Microcomputers I – CE 320

Mohammad Ghamari, Ph.D.
Electrical and Computer Engineering
Kettering University

Announcements

- Do not forget, you will have a quiz on Thursday!

Lecture 8: Assembly Language

Today’s Topics

- Review the concept of memories and registers (accumulators)

- How to generate machine code manually.
  - You are expected to convert assembly code lines to machine codes.

- Files and processes associated with converting assembly source code to machine code

- To learn assembler directives

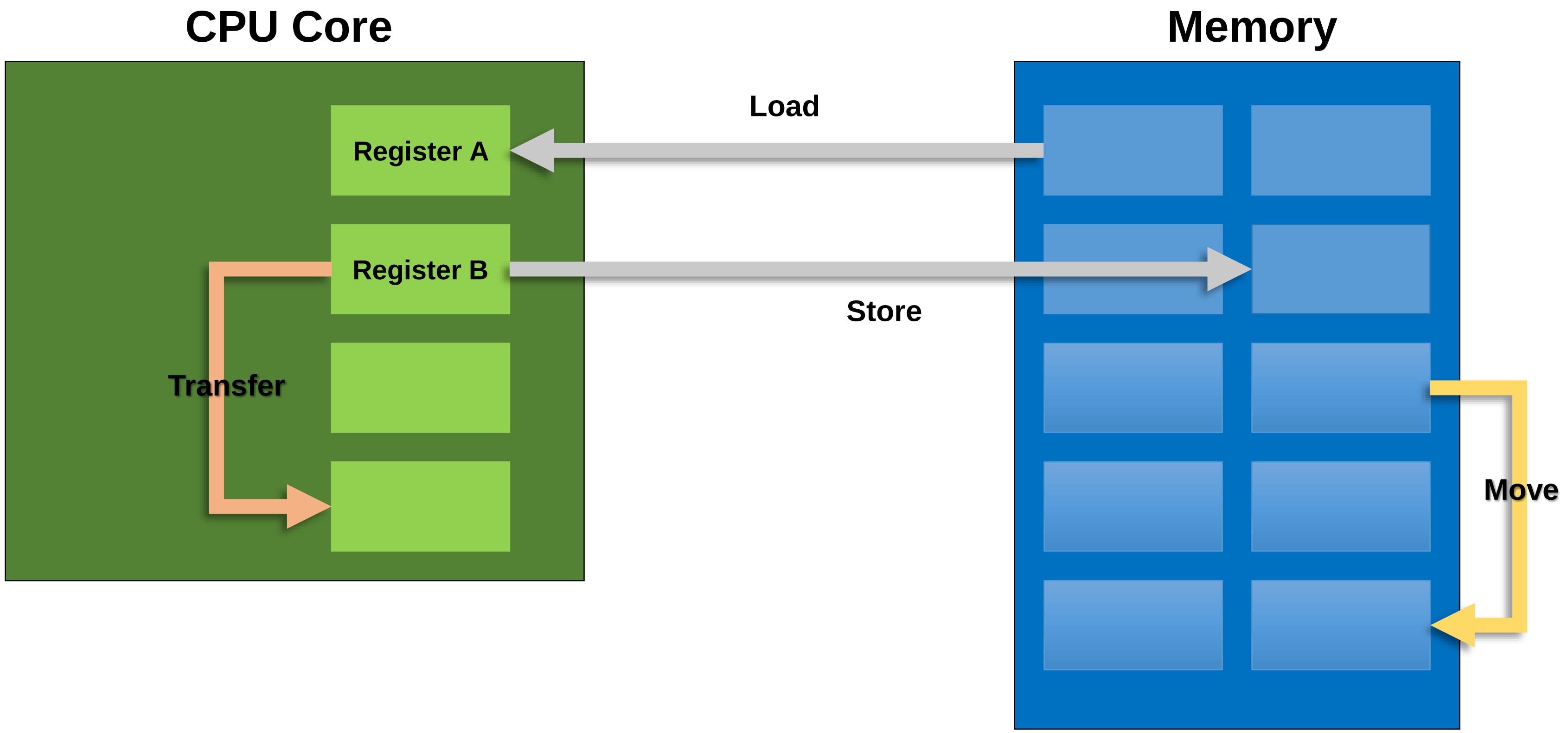
Microcontrollers (or Microcomputers)

Basic ideas

- Microcontroller
  - CPU core + I/O ports + Memories (RAM and ROM) + …
- Memories
  - We only use RAM area to learn assembly language and test programs.
  - No need to worry about burning your program into ROM.
- Registers (accumulators) vs. memories
  - Registers are small read/write memory cells inside CPU core.
  - Memories are located outside CPU.
  - To get a value from a location in a memory, the value should travel through data bus. (Remember memory modules are separated from CPU core)
    - This takes time (it is much longer than getting from/setting to Registers)

Microcontrollers (or Microcomputers)

Load and Store / Move / Transfer



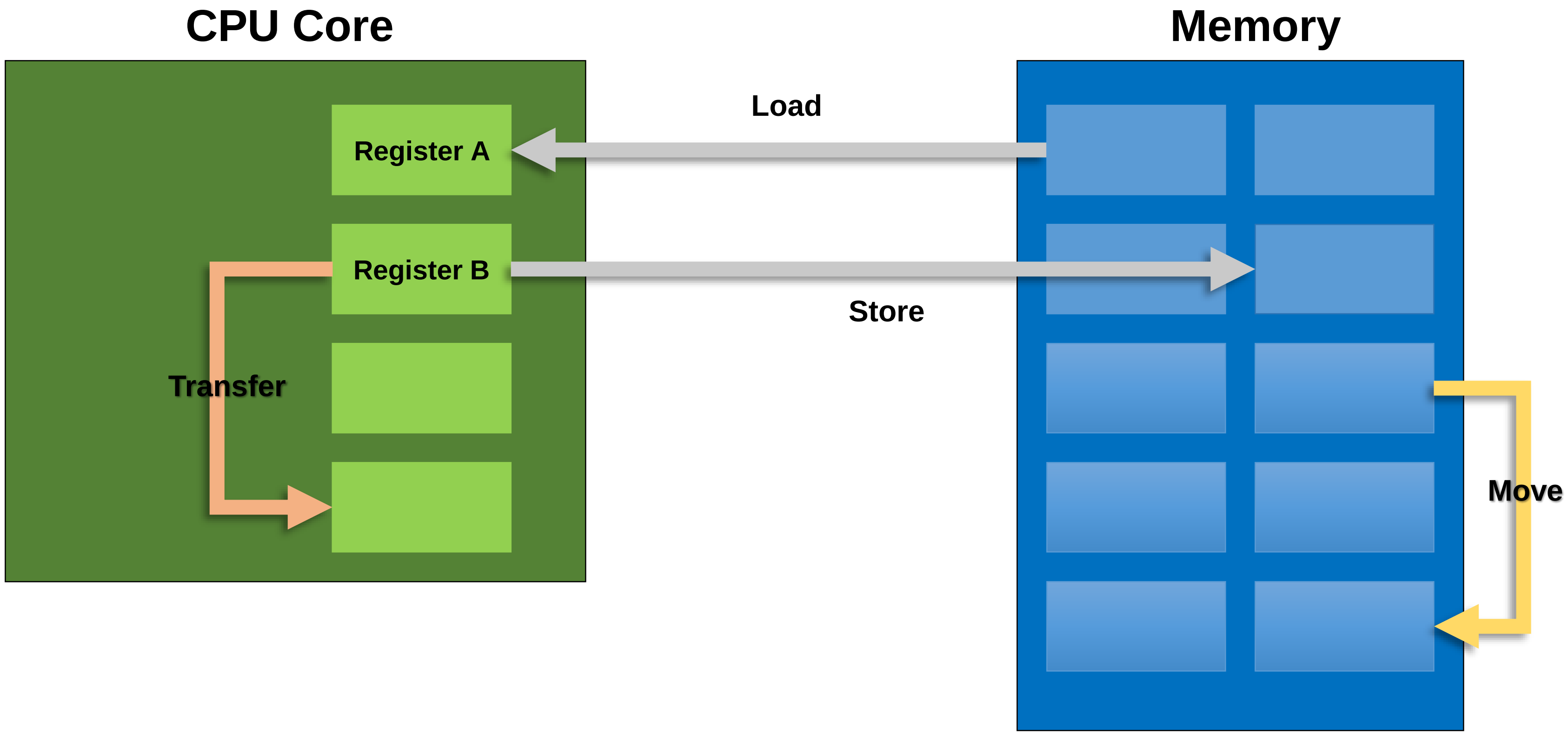
Before doing arithmetic operations including comparisons, the microcontroller requires a value on a register to do the operations.

Microcontrollers (or Microcomputers)

Load and Store / Move / Transfer

As a programmer, you have two main tools for now (bunch of other things will come soon); a storage to save data; and a process unit to manipulate the data to conduct arithmetic operations and logical decisions.

We write programs to control the processor and manipulate the memory.



Code Line and Program Counter

- When we say **Code Line** in Lab assignments and quizzes, the instruction line is completed (executed).
  - So registers are supposed to be affected by the execution.
- **Program Counter** always points the NEXT instruction!!
  - Caution on Branches. PC depends on whether the branch is taken or not.
- Example:

```
1:   1500   CE 2000        LDX #$2000
2:   1503   180B FF 1000   MOVB #$FF,$1000
3:   1508   C6 02          LDAB #2
4:   150A   27 0E          BEQ 14
5:   150C   A6 00          LDAA 0,X
6:   150E   B1 1000        CMPA $1000
7:   1511   24 03          BHS 3
8:   1513   7A 1000        STAA $1000
9:   1516   08             INX
10:  1517   53             DECB
11:  1518   20 F0          BRA -16
12:  151A   3F             SWI
```

Trace	Line	PC	A	B	X	N	Z	V	C
1	1	1503	-	-	2000	0	0	0	-
2	2	1508	-	-	2000	0	0	0	-
3	3	150A	-	02	2000	0	0	0	-
4	4	150C	-	02	2000	0	0	0	-
5	5	150E	40	02	2000	0	0	0	-
6	6	1511	40	02	2000	0	0	0	1
7	7	**1513**	40	02	2000	0	0	0	1
8	8	1516	40	02	2000	0	0	0	1
9	9	1517	40	02	2001	0	0	0	1
10	10	1518	40	01	2001	0	0	0	1

Machine Code

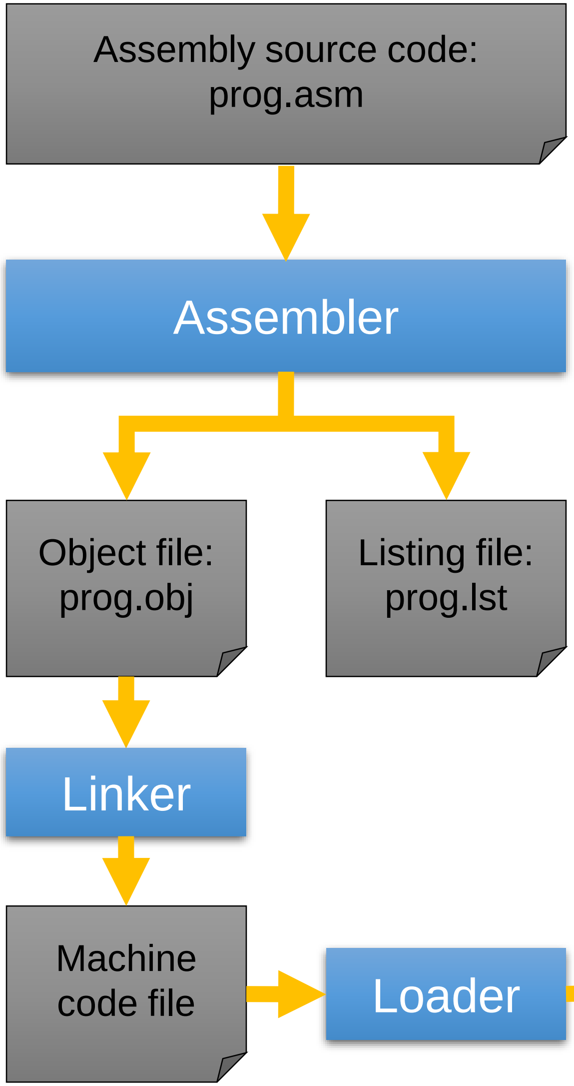
Manually generate machine code*

Source Form	Operation	Addr. Mode	Machine Coding (hex)	S X H I	N Z V C
LDX #*opr16i*	(M:M+1) ⇒ X	IMM	CE jj kk	– – – –	Δ Δ 0 –
LDX *opr8a*	Load Index Register X	DIR	DE dd		
LDX *opr16a*		EXT	FE hh ll		
LDX *oprx0_xysp*		IDX	EE xb		
LDX *oprx9,xysp*		IDX1	EE xb ff		
LDX *oprx16,xysp*		IDX2	EE xb ee ff		
LDX [D,*xysp*]		[D,IDX]	EE xb		
LDX [*oprx16,xysp*]		[IDX2]	EE xb ee ff		

Address	Machine Code	Source Code
1500	CE 2000	LDX #$2000
1503	180B FF 1000	MOVB #$FF, $1000
1508	C6 02	LDAB #2
150A	27 0E	BEQ 14
150C	A6 00	LDAA 0,X
150E	B1 1000	CMPA $1000
1511	24 03	BHS 3
1513	7A 1000	STAA $1000
1516	08	INX
1517	53	DECB
1518	20 F0	BRA -16
151A	3F	SWI

Assembly Process

General case



ASCII text file
.asm extension

Cross-assembler:
Translate assembly code into object code.

Object code:
Mix of machine code and additional information

Listing file:
Human readable log file containing the original assembly and the machine code.

Linker:
Combines multiple object files into a single piece of machine code

Proper Assembly Code

- Separate the source code into **constant section**, **data and variable sections**, and **code section**.
- Do not use numbers within the code
  - Except for possibly 0 or 1 in obvious situation
- Always begin with a comment block stating
  - Purpose of the program
  - Inputs
  - Outputs
  - Programmer
  - Anything else useful
- Comment within the code
  - Assume that a reader understands the processor’s assembly code, so do not use comments to simply rephrase the assembly code.

Assembly Language Program Structure

- HCS12 assembly program consists of **three sections**:
  - **Assembler directives**
    - **Command to the assembler** **(not executable by microprocessor)** to process subsequent assembly language instructions.
    - Also provide a way to define **program constants** and **reserve space for dynamic variables**.
    - Some directives may also set a location counter.
  - **Assembly language instructions**
    - These instructions are HCS12 instructions.
    - Some instructions are defined with labels.
  - **Comments**
    - There are two types of comments:
      - The first type is used to explain the function of a single instruction or directive.
      - The second type explains the function of a group of instructions or directives or a whole routine.

Assembly Language Program Structure

- **Each line** of a HCS12 assembly program is comprised of **four** distinct fields:

  1. **Label**
  2. **Operation**
  3. **Operand**
  4. **Comment**

Some of the fields may be empty in a line.

But the order of these fields is very important in each line.

[label:] [command] [operand(s)] [;comment]
or
; comment

where '**:**' indicates the end of label and '**;**' defines the start of a comment.
The *command* field may be an instruction, a directive or a macro call.

Assembly Language Program Structure

Label Field

- Labels are **symbols defined by the user to identify memory locations** in the programs and data areas of the assembly module.

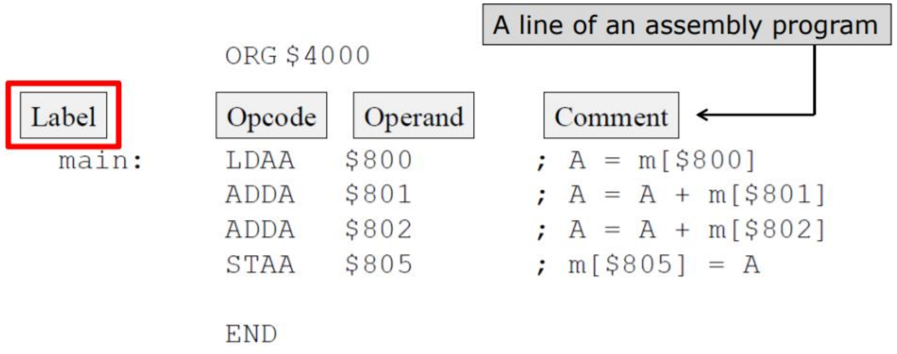
- For most instructions and assembler directives, the label is **optional**.

- The rules for forming a label are as follows:

  - A label must start at **column one** and begin with a letter (A-Z, a-z), and the letter can be followed by letters, digits, or special symbols.

  - The asHCS12 assembler allows a label to be terminated by “**:**”

Sample Program:



Example of **valid** labels:

begin ldaa #10 ; label begins in column 1

print: jsr hexout ; label is terminated by a colon

Example of **invalid** labels:

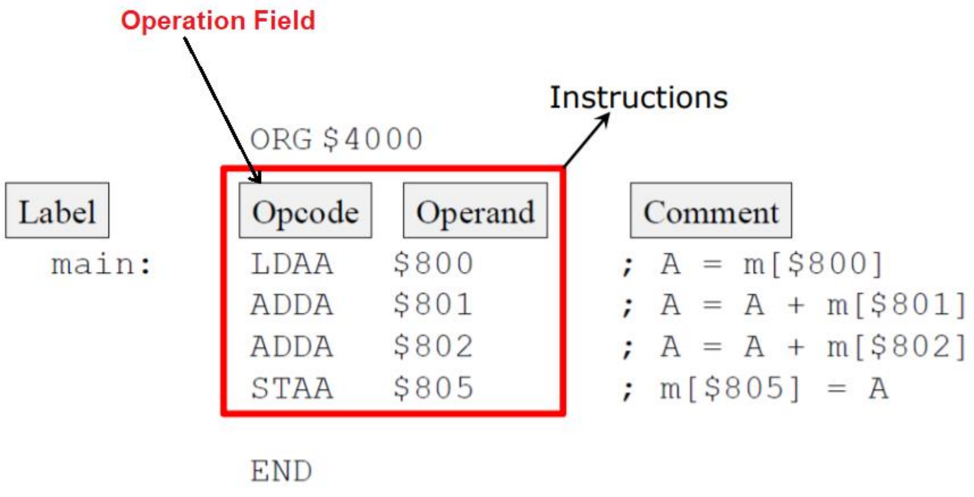
here is adda #5 ; a space is included in the label

Assembly Language Program Structure

Operation Field

- This field contains the mnemonic names for **machine instructions** and **assembler directives**.
  - If a label is present, the opcode or directive must be separated from the label field by at least one space.
  - If there is no label, the operation field must be at least one space from the left margin.

Sample Program:



Examples of operation fields:

```
adda    #$02          ; adda is the instruction mnemonic
true    equ     1     ; equate directive equ occupies the operation field
```

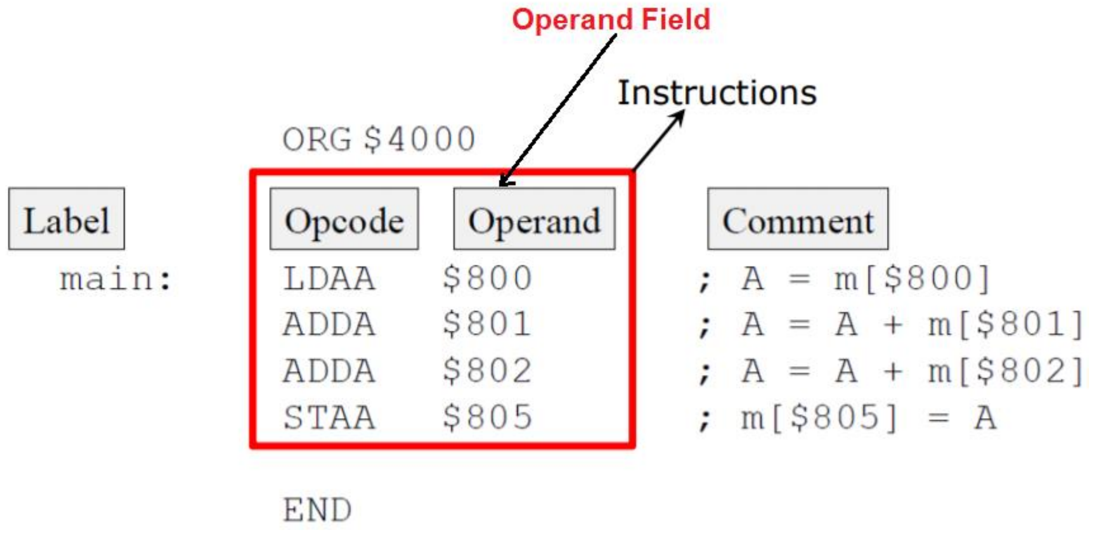
Assembly Language Program Structure

Operand Field

- It follows the operation field and is separated from the operation field by at least one space.

- The operand field may contain **operands for instructions** or **arguments for assembler directives**.

Sample Program:



Examples of operand fields:

```
TCNT    equu    $0084          ; $0084 is the operand field
TC0     equ     $0090          ; $0090 is the operand field
```

Assembly Language Program Structure

Comment Field

- Is optional and is added mainly for documentation purposes.
- It is ignored by the assembler.
- Rules for comments:
  - Any line beginning with an * (asterisk) is a comment.
  - Any line beginning with a ; (semi-colon) is a comment.

Sample Program:

Label	Opcode	Operand	Comment
	ORG $4000		
main:	LDAA	$800	; A = m[$800]
	ADDA	$801	; A = A + m[$801]
	ADDA	$802	; A = A + m[$802]
	STAA	$805	; m[$805] = A
	END		

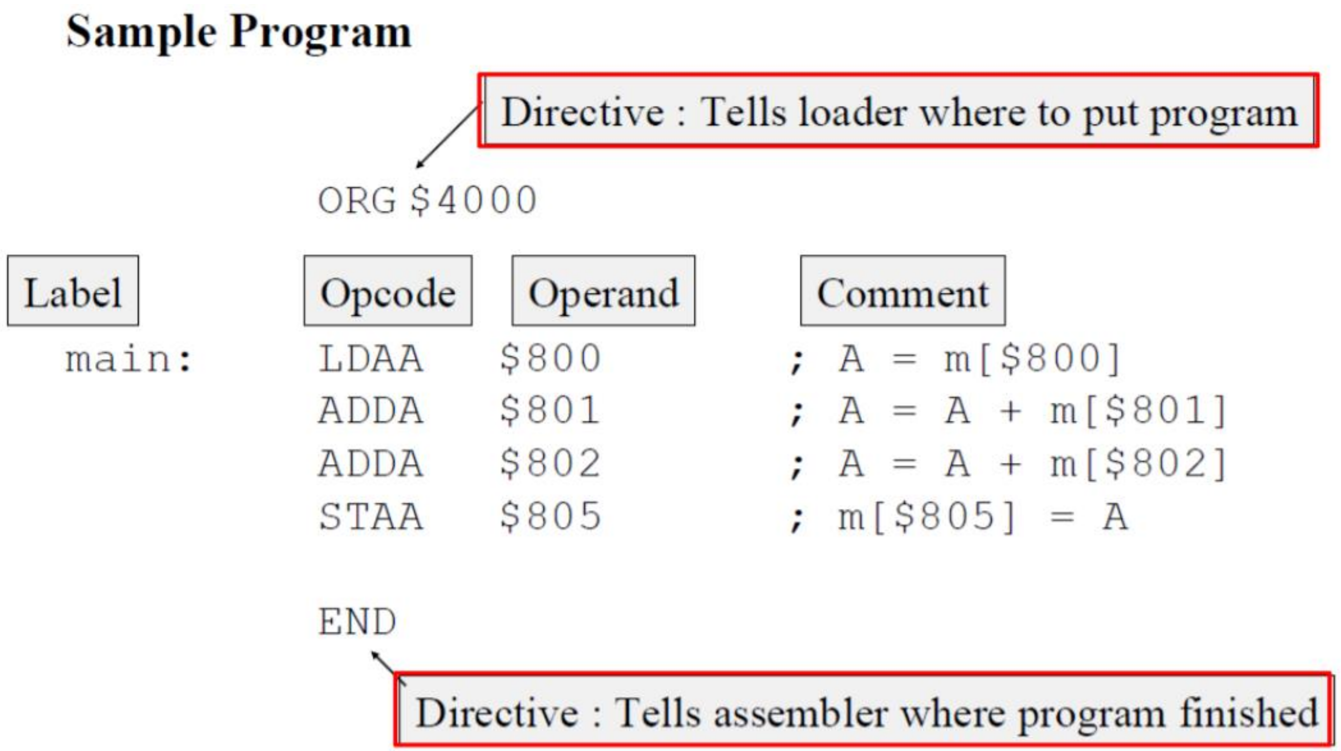
Examples of comment fields:

```
; this program computes the square root of N 8-bit integers.

org     $1000          ; set the location counter to $1000
dec     lp_cnt         ; decrement the loop count
```

Assembler Directives

- Look just like instructions in an assembly language program
- But they tell **assembler** to do something other than creating the machine code for an instruction
- Define program constants and reserve space for dynamic variable
- Specifies the end of a program



Assembler Directives

end directive

- The end directive is used to end a program to be processed by the assembler.

- In general, an assembly program looks like this:

```
(your program)
end
```

- Any statement following the end directive is ignored.

- A warning message will be raised if the end directive is missing from the source code; however, the program will still be assembled correctly.

Assembler Directives

org (origin) directive

- The assembler uses a **location counter** to keep track of the memory location where the next machine code byte should be placed.

- If the programmer wants to force the *program* or *data array* to start from a certain memory location, then **org** directive can be used.
  - For example, the statement:

```
org     $1000   ;forces the location counter to be set to $1000
ldab    #$FF    ;this instruction will be stored in memory starting
                 from location $1000.
```

Assembler Directives

db, dc.b and fcb directives

db (define byte)
dc.b (define constant byte)
fcb (form constant byte)

- These three directives define the value of a byte or bytes that will be placed at a given memory location.
- They assigns the value of the expression to the memory location pointed to by the location counter. Then the location counter is incremented.

For example, the statement:

```
array    db    $11,$22,$33,$44,$55
```

initializes five bytes in memory to:

$11
$22
$33
$44
$55

and the assembler will use **array** as the **symbolic address of the first byte** whose initial value is $11.

Assembler Directives

db, dc.b and fcb directives

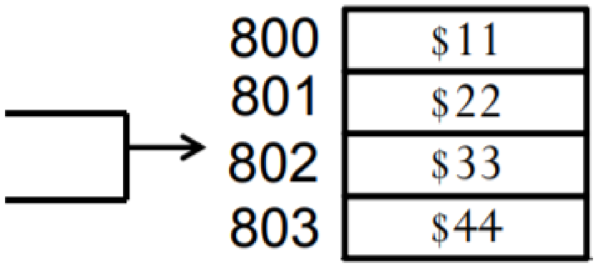
db (define byte)

dc.b (define constant byte)

fcb (form constant byte)

- The program can also force these five bytes to a particular address by adding the **org** directive. For example, the sequence:

```
        org     $800
array   dc.b    $11,$22,$33,$44
```



Assembler Directives

dw, dc.w and fdb directives

dw (define word)

dc.w (define constant word)

fdb (form double bytes)

- These three directives define the value of a word or words that will be placed at a given address.
- For example:

```
        org     $800
array   dc.w    $AC11,$F122,$33,$F44
```

800	$AC
801	$11
802	$F1
803	$22
804	$00
805	$33
806	$0F
807	$44

Assembler Directives

fcc directive

fcc (form constant character)

- This directive allows us to define a **string of characters (a message).**
- The first character in the string is used as the delimiter.
- The last character must be the same as the first character because it will be used as the delimiter.
- The delimiter must not appear in the string.
- The space character cannot be used as the delimiter.
- Each character is encoded by its corresponding **ASCII code**.

- For example:

```
        org   $1000
Alpha   fcc   “def”
```

1000	$64
1001	$65
1002	$66

  - Assembler will convert to Ascii

ASCII, (American Standard Code for Information Interchange)

ASCII is a character encoding standard

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

Assembler Directives

fill directive

fill (fill memory)

- This directive allows a user to fill a certain number of memory locations with a given value.
- The syntax of this directive is as follows:

fill value, count

  - where the number of bytes to be filled is indicated by count
  - and the value to be filled is indicated by value.

For example, the statement:

space_line fill $20, 40

will fill 40 bytes with the value of $20 starting from the memory location referred to by the label **space_line**.

Assembler Directives

ds, rmb, ds.b directives

ds (define storage)

rmb (reserve memory byte)

ds.b (define storage bytes)

- Each of these three directives reserves a number of bytes for later use.

Example: **buffer ds 100**

reserves 100 bytes starting from the location represented by buffer - none of these locations is initialized

Assembler Directives

ds.w, rmw directives

ds.w (define storage word)

rmw (reserve memory word)

- Each of these directives reserves a number of words for later use.

Example: **Dbuf ds.w 20** ;Reserves 20 words (or 40 bytes) starting from the current location counter

Assembler Directives

equ directive

equ (equate)

- This directive assigns a value to a label.
- Using **equ** to define constants will make our program more readable.
- For example, the statement:

loop_cnt equ 40

informs the assembler that whenever the symbol **loop_cnt** is encountered, it should be replaced with the value of 40.

Example 1: Array of bytes

```
org $800
a1 db $11, $22, $33, $44
a2 dc.b $01
   dc.b $02
   dc.b $03
a3 rmb 2
```

Memory Map

800	11
801	22
802	33
803	44
804	01
805	02
806	03
807	?
808	?

Example 2: Array of words

```
org $800
a1 dw $11, $22, $33, $44
a2 dc.w $01
   dc.w $02
   dc.w $03
a3 rmw 2
```

Memory Map

800	00
801	11
802	00
803	22
804	00
805	33
806	00
807	44
808	00
809	01
80A	00
80B	02
80C	00
	...

Some More Instructions

Load Effective Address Instructions

- Load effective address instructions
  - LEAX: Load effective address into X
    - LEAX   10,X
  - LEAY: Load effective address into Y
    - LEAY   B, Y
  - LEAS: Load effective address into SP
    - LEAS   0,PC

- Can you tell what is the meaning of “LEAX 10,X”?
  - Assuming (X) = 1200, the content at 120A is 34h, and at 120B is 56h
    - “LEAX 10,X” **makes X be 120A** (**X= the address (not the content)**= X+10)
      - Address of X=1200, LEAX 10,X means X+10, thus 1200+A=120A

Some More Instructions

Addition and Subtraction

- 8 bit addition
  - ABA: (A) + (B) → A; Note that there is no AAB instruction!
  - ADDA: (A) + (M) → A
    - ADDA $1000
  - ADDB: (B) + (M) → B
    - ADDB #10
  - ADCA: (A) + (M) + C → A **Add with carry to A**
  - ADCB: (B) + (M) + C → B
- 8 bit subtraction
  - SBA: (A) – (B) → A; Subtract B from A (Note: not SAB instruction!)
  - SUBA: (A) – (M) → A; Subtract M from A
  - SUBB: (B) – (M) → B
  - SBCA: (A) – (M) – C → A **Subtract with Borrow from A**
  - SBCB: (B) – (M) – C → B
- 16 bit addition and subtraction
  - ADDD: (A:B) + (M:M+1) → A:B
  - SUBD: (A:B) – (M:M+1) → A:B
  - ABX: (B) + (X) → X
  - ABY: (B) + (Y) → Y

Some More Instructions

Increments, Decrements, and Negate

- Increments
  - INC: (M) + 1 → M — **Increment Memory Byte**
  - INCA: (A) + 1 → A — **Increment Acc. A**
  - INCB
  - INS
  - INX
  - INY
- Decrements
  - DEC
  - DECA
  - DECB
  - DES
  - DEX
  - DEY
- Negate
  - NEG: negate a memory byte
  - NEGA
  - NEGB

Homework Example

- Write a program to copy a table of one-byte values.

- Your table will be defined by a starting address, supplied at $1000, and by a one-byte number of elements in the table, supplied at $1002.

- The table will be copied a given distance from the original table, and this two-byte offset will be supplied at address $1003.

```
        =00001000                        ORG    $1000
     1000 +0002               table      ds.w   1
     1002 +0001               length     ds.b   1
     1003 +0002               offset     ds.w   1

        =00001800                        ORG    $1800
     1800 FE 1000                        LDX    table
     1803 B7 54                          TFR    X,D
     1805 F3 1003                        ADDD   offset
     1808 B7 46                          TFR    D,Y
     180A F6 1002                        LDAB   length
     180D 27 09               loop       BEQ    done
     180F 180A 00 40                     MOVB   0,X,0,Y
     1813 08                             INX
     1814 02                             INY
     1815 53                             DECB
     1816 20 F5                          BRA    loop
     1818 3F                  done       SWI
Symbols:
done                          *00001818
length                        *00001002
loop                               *0000180d
offset                        *00001003
table                         *00001000
```

Just one example

…	
1000	20
1001	00
1002	40
1003	05
1004	00
…	
2000	12
2001	34
2002	56
2003	78
2004	55
…	
2500	
2501	
2502	
2503	
2504	
…	

Modification of the Example

- What changes are required to handle a table of two-byte numbers?
  - Need to copy two bytes instead of one byte.

- What changes are required to handle a two-byte length?
  - The length should represent two-byte numbers!

```
        =00001000                 ORG     $1000
    1000 +0002          table     ds.w    1
    1002 +0001          length    ds.b    1
    1003 +0002          offset    ds.w    1

        =00001800                 ORG     $1800
    1800 FE 1000                  LDX     table
    1803 B7 54                    TFR     X,D
    1805 F3 1003                  ADDD    offset
    1808 B7 46                    TFR     D,Y
    180A F6 1002                  LDAB    length
    180D 27 09          loop      BEQ     done
    180F 180A 00 40               MOVB    0,X,0,Y
    1813 08                       INX
    1814 02                       INY
    1815 53                       DECB
    1816 20 F5                    BRA     loop
    1818 3F             done      SWI

Symbols:
done                  *00001818
length                *00001002
loop                  *0000180d
offset                *00001003
table                 *00001000
```

length ds.w 1

LDD length

MOVW 0,X,0,Y

Add additional **INX** instruction

Add additional **INY** instruction

SUBD #1

Questions?

Wrap-up

What we’ve learned

- Registers and memories
- Generating machine code manually.
- Concept of assembly language
- Assembler directives

What to Come

- Flowcharts
- Some assembly programming examples