EE 319K
Introduction to Embedded Systems

Lecture 2: Addressing modes, Memory Operations, Subroutines, I/O, Logical/Shift Operations, Introduction to C

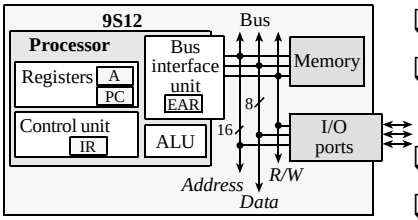
Announcements

- Lab
  - No lab this week
    - TAs will give intro and TExaS demo in the lab
  - First lab due next week
    - Digital lock in TExaS
    - Select lab partner (e.g. during TA demo)
    - Get started!
- Homework
  - First homework due next Monday
    - Hand assembling of a simple program into object code
- Equipment
  - Install TExaS on your computer/laptop

Agenda

- Recap
  - Data and number representations
  - 9S12 assembly programming
  - TExaS simulator
- Outline
  - 9S12 execution
    - Addressing modes, memory operations
  - Input/output
  - Logical/shift operations
  - Introduction to C
    - Structure of a C program
    - Boolean expressions and assignments

The 9S12



Major Components:

- Control Unit (CU)
- Instruction Register (IR)
- Arithmetic Logic unit (ALU)
- Program counter (PC)
- Bus interface unit (BIU)
- Effective Address Register (EAR)

Ramesh Yerraballi

3-4

Control Unit (CU): Controls the sequence of operations in the processor.

Instruction Register (IR): contains the op code for the current instruction

Arithmetic Logic unit (ALU): performs operations such as addition, subtraction, multiplication and division.

Program counter (PC): points to the memory containing the instruction to execute next.

Bus Interface Unit (BIU): reads data from the bus during a read cycle, and writes data onto the bus during a write cycle.

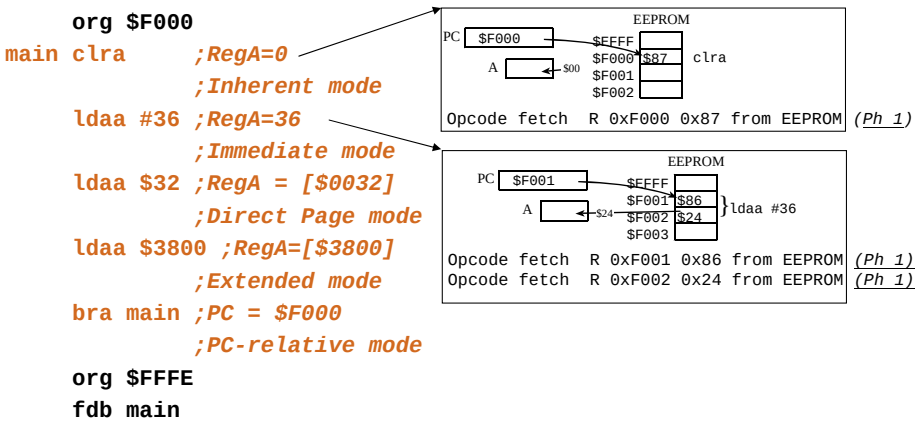
Effective Address Register (EAR): contains the data address for the current instruction.

Instruction Execution: Phases

Phase	*Function*	*R/W*	*Address*	*Comment*
1	**Op code fetch** **Operand fetch**	**read** **read**	**PC++** **PC++**	**Put op code into IR** **Immediate or calculate EA**
2	Decode Instruction	None		Figure out what to do
3	Evaluate Address	none		Determine EAR
4	**Data read**	**read**	**SP, EAR**	**Data passes thru ALU**
5	Free cycle	read	PC/SP/$ FFFF	ALU operations, set CCR
6	**Data Store**	**write**	**SP, EAR**	**Results stored in memory**

Note: Bold text shows those phases that generate bus cycles

Simple Addressing Modes



```
      org $F000
main  clra      ;RegA=0
                ;Inherent mode
      ldaa #36  ;RegA=36
                ;Immediate mode
      ldaa $32  ;RegA = [$0032]
                ;Direct Page mode
      ldaa $3800 ;RegA=[$3800]
                ;Extended mode
      bra main  ;PC = $F000
                ;PC-relative mode
      org $FFFE
      fdb main
```


Inherent addressing mode has no operand field

sometimes there is no data

ex:- stop

sometimes the data for the instruction is implied.

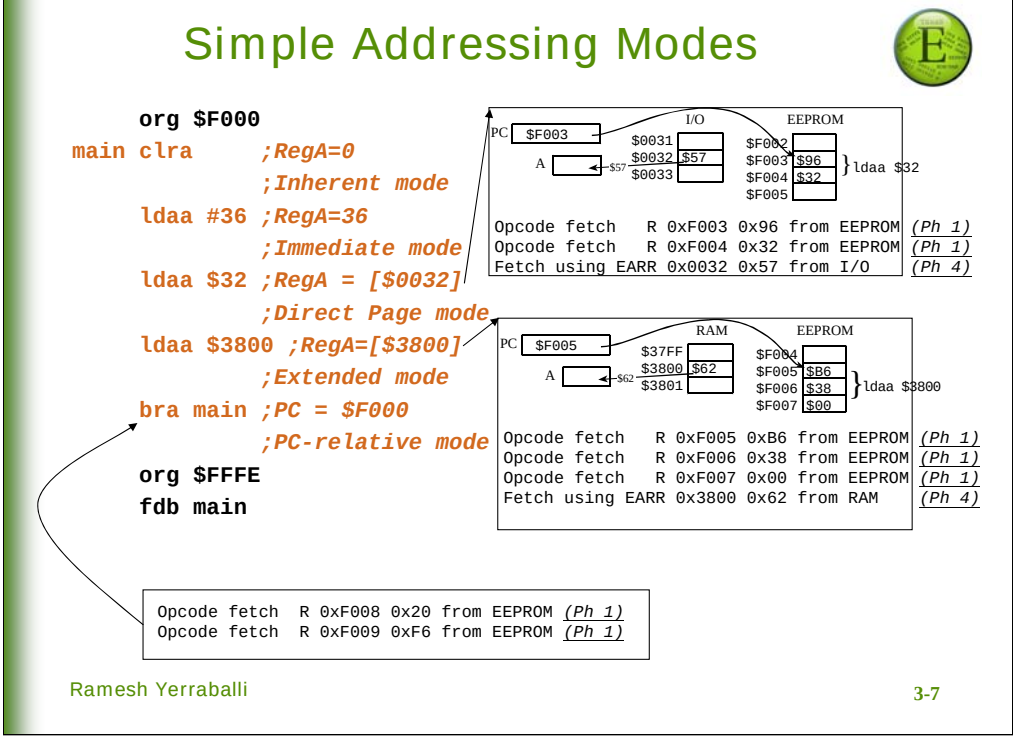
ex:- clra

sometimes the data must be fetched, but the address is implied

ex:- pula

Immediate addressing mode uses a fixed data constant

Simple Addressing Modes



Direct Page addressing mode

- Uses an 8-bit address
- Access from addresses 0 to $00FF
- Called zero-page.
- The < operator forces Direct Page addressing
- On the 6812 they reference the I/O ports : On the 9S12DP512, *Port K* is at address $0032

Extended Addressing mode

- Uses a 16-bit address
- Size of data depends on the op code (which register is uses)
- Access all memory and I/O devices
- Outside Motorola family this addressing mode is called direct
- The > operator forces extended addressing

PC-Relative Addressing mode

- Used for the branch instructions
- Stored in the machine code is not the absolute address but the 8-bit signed offset relative distance from the current PC value
- The PC already points to the next instruction
- The assembler calculates it for us

ldaa

LDAA Load Accumulator A LDAA

Operation: (M) ⇒ A

Description: Loads the content of memory location M into accumulator A. The condition codes are set according to the data.

Condition Codes and Boolean Formulas:

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	0	–

N: Set if MSB of result is set; cleared otherwise.
Z: Set if result is $00; cleared otherwise.
V: 0; Cleared.

Addressing Modes, Machine Code, and Execution Times:

Source Form	Address Mode	Object Code	Cycles	Access Detail
LDAA *#opr8i*	IMM	86 ii	1	P
LDAA *opr8a*	DIR	96 dd	3	rfP
LDAA *opr16a*	EXT	B6 hh ll	3	rOP
LDAA *oprx0_xysp*	IDX	A6 xb	3	rfP
LDAA *oprx9,xysp*	IDX1	A6 xb ff	3	rPO
LDAA *oprx16,xysp*	IDX2	A6 xb ee ff	4	frPP
LDAA [D,*xysp*]	[D,IDX]	A6 xb	6	fIfrfP
LDAA [*oprx16,xysp*]	[IDX2]	A6 xb ee ff	6	fIPrfP

Ramesh Yerraballi

3-8

LDAA	#opr8i	IMM	86 ii
LDAA	opr8a	DIR	96 dd
LDAA	opr16a	EXT	B6 hh ll
LDAA	oprx0_xysp	IDX	A6 xb
LDAA	oprx9,xysp	IDX1	A6 xb ff
LDAA	oprx16,xysp	IDX2	A6 xb ee ff
LDAA	[D,xysp]	[D,IDX]	A6 xb
LDAA	[oprx16,xysp]	[IDX2]	A6 xb ee ff

(M)⇒ A Load Accumulator A

Simple Addressing Modes

- Clarifications:
  - Immediate mode can use more than 8-bit values:

```
ldd #W ;RegD=W load a 16-bit constant into RegD
lds #W  ;SP=W load a 16-bit constant into SP
```

  - Branch uses a 8-bit offset however there is a long Branch instruction that can increase this to 16-bits
    - bra, bmi, bne, bpl use 8-bit offset:

```
bra rel8            20 rr
```

    - lbra, lbmi, lbne, lbpl, and 16 other long branch instructions use 16-bit offset:

```
lbra rel16          18 20 qq rr
```

    - Jmp uses 16-bit destination address in extended addr mode:

```
jmp opr16           06 hh ll
```

Ramesh Yerraballi 3-9

Assume `bra there` is at $5000

Assume `there` is at $5036

What is machine code?

Assume `bra there` is at $6000

Assume `there` is at $6096

What is machine code?

Memory to memory move

- Note that the "addressing mode" applies to the operands. If an instruction has two operands each has its own addressing mode:
  - **movb #w,addr** ; [addr]=w EXT-IMM
    Move an 8-bit constant into memory
  - **movb addr1,addr2** ; [addr2]=[addr1] EXT-EXT
    Move an 8-bit value memory to memory
  - **movw #W,addr** ; {addr}=W EXT-IMM
    Move a 16-bit constant into memory
  - **movw addr1,addr2** ; {addr2}={addr1} EXT-EXT
    Move a 16-bit value memory to memory

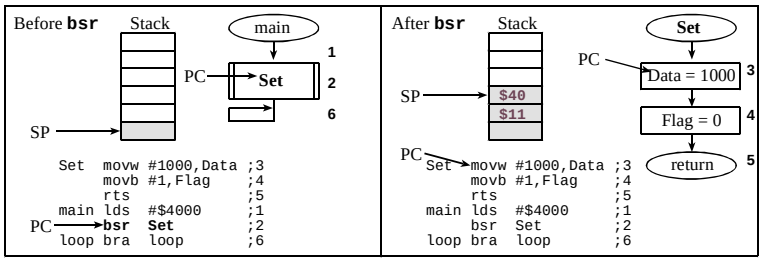
Subroutines

```
$0800                       org  $0800
$0800                 Flag  rmb  1
$0801                 Data  rmb  2
$4000                       org  $4000
                      ;*****Set**************
                      ; Set Data=1000, and Flag=1
                      ; Input: None
                      ; Output: None
$4000 180303E80801    Set   movw #1000,Data  ;3
$4006 180B010800            movb #1,Flag     ;4
$400B 3D                    rts              ;5
$400C CF4000          main  lds  #$4000      ;1
$400F 07EF                  bsr  Set         ;2
$4011 20FE            loop  bra  loop        ;6
$FFFE                       org $fffe
$FFFE 400C                  fdb main
```

Ramesh Yerraballi 3-11

Since the ***bsr*** instruction uses relative addressing, it can only be used to call a subroutine near the current instruction. Since the ***jsr*** instruction allows extended addressing, it can be used to call a subroutine anywhere in memory.

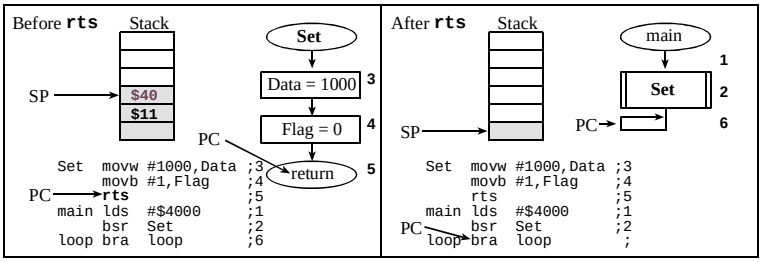
Stack Use in Subroutines: **bsr**



bsr Execution:

Opcode fetch	R 0x400F 0x07 from ROM	*Phase 1*
Operand fetch	R 0x4010 0xEF from ROM	*Phase 1*
Stack store lsbW	0x3FFF 0x11 to RAM	*Phase 6*
Stack store msbW	0x3FFE 0x40 to RAM	*Phase 6*

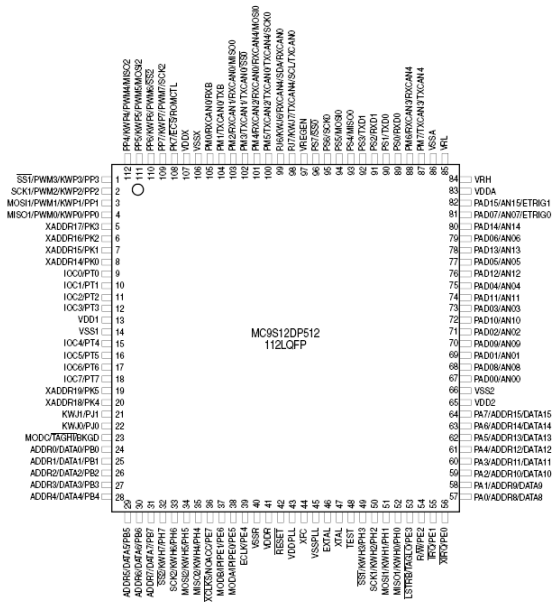
Stack Use in Subroutines: **rts**



rts Execution:

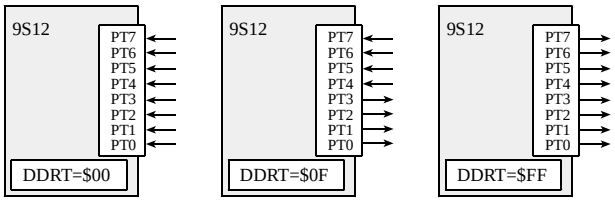
```
Opcode fetch   R 0x4009 0x3D from ROM     Phase 1
Stack read msb R 0x3FFE 0x40 from RAM     Phase 4
Stack read lsb R 0x3FFF 0x11 from RAM     Phase 4
```

Input/Output: 9S12DP512



I/O Ports and Direction Registers

Address	Bit7	6	5	4	3	2	1	Bit0	Name
$0240	PT7	PT6	PT5	PT4	PT3	PT2	PT1	PT0	PTT
$0242	DDRT7	DDRT6	DDRT5	DDRT4	DDRT3	DDRT2	DDRT1	DDRT0	DDRT



- *The input/output direction of a bidirectional port is specified by its direction register.*
- DDRH, DDRP, DDRJ, DDRT, specify if corresponding pin is input or output:
  - 0 means input
  - 1 means output

Ramesh Yerraballi

3-15

To make all pins input, we clear the direction register:

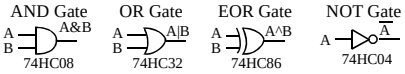
```
movb #0,DDRT ;make all PTT pins an input
```

Make PTT pins 7-4 input, and pins 3-0 output, then make PT3-PT0 output high:

```
 ldaa #$0F
staa DDRT   ;PT7-PT4 inputs, PT3-PT0 outputs
ldaa #$0F
staa PTT   ;make PT3-0 high
```

Logical Operations

A	B	A&B	A|B	A^B
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0




```
anda #w   ;RegA=RegA&w
anda u    ;RegA=RegA&[u]
anda U    ;RegA=RegA&[U]
oraa #w   ;RegA=RegA|w
oraa u    ;RegA=RegA|[u]
oraa U    ;RegA=RegA|[U]
eora #w   ;RegA=RegA^w
eora u    ;RegA=RegA^[u]
eora U    ;RegA=RegA^[U]
coma      ;RegA=~RegA
```


Logical ops bita, bitb, oraa, orab perform the logical operation and set the condition code bits accordingly but do not modify registers A or B

To set

The **or** operation to set bits 1 and 0 of the register DDRT.
The other six bits of DDRT remain constant.

Friendly software modifies just the bits that need to be.

```
DDRT |= 0x03; // PT1,PT0 outputs
```

Assembly:

```
ldaa DDRT   ;read previous value
oraa #$03   ;set bits 0 and 1
staa DDRT   ;update
```

c_7	c_6	c_5	c_4	c_3	c_2	c_1	c_0	
0	0	0	0	0	0	1	1	**$03** constant
c_7	c_6	c_5	c_4	c_3	c_2	1	1	result of the **oraa**

value of **DDRT**

To toggle

The **exclusive or** operation can also be used to toggle bits.

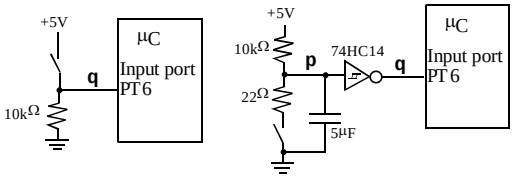
```
PTT ^= 0x80;  /* toggle PT7 */
```

Assembly:

```
ldaa PTT   ;read output Port T
eora #$80  ;toggle bit 7
staa PTT   ;update
```

b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0	value of **PTT**
1 0 0 0 0 0 0 0	**$80** constant
$\sim b_7$ b_6 b_5 b_4 b_3 b_2 b_1 b_0	result of the **eora**

Switch Interfacing



The **and** operation to extract, or *mask*, individual bits:

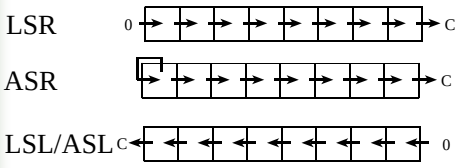
```
Pressed = PTT&0x40;//true if the switch pressed
```

Assembly:

```
ldaa PTT      ;read input Port T
anda #$40     ;clear all bits except bit 6
staa Pressed ;true iff the switch is pressed
```

a_7	a_6	a_5	a_4	a_3	a_2	a_1	a_0	
0	1	0	0	0	0	0	0	value of **PTT**
0	1	0	0	0	0	0	0	**$40** constant
0	a_6	0	0	0	0	0	0	result of the **anda**

Shift Operation




```
asla ;RegA=RegA*2
lsla ;RegA=RegA*2
asra ;RegA=RegA/2
lsra ;RegA=RegA/2
```

*Use the **asla** instruction when manipulating signed numbers, and use the **lsla** instruction when shifting unsigned numbers*

Shift Example

High and **Low** are unsigned 4-bit components, which will be combined into a single unsigned 8-bit **Result**.

```
Result = (High<<4)|Low;
```

Assembly:

```
ldaa High   ;read value of High
lsla        ;shift into position
lsla
lsla
lsla
oraa Low    ;combine the two parts
staa Result ;save answer
```

0 0 0 0 h_3 h_2 h_1 h_0	value of **High**
0 0 0 h_3 h_2 h_1 h_0 0	after first **lsla**
0 0 h_3 h_2 h_1 h_0 0 0	after second **lsla**
0 h_3 h_2 h_1 h_0 0 0 0	after third **lsla**
h_3 h_2 h_1 h_0 0 0 0 0	after last **lsla**
0 0 0 0 l_3 l_2 l_1 l_0	value of **Low**
h_3 h_2 h_1 h_0 l_3 l_2 l_1 l_0	result of the **oraa** instruction

Introduction to C

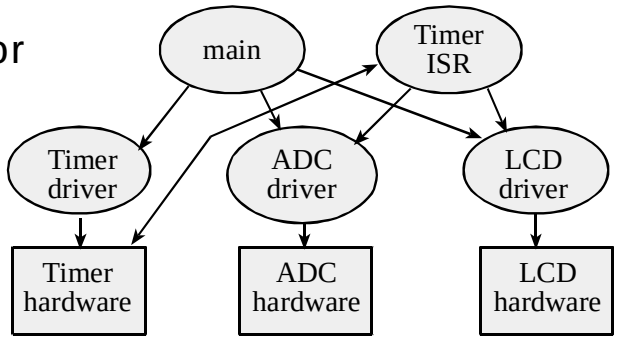
- C is a high-level language
  - Abstracts hardware
  - Expressive
  - Readable
  - Analyzable
- C is a *procedural language*
  - The programmer explicitly specifies steps
  - Program composed of procedures
    - Functions/subroutines
- C is compiled (not interpreted)
  - Code is analyzed as a whole (not line by line)

Why C?

- C is popular
- C influenced many languages
- C is considered close-to-machine
  - Language of choice when careful coordination and control is required
  - Straightforward behavior (typically)
- Typically used to program low-level software (with some assembly)
  - Drivers, runtime systems, operating systems, schedulers, …

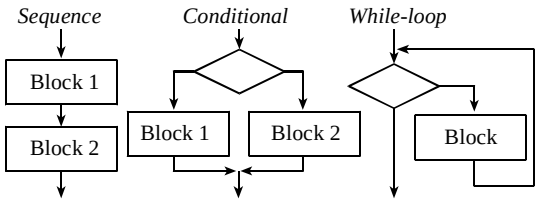
Introduction to C

- Program structure
  - Subroutines and functions
- Variables and types
- Statements
- Preprocessor
- DEMO



A C Program (demo)

- Preprocessor directives
- Variables
- Functions
- Statements
- Expressions
- Names
- Operators
- Comments
- Syntax



Important Notes

- C comes with a lot of “built-in” functions
  - printf() is one good example
  - Definition included in *header files*
  - #include<header_file.h>
- C has one special function called *main*()
  - This is where execution starts (reset vector)
- C development process
  - Compiler translates C code into assembly code
  - Assembler (e.g. built into TExaS) translates assembly code into object code
  - Object code runs on machine