

MULTIPLY ALGORITHM TO ASSEMBLY PROGRAM

Multiplication Program in LC-4 Assembly (no loop)

*Implements C=A*B*

Pseudocode of Algorithm:

```
A = 2
B = 3
C = 0

C=C+A
C=C+A
C=C+A
```

Register allocation: R0=A, R1=B, R2=C

Assembly Program:

```
CONST R0, #2
CONST R1, #3
CONST R2, #0

; we add A to itself 3 times
ADD R2, R0, R2
ADD R2, R0, R2
ADD R2, R0, R2
```

Storing Variables A, B, C In Register File

The Program

Multiplication Program in LC-4 Assembly (loop)

*Implements C=A*B*

Pseudocode of Algorithm:

```
A = 2

B = 3

C = 0

while (B > 0)
{

   C = C + A;
   B = B – 1;
}
```

Register allocation: R0=A, R1=B, R2=C

Assembly Program:

```
CONST R0, #2
CONST R1, #3
CONST R2, #0
```

Storing Variables A, B, C In Register File

```
CMPI R1, #0        ; sets NZP
BRnz #3            ; tests NZP
ADD R2, R2, R0     ; C=C+A
ADD R1, R1, #-1    ; B=B-1
BRnzp #-5
```

Multiplication Program in LC-4 Assembly (loop + labels)

*Implements C=A*B*

Pseudocode of Algorithm:

```
A = 2

B = 3

C = 0

while (B > 0)
{

   C = C + A;
   B = B – 1;
}
```

Offsets are calculated for you by assembler

Register allocation: R0=A, R1=B, R2=C

Assembly Program:

```
CONST R0, #2
CONST R1, #3
CONST R2, #0
LOOP
  CMPI R1, #0        ; sets NZP
  BRnz END           ; tests NZP
  ADD R2, R2, R0     ; C=C+A
  ADD R1, R1, #-1    ; B=B-1
  BRnzp LOOP
 END
```

Storing Variables A, B, C In Register File

Label (LOOP)

Label (END)

Multiplication Program in LC-4 Assembly (loop + labels)

*Implements C=A*B*

Pseudocode of Algorithm:

```
A = 2
B = 3
C = 0

while (B > 0)
{

    C = C + A;
    B = B – 1;
}
```

Offsets are calculated for you by assembler

Register allocation: R0=A, R1=B, R2=C

Assembly Program:

```
CONST R0, #2
CONST R1, #3
CONST R2, #0
LOOP
    CMPI R1, #0       ; sets NZP
    BRnz END          ; tests NZP
    ADD R2, R2, R0    ; C=C+A
    ADD R1, R1, #-1   ; B=B-1
    JMP LOOP   ; same idea
  END
```

Storing Variables A, B, C In Register File

Label

SUB instead?

Label

ASSEMBLY PROGRAM TO MACHINE CODE

What Happens Next?

Assembly Program (.ASM):

```
CONST R0, #2
CONST R1, #3
CONST R2, #0

CMPI R1, #0
BRnz #3
ADD R2, R2, R0
ADD R1, R1, #-1
BRnzp #-5
```

Machine Code (.OBJ):

```
1001000000000010 ; CONST
1001001000000011 ; CONST
1001010000000000 ; CONST

0010001100000000 ; CMPI
0000110000000011 ; BRnz
0001010010000000 ; ADD
0001001001111111 ; ADDI
0000111111111011 ; BRnzp
```

An "Assembler" Program is used to translate Assembly Programs (.ASM) into Machine Code (.OBJ)

Operates in two phases:

First phase (1st pass), converts labels into offsets, removes comments

Second phase (2nd pass), converts assembly code into machine code

Uses the ISA to do this, saves in a .OBJ file (object file)

After "Assembler" finishes…a "loader" program loads it into computer's memory

PENNSIM DEMO

SUBROUTINES IN ASSEMBLY: OVERVIEW

Subroutines in Assembly – Overview

- A subroutine is *similar to* a "function" in a high level language
- To enable, call, and return from subroutines in assembly, use the following outline:

1) Give the subroutine a unique name using a *LABEL*

2) Ensure subroutine is loaded at memory address that is multiple of 16 *(How?? Our first directive: .FALIGN)*

3) Pass it *arguments* by using the register file: R0->R6

4) *Call* the subroutine using ISA instruction: *JSR*

JSR = Jump to Subroutine

Mnemonic: JSR IMM11 <LABEL> ; converts label into IMM11

Semantics: R7 = PC + 1 ; stores PC value in R7

PC = (PC & 0x8000) | (IMM11 << 4) ; sets PC = IMM11 << 4

(IMM11 << 4) makes IMM11 a multiple of 16! Extends range of JSR

5) *Return* data using the register file: R0-R6

6) *Return from* the subroutine using ISA pseudo-instruction: **RET**

RET = Return from Subroutine (but its actually a JMPR)

Semantics: JMPR R7 ; PC = R7

SUBROUTINES IN ASSEMBLY: SPECIFIC EXAMPLE

Subroutines in Assembly – Specific Example

LINE #s In Hex

```
0:     CONST R0, #2       ; SETUP ARGUMENTS: A=2
1:     CONST R1, #3       ; B=3
2:     CONST R2, #0       ; C=0
3:     JSR SUB_MULT       ; CALL TO SUBROUTINE: C=A*B
4:     CONST R0, #4       ; SETUP ARGUMENTS: A=4
5:     CONST R1, #5       ; B=5
6:     CONST R2, #0       ; C=0
7:     JSR SUB_MULT       ; CALL TO SUBROUTINE: C=A*B
8:     JMP END            ; jumps over subroutine
9:     .FALIGN            ; aligns the subroutine
A:     SUB_MULT           ; ARGS: R0(A), R1(B), RET: R2(C)
B:       CMPI R1, #0      ; while loop
C:       BRnz END_MULT
D:       ADD R2, R2, R0   ; C=C+A
E:       ADD R1, R1, #-1  ; B=B-1
F:      BRnzp SUB_MULT    ; end loop
10:     END_MULT
11:     RET               ; end subroutine
12:     END               ; end program
```

JSR does the following:

1) R7=PC+1, so R7=3+1 = 4
2) PC=<label> << 4, hence .FALIGN

Subroutines in Assembly – Effect of .FALIGN

JSR can only advance PC to a multiple of 16

This gives JSR a bigger range, since It can only take an 11-bit argument

Effect of .FALIGN -placed "SUB_MULT" at a multiple of 16 (x0010 is 16 in decimal)

Memory

BP	Address	Value
☐	x0000	CONST R0, #2
☐	x0001	CONST R1, #3
☐	x0002	CONST R2, #0
☐	x0003	JSR SUB_MULT
☐	x0004	CONST R0, #4
☐	x0005	CONST R1, #5
☐	x0006	CONST R2, #0
☐	x0007	JSR SUB_MULT
☐	x0008	JMP END
☐	x0009	NOP
☐	x000A	NOP
☐	x000B	NOP
☐	x000C	NOP
☐	x000D	NOP
☐	x000E	NOP
☐	x000F	NOP
☐	SUB_MULT	CMPI R1, #0
☐	x0011	BRnz END_MULT
☐	x0012	ADD R2, R2, R0
☐	x0013	ADD R1, R1, #-1
☐	x0014	BR SUB_MULT
☐	END_MULT	JMPR R7
✔	END	NOP

REGISTER FILE LIMITATIONS & DATA MEMORY MAP

Working with DATA Memory in LC-4 Assembly

ALU can only operate on #'s in the register File

- LC-4 Register File holds only 8 Numbers (in R0->R7)

How could we work with more than 8 numbers at a time?

- *For example…what if we wanted to ADD up 10 numbers?*
- We could use DATA memory to hold the extra #s
  - ➢ One difficulty: ALU can only add #s in the register file
  - ➢ Must get #s out of DATA memory into register file
    - – *Just not all at once!*
- How do we get numbers from DATA memory to register file?
  - ➢ *The LOAD Register instruction (LDR in ISA)*

LC-4 Memory Map

In the LC-4 Architecture Block Diagram:

- Program and Data Memory are separate

Actually, there is only 1 memory:

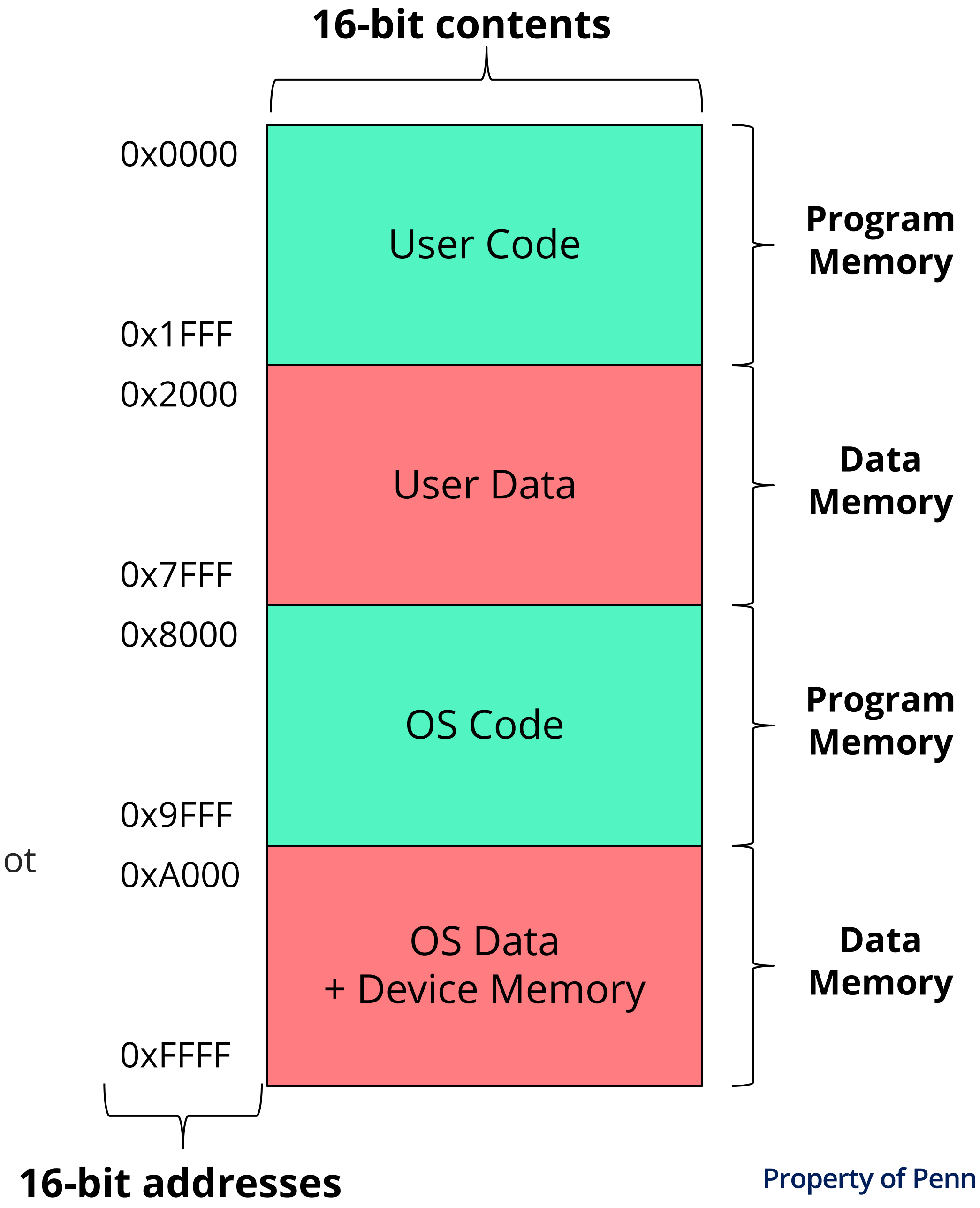
- The separation of Program & Data Mem
  - *is purely logical for the ISA*
- We partition it even further into 2 regions:
  - **User region**
  - **Operating System Region**

User Region

- Programs run by users (MS Word for ex)
- Processes run in user mode with PSR[15]=0 are not allowed to access OS locations in the memory.

Operating System Region

- Processes run in OS mode with PSR[15]=1
- Note: address x8200, first address of your OS



DATA MEMORY EXAMPLE: HOW DO WE ADD UP MORE THAN 8 NUMBERS?

Working with DATA Memory in LC-4 Assembly

How do we add up more than 8 numbers?

Step 1: Load R0 with address x4000

```
CONST R0 x00        ; Load low byte
HICONST R0 x40      ; Load high byte
```

R0	0
R1	0
R2	0
R3	0
R4	0
R5	0
R6	0
R7	0

Register File

Address	Contents
x4000	1
x4001	2
x4002	3
x4003	4
x4004	5
x4005	6
x4006	7
x4007	8
x4008	9
x4009	10

DATA Memory

Working with DATA Memory in LC-4 Assembly

How do we add up more than 8 numbers?

Step 1: Load R0 with address x4000

```
CONST R0 x00      ; Load low byte
HICONST R0 x40    ; Load high byte
```

R0 – hold current DATA Memory Address (a pointer)

R0	x4000
R1	0
R2	0
R3	0
R4	0
R5	0
R6	0
R7	0

Register File

Address	Contents
x4000	1
x4001	2
x4002	3
x4003	4
x4004	5
x4005	6
x4006	7
x4007	8
x4008	9
x4009	10

DATA Memory

Working with DATA Memory in LC-4 Assembly

How do we add up more than 8 numbers?

Step 2: Load R3 with DATA from x4000

```
LDR R3, R0, #0        ; offset 0
```

R0 – hold current DATA Memory Address (a pointer)

R0	x4000
R1	0
R2	0
R3	0
R4	0
R5	0
R6	0
R7	0

Register File

Address	Contents
x4000	1
x4001	2
x4002	3
x4003	4
x4004	5
x4005	6
x4006	7
x4007	8
x4008	9
x4009	10

DATA Memory

Working with DATA Memory in LC-4 Assembly

How do we add up more than 8 numbers?

Step 2: Load R3 with DATA from x4000

```
LDR R3, R0, #0        ; offset 0
```

R0	x4000
R1	0
R2	0
R3	1
R4	0
R5	0
R6	0
R7	0

Register File

Address	Contents
x4000	1
x4001	2
x4002	3
x4003	4
x4004	5
x4005	6
x4006	7
x4007	8
x4008	9
x4009	10

DATA Memory

R0 – hold current DATA Memory Address (a pointer)

R3 – hold contents of address pointed to by R0

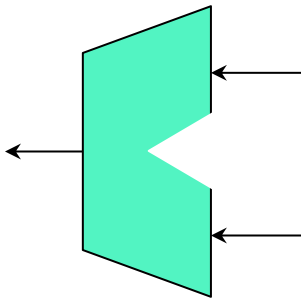
Working with DATA Memory in LC-4 Assembly

How do we add up more than 8 numbers?

Step 3: Use ALU to add 1st # to total

```
ADD R1, R1, R3     ; store total in R1
```

R1
=1



ALU

R1 – hold the "running total"

R0	x4000
R1	0
R2	0
R3	1
R4	0
R5	0
R6	0
R7	0

Register File

Address	Contents
x4000	1
x4001	2
x4002	3
x4003	4
x4004	5
x4005	6
x4006	7
x4007	8
x4008	9
x4009	10

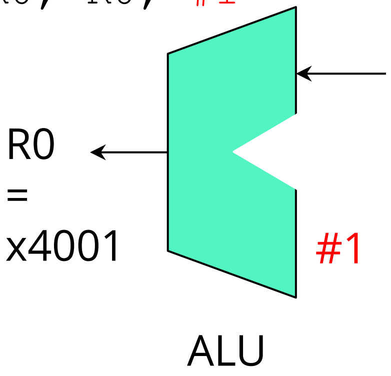
DATA Memory

Working with DATA Memory in LC-4 Assembly

How do we add up more than 8 numbers?

Step 4: Use ALU to increment REG holding DATA Address

```
ADD R0, R0, #1
```



R0	x4000
R1	1
R2	0
R3	1
R4	0
R5	0
R6	0
R7	0

Register File

Address	Contents
x4000	1
x4001	2
x4002	3
x4003	4
x4004	5
x4005	6
x4006	7
x4007	8
x4008	9
x4009	10

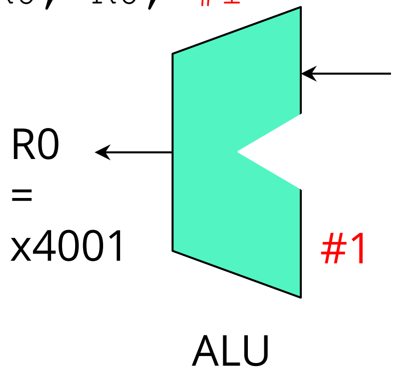
DATA Memory

Working with DATA Memory in LC-4 Assembly

How do we add up more than 8 numbers?

Step 4: Use ALU to increment REG holding DATA Address

```
ADD R0, R0, #1
```



R0	x4001
R1	1
R2	0
R3	1
R4	0
R5	0
R6	0
R7	0

Register File

Address	Contents
x4000	1
x4001	2
x4002	3
x4003	4
x4004	5
x4005	6
x4006	7
x4007	8
x4008	9
x4009	10

DATA Memory

Working with DATA Memory in LC-4 Assembly

How do we add up more than 8 numbers?

Repeat Step 2: Load R3 with DATA from x4001

```
LDR R3, R0, #0      ; offset 0
```

R0	x4001
R1	1
R2	0
R3	1
R4	0
R5	0
R6	0
R7	0

Register File

Address	Contents
x4000	1
x4001	2
x4002	3
x4003	4
x4004	5
x4005	6
x4006	7
x4007	8
x4008	9
x4009	10

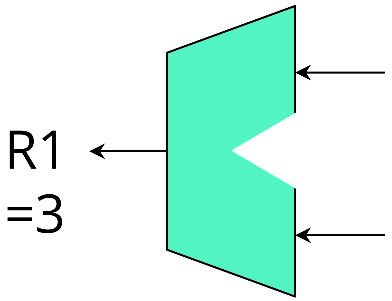
DATA Memory

Working with DATA Memory in LC-4 Assembly

How do we add up more than 8 numbers?

Repeat Step 3: Use ALU to add 1st # to total

```
ADD R1, R1, R3      ; offset 0
```



R0	x4001
R1	1
R2	0
R3	2
R4	0
R5	0
R6	0
R7	0

Register File

Address	Contents
x4000	1
x4001	2
x4002	3
x4003	4
x4004	5
x4005	6
x4006	7
x4007	8
x4008	9
x4009	10

DATA Memory

Working with DATA Memory in LC-4 Assembly

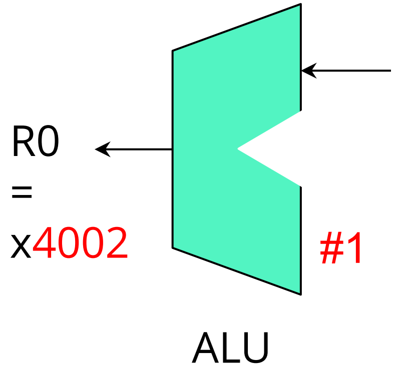
How do we add up more than 8 numbers?

Repeat Step 4: Use ALU to increment REG holding DATA Address

```
ADD R0, R0, #1
```

; offset 0

And keep on going!



R0	x4001
R1	3
R2	0
R3	0
R4	0
R5	0
R6	0
R7	0

Register File

Address	Contents
x4000	1
x4001	2
x4002	3
x4003	4
x4004	5
x4005	6
x4006	7
x4007	8
x4008	9
x4009	10

DATA Memory

POINTER IN ASSEMBLY

A Word on "Pointers"

- This process of storing the address of a memory location in a variable (in this case a register) and then using that address to access the memory location is very common in assembly
- This address variable is referred to as a pointer and we will see the concept reappear in C
  - *A pointer is a 'variable' that holds a memory address*
  - *EX:* `R0=x4000` *(R0 holds an address in data memory)*
    - *R0 "points" to an address in data memory*
- The act of using a pointer value to read or write a memory location
  - *Referred to as dereferencing the pointer*
  - EX: `LDR R3, R0, #0`
    - ➢ *Loads DATA from address pointed to by R0, into R3*
    - ➢ *R0 will still hold address x4000*
    - ➢ *Now R3 holds the "dereferenced data" pointed to by R0 x4000*

HOW DO WE INITIALLY LOAD DATA IN TO DATA MEMORY?

Working with DATA Memory in LC-4 Assembly

Some outstanding questions in our Adding Program:

- How do we initially load DATA into DATA Memory?
  - We'll use three assembly "directives" (not instructions): .DATA .ADDR and .FILL
    - ➢ Assembly Directives provide an indication to the assembler of where it should place various blocks of code or data

Assembly:

```
.DATA
.ADDR x4000
.FILL #1
.FILL #2
.FILL #3
.FILL #4
.FILL #5
```

Address	Contents
x4000	1
x4001	2
x4002	3
x4003	4
x4004	5
…	…

LC-4 Assembly Directives

Assembly Directives provide an indication to the **assembler** of where it should place various blocks of code or data

.DATA

- Next values are in DATA Memory

.ADDR

- Set current address to the specified value

.FILL IMM16

- Set value at the current address to the specified 16 bit value

Ultimately the assembly program gets assembled to an **object file** which is a specification for how the machine memory should be set up

Program to Add Up 10 Numbers (Assembly) – P1

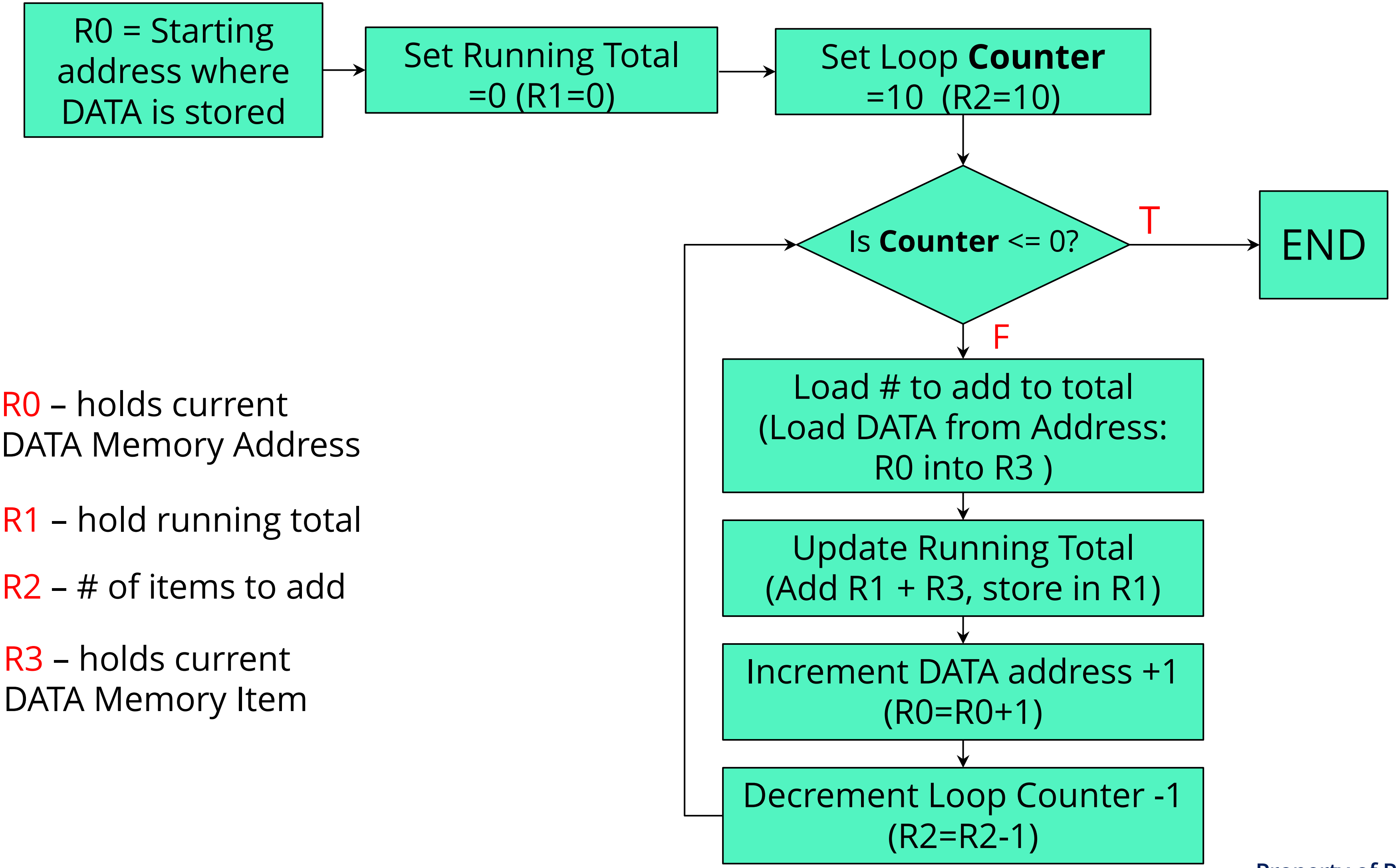
```
        .DATA                   ; lines below are DATA memory
        .ADDR x4000             ; where to start in DATA memory

global_array                    ; label address x4000: global_array
        .FILL #1                ;
        .FILL #2                ;
        .FILL #3                ;
        .FILL #4                ;
        .FILL #5                ;
        .FILL #6                ;
        .FILL #7                ;
        .FILL #8                ;
        .FILL #9                ;
        .FILL #10               ;
```

Address	Data Mem Contents
global_array	1
x4001	2
x4002	3
x4003	4
x4004	5
x4005	6
x4006	7
x4007	8
x4008	9
x4009	10

HOW DO WE INITIALLY LOAD DATA IN TO DATA MEMORY?

Program to Add Up 10 Numbers (Flow Chart)



R0 – holds current DATA Memory Address

R1 – hold running total

R2 – # of items to add

R3 – holds current DATA Memory Item

What type of loop should we use?

What type of loop is best?

- We could use a while loop (we've done this already), but why not a for loop?
  - We need a loop that repeats an exact # of times
    - ➢ *that's exactly what a for loop is designed to do!*

Initial value of counter — *Condition to end loop* — *What to do each time through loop*

```
for (loop_count=10 ; loop_count>0 ; loop_count--)  {
              // load data from memory
              // add loaded data to running total
              // increment data memory address
}
```

Program to Add Up 10 Numbers (Assembly) – P1

```
    .DATA                   ; lines below are DATA memory
    .ADDR x4000             ; where to start in DATA memory

global_array                ; label address x4000: global_array
    .FILL #1                        ; x4000 = 1
    .FILL #2                        ; x4001 = 2
    .FILL #3                        ; x4002 = 3
    .FILL #4                        ; x4003 = 4
    .FILL #5                        ; x4004 = 5
    .FILL #6                        ; x4005 = 6
    .FILL #7                        ; x4006 = 7
    .FILL #8                        ; x4007 = 8
    .FILL #9                        ; x4008 = 9
    .FILL #10                       ; x4009 = 10
```

Program to Add Up 10 Numbers (Assembly) – P2

```
  .CODE                  ; lines below are Program Memory
  .ADDR x0000            ; where to start in Program Memory

INIT
  LEA R0, global_array ; load starting address of data to R0
                         ; same as HICONST=x40, CONST=x00
  CONST R1, #0           ; R1 = running total
  CONST R2, #10          ; R2 = # of items to add (aka – loop counter)

FOR_LOOP
  CMPI R2, #0            ; subtract 0 from the loop counter (R2), set NZP
  BRnz END               ; if R2 is <=0, done, jump to END
  LDR R3, R0, #0                 ; LOAD the data (at R0) into R3
  ADD R1, R1, R3                 ; update running total in R1
  ADD R0, R0, #1                 ; add 1 to address in R0
  ADD R2, R2, #-1                ; reduce for loop counter by 1
  JMP FOR_LOOP           ; could also use: BRnzp LOOP
END
```