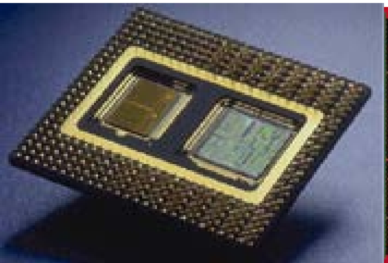
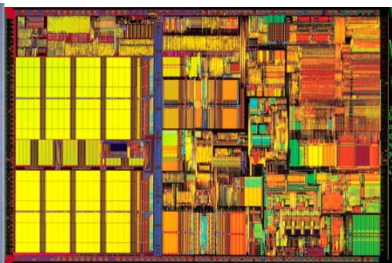
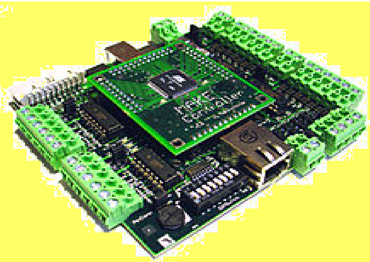
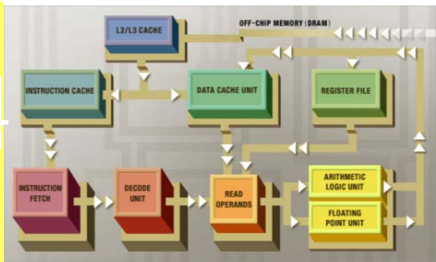
Lecture 5
Assembly Programming: Arithmetic

Bryan Ackland
Department of Electrical and Computer Engineering
Stevens Institute of Technology
Hoboken, NJ 07030

Adapted from HCS12/9S12 An Introduction to Software and Hardware Interfacing   Han-Way Huang, 2010

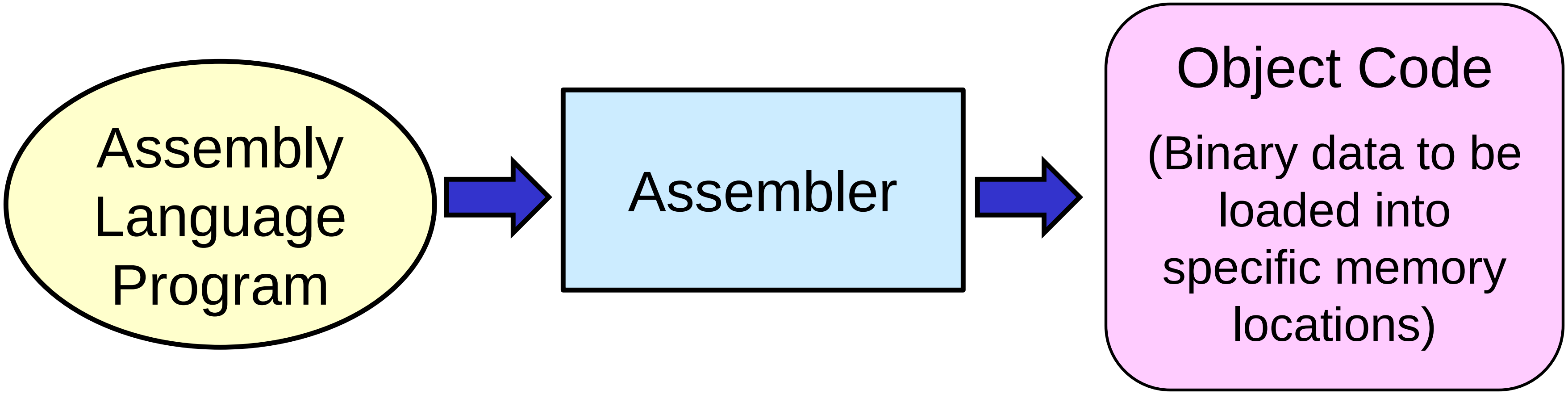



Try These…

1. What is 27_{10} in 8-bit binary?
2. What is -27_{10} in 8-bit binary?
3. What is %10011010 (unsigned) in decimal?
4. What is %10011010 (signed) in decimal?
5. What is %10011010 in hex?
6. What is %10101101 + %00100111 in binary (unsigned)
7. What is %10101101 + %00100111 in binary (signed)
8. What is 299_{10} in 16-bit hex?
9. What is $1A3F in decimal?
10. What is $39C2 + $A175 in hex?

What is Assembly Language?

- Assembly Language (assembly code) allows a programmer to specify machine code *instructions* and *data* that should be loaded into microprocessor memory prior to program execution.
  - Machine code *instructions* are specified using mnemonics and address labels
  - *Data* represents initial values of program variables



- Assembler translates assembly code mnemonics & symbols into raw binary data to be loaded into microprocessor memory

Structure of a HCS12 Assembly Program

You will find three kinds of statements in assembly program:

- Assembler Directives
  - Define data and symbols
  - Reserve and initialize memory locations
  - Set assembler and linking conditions
  - Specify output format
  - Specify end of program
- Assembly Language Instructions
  - mnemonic representation of HCS12 machine code instructions
- Comments
  - Explanation and documentation

Program Structure: Example Code

assembler directive

label

```
                ORG     $800
results:        DS.B    4               ;reserve 4 bytes for result
inc_value:      EQU     $30             ;symbol to represent data
din:            DC.B    $1122           ;label & initialize data
```

comment

assembly language instruction

```
                ORG     $900            ;program begins
                ldd     din             ;load $1122 into D
                subd    #10
                std     results
                adda    #inc_value
                std     results+2
                END                     ;program ends
```

opcode *operand*

Fields of an HCS12 Instruction

loop: adda #$40 ;add $40 to accumulator A

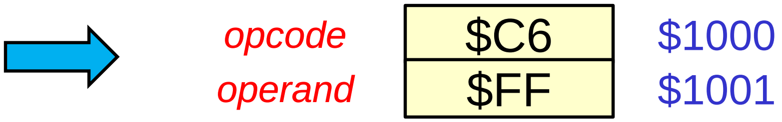
- **Label Field**
  - optional: usually starts from first column
  - start with a letter followed by letters, digits or (_ or **.**)
  - can start any column if ended with a colon **:**
- **Operation (Opcode) Field**
  - mnemonic machine code instructions or assembler directive
  - is separated from label or beginning of line by at least one space
- **Operand Field**
  - operands for instructions or arguments for assembler directives
  - separated from operation field by at least one space
- **Comment Field**
  - optional: starts with **;**
  - separated from operation/operand field by at least one space
  - a line that starts with ***** or **;** is a comment

Some Assembler Directives

- **END**
  - Ends program to be processed by assembler
  - Any statement after **END** is ignored

- **ORG**
  - Assembler uses a location counter to keep track of current memory location
    - where next machine code byte or data byte should be placed
  - ORG directive sets a new value into the location counter
  - for example:

```
ORG     $1000
ldab    #$FF
```



will place the opcode byte for the "ldab" instruction at memory address $1000

Initialize Memory Directives

- **DC.B** (define constant byte)
  - define value of byte (or bytes) at current memory location
  - location counter is updated to point to next byte address
  - value can be specified by expression

- **DC.W** (define constant word)
  - define value of 2-byte word(s)

```
        ORG     $4000
array:  DC.B    $A3
        DC.B    $11, $22, $33
        DC.W    $1234, array-2
```

$A3	$4000
$11	$4001
$22	$4002
$33	$4003
$12	$4004
$34	$4005
$3F	$4006
$FE	$4007

Initialize Text String

- **DC.B** can also be used to define and load a string of ascii characters
  - string specified using quotes (“”)
  - each character represented by one-byte ascii code

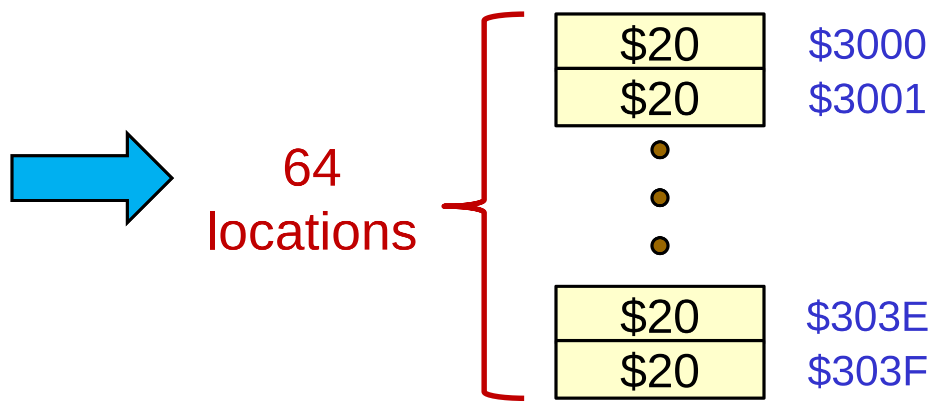
```
        ORG     $1000
msg:    DC.B    “enter y/n:”
```

$1000	$65	e
$1001	$6E	n
$1002	$74	t
$1003	$65	e
$1004	$72	r
$1005	$20	<sp>
$1006	$79	y
$1007	$2F	/
$1008	$6E	n
$1009	$3A	:

Initialize Memory Directives

- **DCB.B** (define constant block of bytes)
  - fill a block of memory locations with same one-byte value
  - syntax is: **DCB.B count ,value**
  - **value** is optional – default value is $00
- **DCB.W** (define constant block of words)
  - fill a block of memory with same two-byte value

```
            ORG     $3000
space_line: DCB.B   64, $20
```



Reserve Memory Directives

- **DS.B** (define storage byte)
  - reserves (and optionally labels) number of bytes at current memory location
  - location counter is updated to point to next byte address following the reserved space
  - reserved locations are not initialized

- **DS.W** (define storage word**)**
  - reserves, and optionally labels: (# words X 2) bytes

```
        ORG     $1400
buffer: DS.B    $100    ;reserves 256 bytes of memory from $1400
                        ;to $14FF with "buffer" labeling first byte
 wbuf:  DS.W    20      ;reserves 20 words (40 bytes) of memory from
                        ;$1500 to $1527 with "wbuf" labeling first byte
```

Equate Directive

- **EQU** (equate)
  - assigns a value (rather than a memory address) to a label
  - does not affect memory contents

```
loop_count:     EQU     16
                …
                ldaa    #loop_count     ;load 16 into accumulator A
```

Assembler Directive Examples

- Show the contents of memory resulting from the following assembler directives:

```
            ORG     $4800
xyz:        EQU     24
abc:        DC.B    $20, 16
            DC.W    $21, $1ACD

res:        DS.B    3
            DC.B    “bcd”
            DC.W    abc+xyz
```

- Show results as a table:

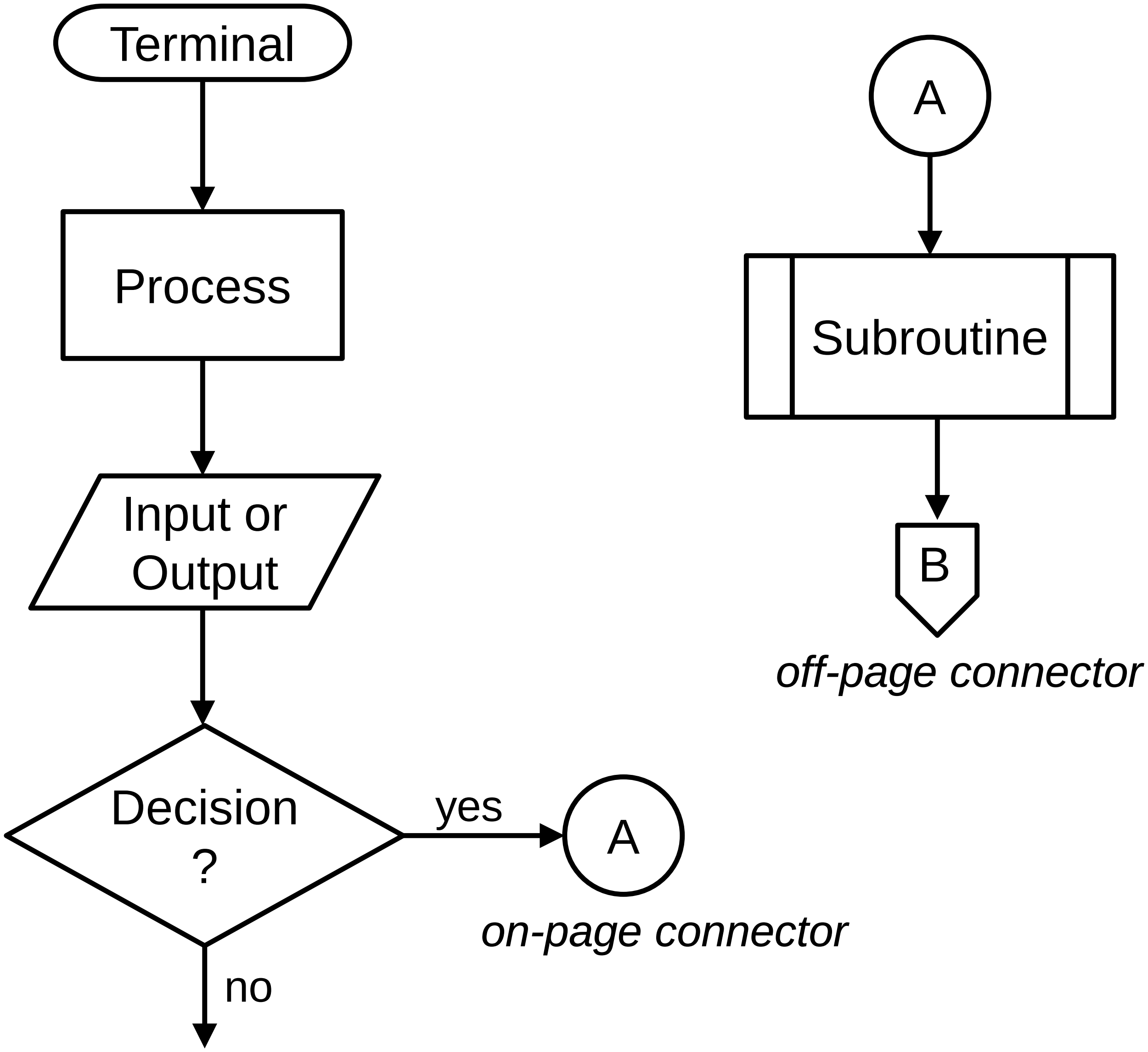
label	address	data

  - ascii code for ‘b’ is $62, ascii code for ‘c’ is $63, ascii code for ‘d’ is $64

Software Development Process

- Problem definition:
  - Identify precisely what needs to be done
- Develop a plan or algorithm
  - computational procedure that takes a set of inputs and produces required outputs
  - may be expressed as a set of steps that need to be performed
  - may include iteration and sub-procedures or subroutines
  - need to specify data structures that may be required
  - algorithm may be expressed in pseudo-code (e.g. $A \leftarrow A+1$)
  - algorithm code may be expressed in flow-chart
- Programming
  - convert computational (or flowchart) steps into executable statements and data structures in target language
- Program testing & debugging
- Program maintenance

Flow-Chart Symbols



Programs to do Simple Arithmetic

- Write a program starting at memory location $1500 to subtract the contents of memory location $1005 from the sum of memory locations $1000 and $1002 and store the difference at $1010.

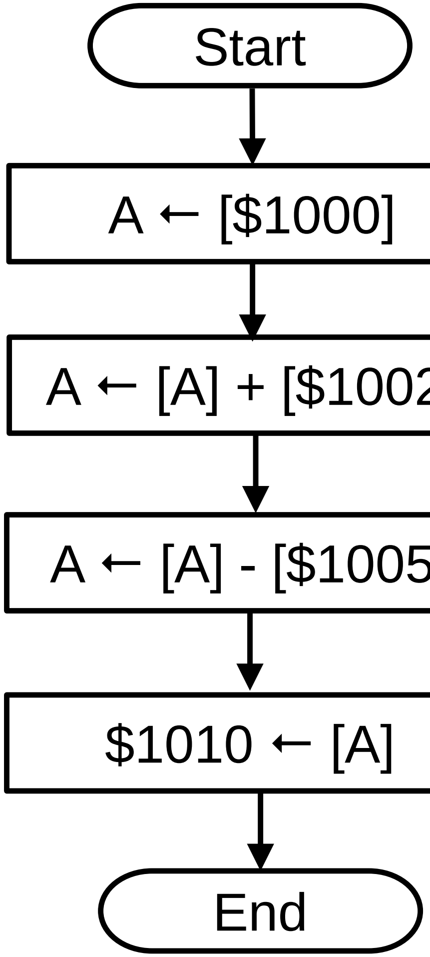
- Solution:

Step1: Load contents of memory loc. $1000 into A

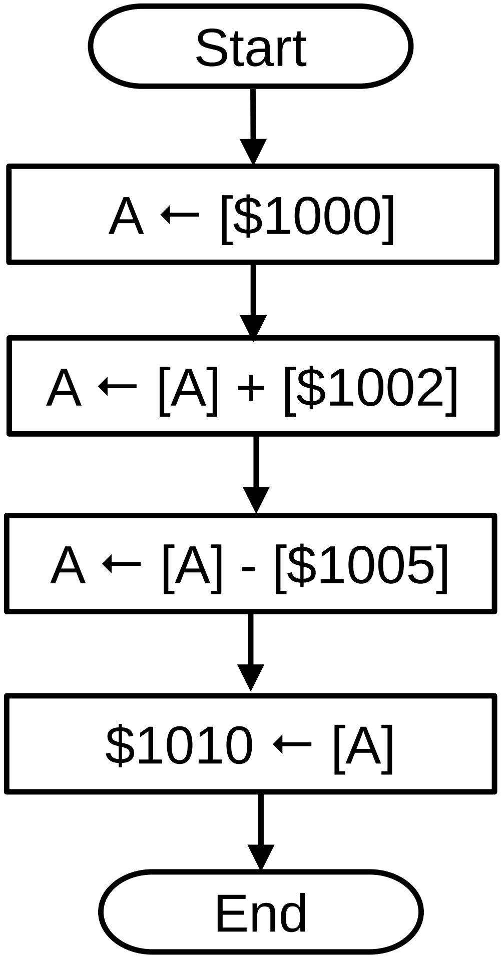
Step2: Add contents of memory loc. $1002 to A

Step3: Subtract contents of memory loc. $1005 from A

Step4: Store contents of A to memory loc. $1010



Algorithm to Assembly Code




```
ORG     $1500     ; starting address
ldaa    $1000     ; A ← [$1000]
adda    $1002     ; A ← [A] + [$1002]
suba    $1005     ; A ← [A] - [$1005]
staa    $1010     ; $1010 ← [A]
bgnd              ; break to debugger
END               ;end of program
```

More on Arithmetic (Add/Sub)

- We know how to add 8-bit quantities using A or B accumulator:

```
ldaa    $1000       ; add 8-bit data in $1000
adda    $1001       ; to 8-bit data in $1001
staa    $2000       ; and store 8-bit result in $2000
```

- and we can add 16-bit quantities using D accumulator:

```
ldd     $1000       ; add 16-bit data in $1000:$1001
addd    $1002       ; to 16-bit data in $1002:$1003
std     $2000       ; and store 16-bit result in $2000:$2001
```

- How can we add quantities of greater precision (e.g. 24-bit) ?

Carry/Borrow Flag

- Carry is LSB of CCR

S	X	H	I	N	Z	V	C

- Carry acts like the n^{th} result bit when doing n-bit add/sub
- Carry set to ‘1’ whenever addition generates carry-out
- Carry set to ‘1’ whenever subtraction requires borrow-out

```
ldaa   #$20
adda   #$30

   20
  +30
 0 50
```

```
ldaa   #$93
adda   #$8B

   93
  +8B
 1 1E
```

```
ldaa   #$96
suba   #$3A

   96
  –3A
 0 5C
```

```
ldaa   #$27
suba   #$54

   27
  –54
 1 D3
```

- also 16-bit:

```
ldd    #$A200
addd   #$7000

   A200
  +7000
 1 1200
```

Multi-precision Addition

- Carry bit allows us to do multi-precision arithmetic
  - i.e. arithmetic on numbers whose precision is greater than that of the ALU

- For example how do we add $59A183 to $5482DB ?

- Solution:

  Step1: Add $A183 to $82DB and remember carry

  Step2: Store 16-bit result as least significant two bytes of answer

  Step3: Add $59 to $54 with carry from step 1

  Step4: Store 8-bit result as MSbyte of answer

32-bit Arithmetic

- Write a program at memory location $4000 to add 4-byte numbers that are stored at $1000~$1003 and $1004~$1007 and store the sum at locations $1010~1013

Multiplication

Mnemonic	Function	Operation
emul	Unsigned 16 x 16 multiply	Y:D ← [D] x [Y]
emuls	Signed 16 x 16 multiply	Y:D ← [D] x [Y]
mul	Unsigned 8 x 8 multiply	D ← [A] x [B]

- Write an instruction sequence to multiply (unsigned) register X by register Y and store result in \$1000~\$1003

```
tfr     x,d       ;transfer X operand into D
emul              ;perform multiplication
sty     $1000     ;save upper 16-bits of product
std     $1002     ;save lower 16-bits of product
```

Multiplication

Mnemonic	Function	Operation
emul	Unsigned 16 x 16 multiply	Y:D ← [D] x [Y]
emuls	Signed 16 x 16 multiply	Y:D ← [D] x [Y]
mul	Unsigned 8 x 8 multiply	D ← [A] x [B]

- n1 and n2 are signed 16-bit integers and n3 is a 32-bit signed integer. Use assembler directives to reserve space for n1, n2 and n3 at memory locations $1000, $1002 and $1004 respectively.

  Write code starting at $4000 to perform: n3 = n1 * n2

Division

Mnemonic	Function	Operation
ediv	Unsigned 32 by 16 divide	$Y \leftarrow [Y]:[D] \div [X]$ $D \leftarrow remainder$
edivs	Signed 32 by 16 divide	$Y \leftarrow [Y]:[D] \div [X]$ $D \leftarrow remainder$
idiv	Unsigned 16 by 16 divide	$X \leftarrow [D] \div [X]$ $D \leftarrow remainder$
idivs	Signed 16 by 16 divide	$X \leftarrow [D] \div [X]$ $D \leftarrow remainder$

- Remember that if divisor > dividend, then quotient = 0

Integers Math and Precision

- When performing integer arithmetic (especially multiplication and division), important to keep track of potential size of results to avoid overflow and/or loss of precision
- Suppose we want to calculate: $$\frac{1200 \times 2500}{1150}$$
- Does the order of the operations matter?
- Correct answer is 2608.6956
  - (but we can only do integer arithmetic)
- If I do multiply first (emul followed by ediv), I get 2608
- If I do divide first (ediv followed by emul), I get 2500

Rounding

- If we take the quotient as being the answer to a divide operation (and ignore the remainder), the result is truncated to the closest integer that is less than the correct answer (2608 instead of 2608.6956)
- A better result would be to round to the nearest integer (2609). This can be achieved by adding half of the divisor to the dividend before executing the divide operation:

$$rounded\ quotient = \frac{dividend + (divisor/2)}{divisor}$$

- This effectively adds 0.5 to the answer before truncation.

Integer Precision: Example

- Multiply the unsigned 16-bit number in locations \$1000:\$1001 by 1.414 (approx. to $\sqrt{2}$), truncating result to nearest integer.

- Multiply the unsigned 16-bit number in locations \$1000:\$1001 by 1.414 (approx. to $\sqrt{2}$), rounding result to nearest integer.