An Introduction to CHASM

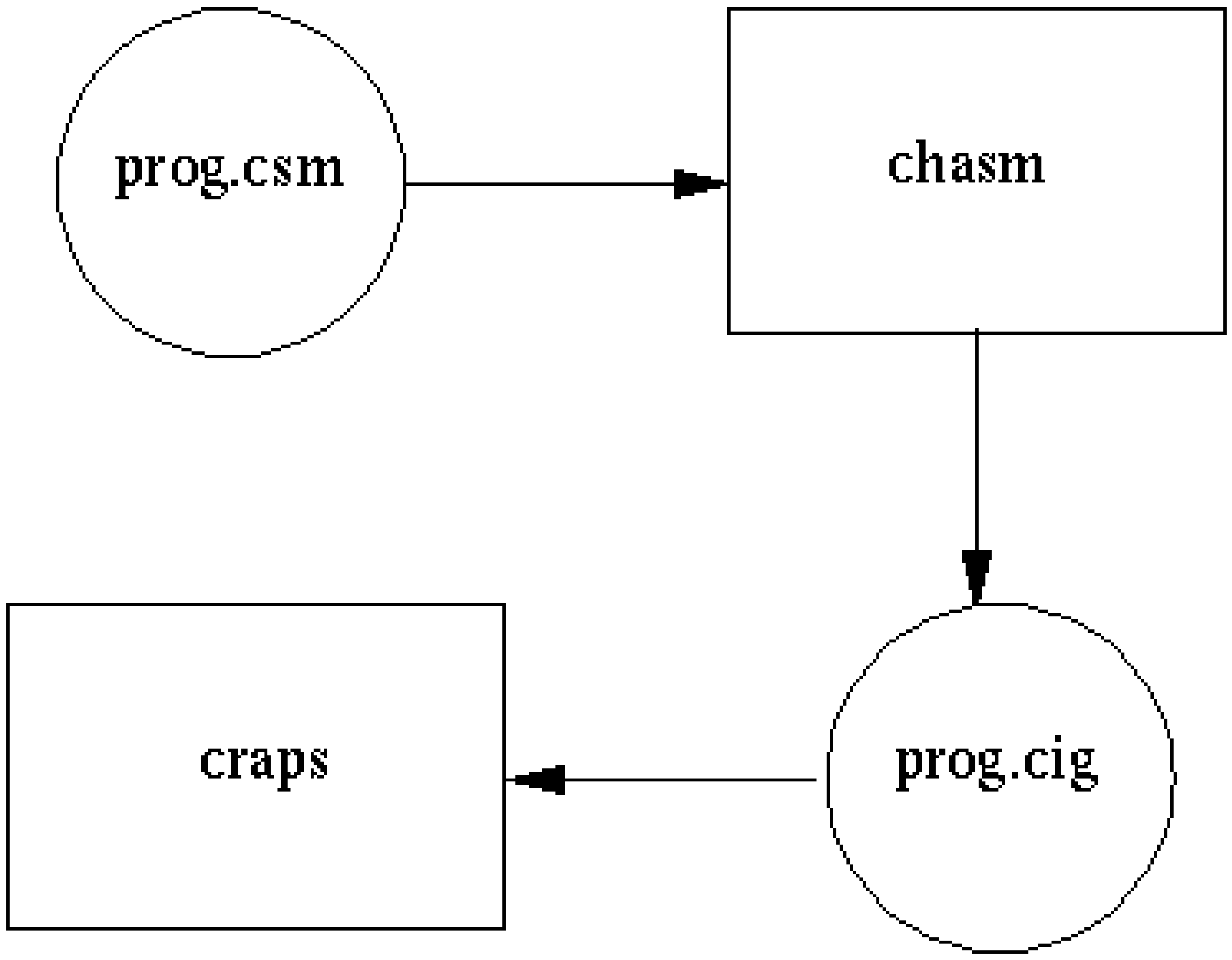
Assembly Programming for the CRAPS emulator

COMP183

Deryk Barker

January 2013 Edition

Assembling and running a CHASM program



The “Four Step” Programme

1. Build source code: assembly source files are regular ASCII text files, just like the source files for any other programming language.
2. Assemble program: the programs which translate assembly source into object files are called assemblers, as opposed to those for high-level languages, which are called compilers. Both take a source file and translate it into object code.

The “Four Step” Programme

3. Link program: the process of building an executable binary file from the object file. May involve linking pre-built libraries - either system- or user-supplied. This step is not required by the CRAPS system. CHASM is what is known as an *absolute assembler* and generates an executable file directly.

CHASM syntax

- Mnemonic
  - A CHASM-defined name for a machine instruction
  - A CHASM directive (or pseudo-op) which may or may not cause machine instructions to be generated
  - May ***not*** *begin in column 1 even when there is no label*
- Operands
  - A comma-separated list of the objects on which the instruction operates
- Comment
  - Any text following white space after the operands
  - CHASM also considers a semicolon (;) character to indicate that the rest of the line is a comment, i.e. will be **ignored** by the assembler

The “Four Step” Programme

4. Run program: the output image file is loaded and executed by the CRAPS simulator/debugger program.

What does CHASM look like?

- An assembly language's form is determined by two things:
  - Processor architecture
    - Dictates which instructions are legal and which are not
    - Strongly affects language structure
  - The assembler program being used
    - An assembler is a compiler (or translator) for an assembly-level language
    - Compilers translate high-level (sometimes called problem-oriented) languages, where one language statements usually translates to several machine instructions
    - Assemblers translate low-level or assembly languages where one language statement usually translates to one machine instruction
    - Both compilers and assemblers produce an object file (or object program) which must then (on most real systems) be linked to produce an executable file (or executable program, or binary image)

CHASM syntax

- Standard CHASM syntax consists of four fields separated by white-space.

  Label Mnemonic Operands Comment

- Label
  - Is optional
  - Identifies a named location (address) in the code
  - Used as target of goto type instructions, and (occasionally) of call instructions
  - Also used to name reserved storage (for use as variables)
  - Identifier must be in column 1 order to be recognized as a label

CHASM syntax

- Standard CHASM syntax consists of four fields separated by white-space.

Label Mnemonic Operands Comment

- Label
  - Is optional
  - Identifies a named location (address) in the code
  - Used as target of goto type instructions, and (occasionally) of call instructions
  - Also used to name reserved storage (for use as variables)
  - Identifier must be in column 1 order to be recognized as a label

CHASM syntax

- **CHASM is (semi-)free-format**
  - There are no fixed column positions for the various fields, except for labels, which must begin in column 1
  - Fields may be separated by one or more spaces or tabs
  - Labels are not required
  - Labels may appear on a line by themselves, in this case it is the memory address of the next statement which will be referred to by the label
  - Blank lines are ignored
  - Statements (apart from a label and its associated statement) cannot be split across lines
  - Only one statement is allowed per line
  - The built-in names (registers, mnemonics) are case-insensitive; user-defined names are case sensitive.

CHASM instructions

- CHASM instructions have 0, 1, 2 or 3 operands, depending on the individual instruction
- **Zero-operand instructions**

```
nop              ; this instruction is the no-op
halt             ; stop the CPU (deprecated - use the
                   SYStem call)
```

- ***Single-operand instructions***

```
br    Label        ; jump to Label
call  routine      ; jump to routine but remember
                     from whence we came
```

CHASM instructions

- **Two-operand instructions**

  mov r1, r2   ; copy contents r1 to r2

  means move (i.e. copy) the value of register r1 to register r2. The contents of r1 are unchanged. The contents of r2 are overwritten

- **Three-operand instructions**

  add r1, r2, r3   ; add r1, r2 result -> r3

CHASM instructions

- Assembly programming largely consists of moving data between the three functional components of the system: the CPU, memory (RAM) and the I/O subsystem (i.e. disks, video, CD-ROM, etc) and manipulating it within the CPU in special fast storage areas called *registers* also known as *scratchpad* memory.

CHASM instructions

- In Chasm there are essentially 4 instruction types:
  - Data movement
  - Data manipulation
  - Flow of control
  - System calls

CHASM instructions

- **Data movement**
  - Instructions which copy (move is a bad choice of word, but we're stuck with it for historical reasons) data from one component to another or, in the case of the CPU, from one register to another.
- **Data manipulation**
  - **Arithmetic instructions**
    - add, sub, mul, div
  - **Logic instructions**
    - and, or, xor, shift, rotate

CHASM instructions

- **Data movement**
  - Instructions which copy (move is a bad choice of word, but we're stuck with it for historical reasons) data from one component to another or, in the case of the CPU, from one register to another.
- **Data manipulation**
  - **Arithmetic instructions**
    - add, sub, mul, div
  - **Logic instructions**
    - and, or, xor, shift, rotate

CHASM instructions

- **Data movement**
  - Instructions which copy (move is a bad choice of word, but we're stuck with it for historical reasons) data from one component to another or, in the case of the CPU, from one register to another.
- **Data manipulation**
  - **Arithmetic instructions**
    - add, sub, mul, div
  - **Logic instructions**
    - and, or, xor, shift, rotate

CHASM instructions

- **Flow of Control**
  - Instructions which alter the normal sequential flow of control through the code
    - br, be, call, ret
- **System Calls**
  - In CRAPS these are used these for I/O and program termination

A Skeleton CHASM Program

```
        PROGRAM     Test
        STACK       60
        DATA
        ;data goes here
        CODE
Test
        ;code goes here
        sys         0xffff ; terminate the program
        END
```

But What Does It All Mean?

- PROGRAM
  an assembler ***directive*** which informs CHASM of the label on the *entry-point* to the program. This is the location at which the program will start executing; it doesn't have to be the first instruction in memory.

But What Does It All Mean?

- STACK 60
  reserves 60_{10} bytes of stack space. The SP register is automatically initialized at load time
  Strictly speaking, the STACK directive is not necessary, but if the program does not include it and attempts to use the stack it will probably crash
- DATA
  marks the beginning of the data region where all program-defined variables reside

But What Does It All Mean?

- CODE
  marks the beginning of the code region where all executable instructions reside
- Test
  a label marking the beginning of the executable code

But What Does It All Mean?

- sys
  - the *sys* instruction is effectively a call to the OS (or BIOS) to ask for privileged functionality to be performed. In this case we are making system call number 65535 ($ffff_{16}$) which requests that the program be terminated normally.
  - This is the approved method (rather than the HALT instruction) of terminating a program.
- END
  - a pseudo-op which signifies the end of the end of the code. Note that this does **not** generate a HALT or a SYS instruction, but it does cause the assembler to **stop reading the source file**

A Very Simple Program

```
                program myprog
;
EXIT            EQU       0xffff
CONSOLE         EQU       1
PUTSTR          EQU       3
CR              EQU       13
LF              EQU       10
                STACK      10
                data
v1              db        'Hello'
v2              db        'World', CR, LF
;
                code
myprog
                mov       PUTSTR, r1
                lea       v1, r2
                sys       CONSOLE
                mov       PUTSTR, r1
                lea       v2, r2
                sys       CONSOLE
                sys       EXIT
                END
```

A Very Simple Program

- This program should simply print out the message 'Hello World!' and then terminate
- As Manuel would say: **Que?**
- A few points:
  - EQU makes a shorthand as you EQUate a name to a value:
    sys EXIT tends to to be more meaningful than **sys 0xffff**
  - db (Define Bytes) reserves and (optionally) initializes storage – in this case to our two messages; the trailing CR LF are to move the cursor to position 1 (CR) and the next line (LF) on the screen after the second message
  - Printing the message takes three instructions:
    - a mov to put the CRAB function code into R1
    - an lea to Load the **Effective Address** of the first byte of the message into r2 (this is specified in the CRAB documentation)
    - a sys CONSOLE instruction to send the message

A Very Simple Program

OK – let's do it

```
$ chasm test/notes1.csm
CHASM version 2.11a September 20, 2003
File: test/notes1.csm, Entrypoint: c, Errors: 0, Warnings: 0
$ craps test/notes1
HelloWorld
!World
!$
```

Well, close but no cigar – what's going wrong?

A Very Simple Program

- What was the output **supposed** to look like?
- Now, how did what we observe differ?
  1. The **World** message came out twice
  2. Each time, after **World** there was a spurious **! on the next line**
- We use this information to debug the program, but we can get help
- Firstly, we'll get CHASM to print a **listing** file

```
$ chasm -l notes1
```

The Listing file

```
File notes1.csm Program myprog Assembled Sat Jan  3 19:30:45 2004

   0001                       program myprog
   0002               ;
   0003          ffff EXIT    EQU             0xffff
   0004          0001 CONSOLE EQU             1
   0005          0003 PUTSTR  EQU             3
   0006          000d CR      EQU     13
   0007          000a LF      EQU     10
   0008                       STACK   10
   0009                       data
   0010 0000 48656c6c v1      db      'Hello'
        0004 6f
   0011 0005 576f726c v2      db      'World', CR, LF
        0009 640d0a
   0012               ;
   0013                       code
   0014 000c           myprog
   0015 000c 04210003         mov     PUTSTR, r1
   0016 0010 08210000         lea     v1, r2
   0017 0014 82200001         sys     CONSOLE
   0018 0018 04210003         mov     PUTSTR, r1
   0019 001c 08210005         lea     v2, r2
   0020 0020 82200001         sys     CONSOLE
   0021 0024 8220ffff         sys     EXIT
   0022                       END

File: notes1.csm, Entrypoint: c, Errors: 0, Warnings: 0

Name                Location/Value Defined Referenced
====                ========/===== ======= ==========
CONSOLE                      0001    4      17   20
CR                          000d    6      11
EXIT                        ffff    3      21
LF                           000a    7      11
PUTSTR                       0003    5      15   18
myprog                 000c         14       1
v1                     0000         10      16
v2                     0005         11      19
```

What The Listing Tells Us

- We can see exactly what is in memory and at what location
- Used in conjunction with the debugger this can help us to find the problem

```
$ craps -l notes1
CRAPS debugger version 2.11b September 18, 2003
CRAPS virtual machine version 2.0 February 27, 2003
loaded notes1 entrypoint at 000c
craps> d 0,20:ax
0000: HelloWor48 65 6c 6c 6f 57 6f 72
0008: ld...!..6c 64 0d 0a 04 21 00 03
0010: .!..    08 21 00 00
```

The CRAPS CPU – Register Set

- **General Purpose Registers**
  - The CRAPS CPU contains 16 16-bit *general purpose registers*, known as R0 through R15. With two exceptions each register may be used for any purpose.
  - The exceptions are:
    - R0 - this register is "wired" to zero. Using it as a source always gives the value zero; using it as a destination is effectively discarding the result of an operation, because no non-zero value will be stored in R0.
    - R15 - this register is used as the *stack pointer* and may also be referred to as SP. Only programmers *who know what they are doing* should use this register.

The CRAPS CPU – Register Set

- **Miscellaneous registers**
  - **PC** (*program counter* register) - holds the address of the next instruction to be fetched.
  - **Flags** - collection of single bits each with a distinct meaning. These are mainly set and cleared as by-products of execution of other instructions

The CRAPS CPU - Flags

- **C**(arry) - set **on** if the result of an instruction will not fit in to the 16-bit destination register. Unsigned values are limited to the range 0 to 65535 (2^{16}-1)

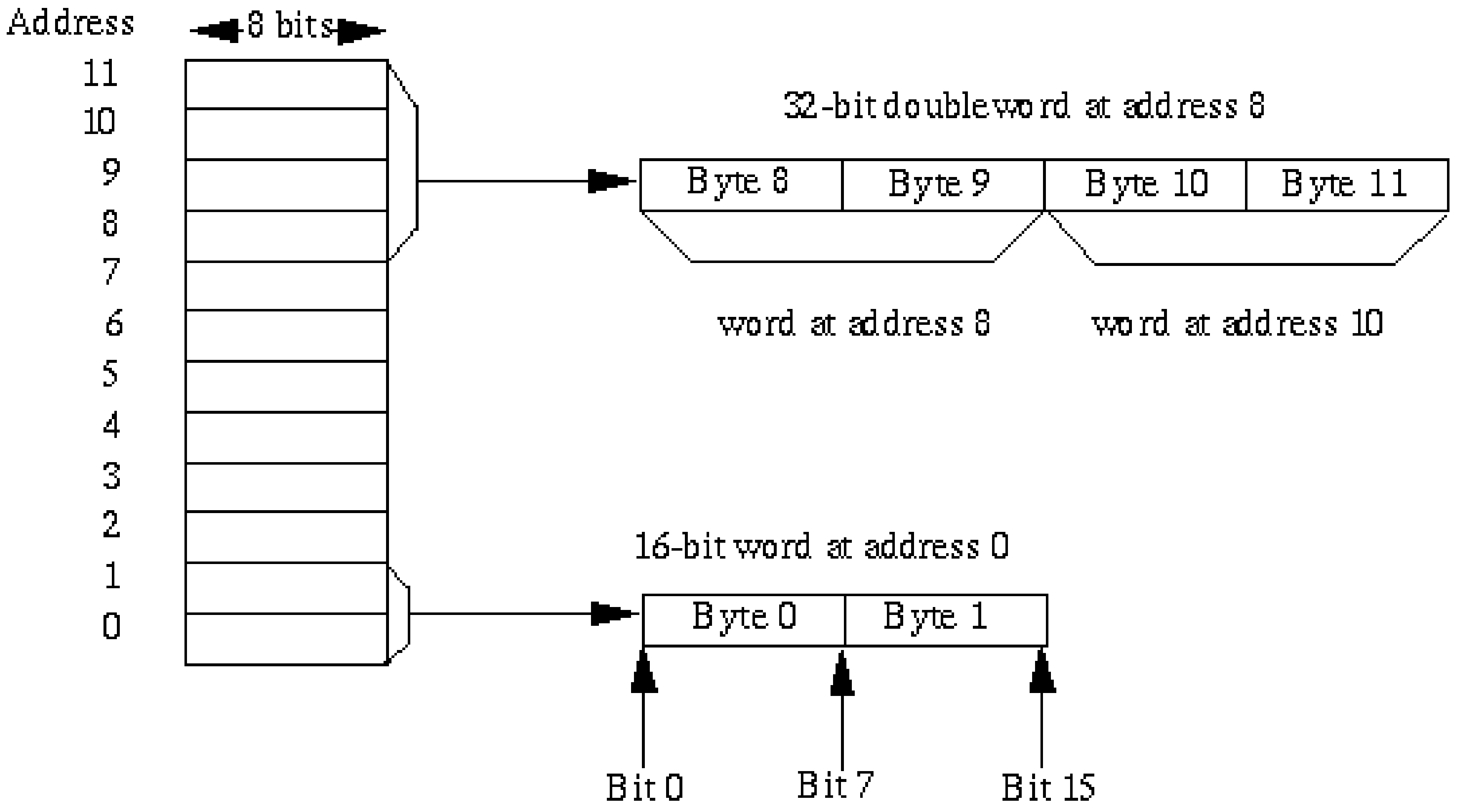
- **O**(verflow) - set **on** if the result of an instruction *considered as a signed number* will not fit into a 16-bit register. Signed values are limited to the range -32768 (-2^{15}) to 32767 (2^{15}-1)

- **S**(ign) - set equal to bit 0 (the leftmost bit) of the result of an operation. If set, the result value, *considered as a signed number*, will be negative.

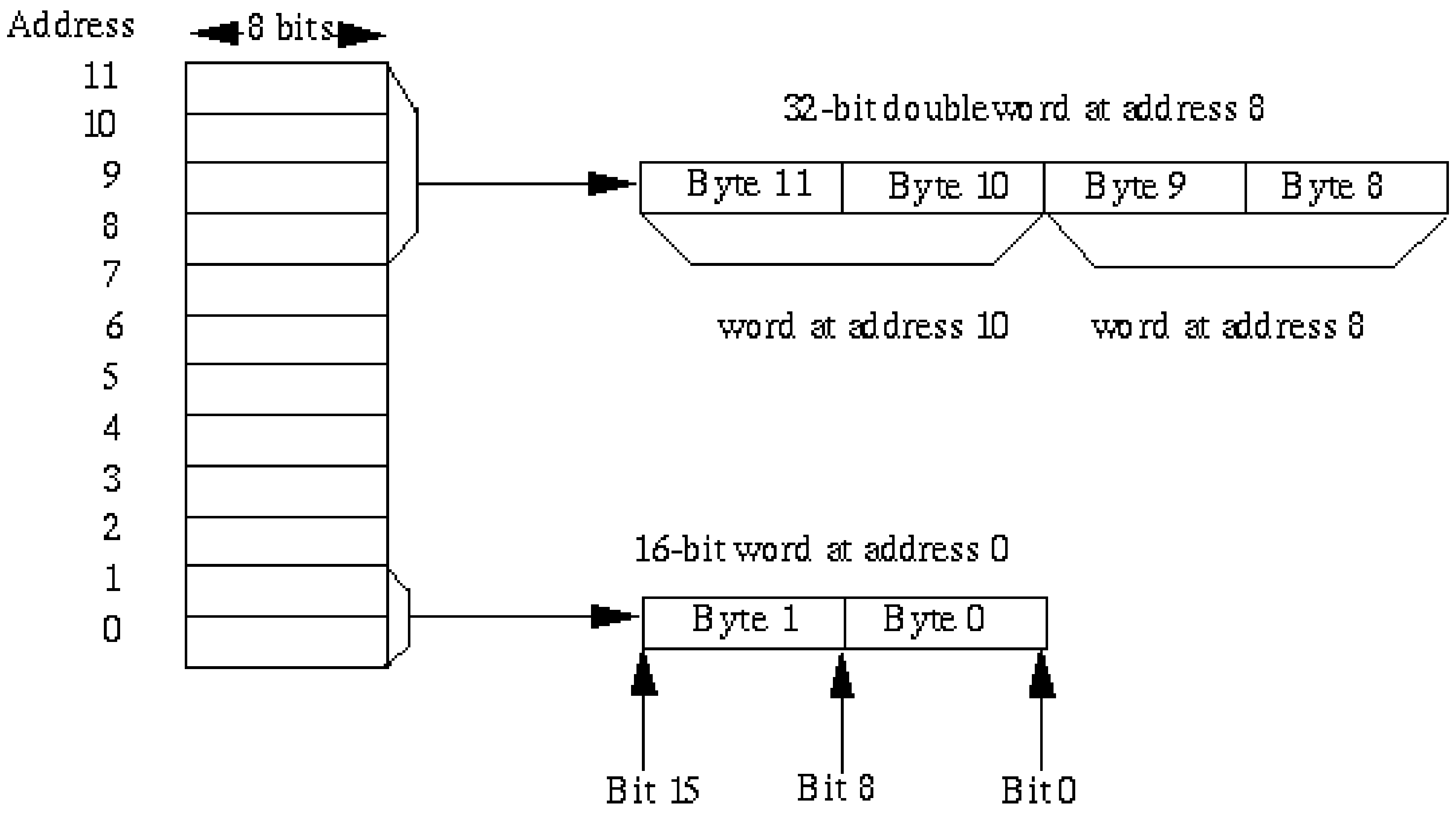
- **Z**(ero) - set on if the result of an instruction is all zero bits.

Note that not all instructions set all flags.
Consult individual instruction documentation for details.

The CRAPS view of memory



The Intel x86 view of memory



The CRAPS chips use is what is known as a ***Big-endian***, as opposed to ***Little-endian*** (e.g. Intel x86 chips) view of memory

CRAPS Instructions

- **Data Movement**
  - **Move Data**

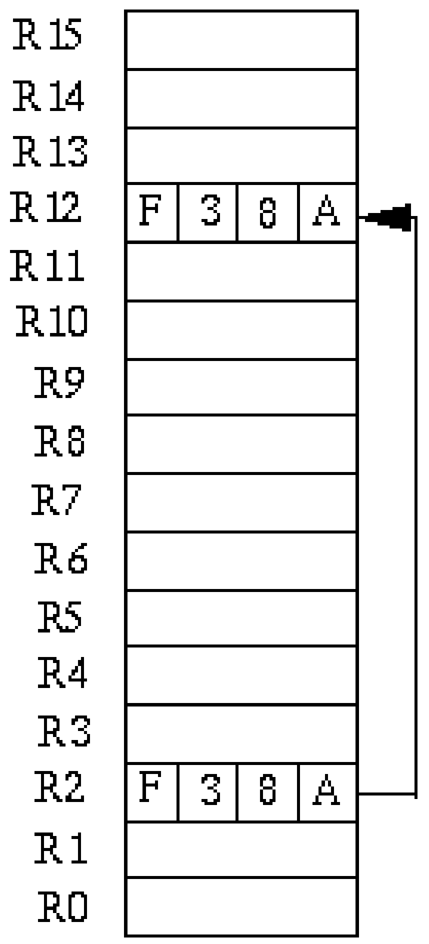
Name:	*Move data*
Mnemonic:	*MOV*
Format:	*mov src, dest*
Function:	*dest := src*
Flags:	*OC set off, SZ set from value moved*

The mov instruction is used to copy data between the registers. It is not used to access memory, for this purpose the various *load* and *store* instructions are used.

MOV Example

```
mov   r2, r12
```

Note that the contents of r2 remain unchanged



CRAPS Load Instructions

- **LOAD instructions**
  - There is a total of 5 instructions for loading data from memory into a register.

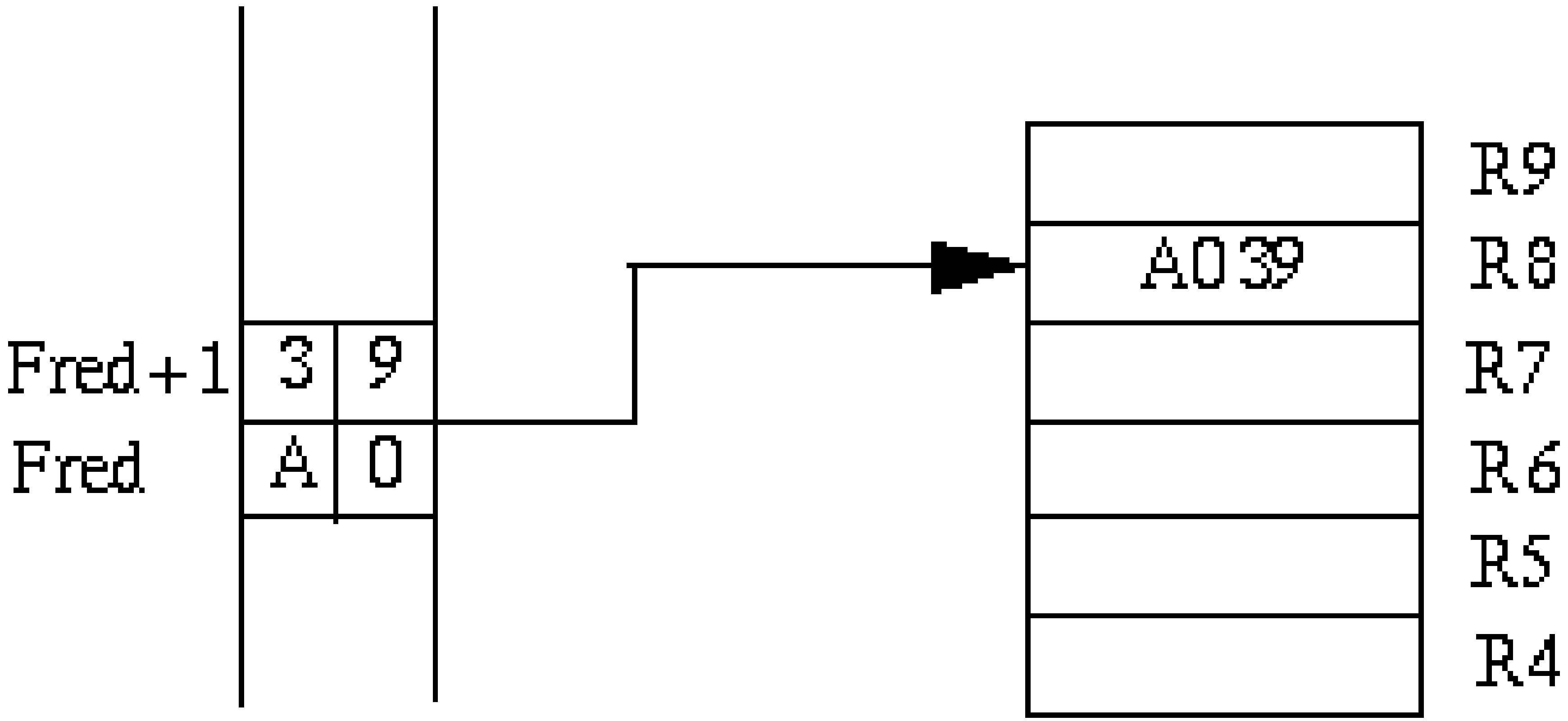
  - **LODW**

Name:	*Load Word*	
Mnemonic:	*LODW*	
Format:	*lodw address, [src], dest*	
Function:	*dest (bits 0-7)*	*:= M[address+src]*
	dest (bits 8-15)	*:= M[address+src+1]*
Flags:	*None*	

The destination register is loaded with the contents of the 16-bit word whose location (the ***effective* address**) is found by adding the specified *address* and the specified source register. This register is optional; if omitted, then the contents of the word designated by **address** will be loaded to the destination register.

LODW Example

lodw Fred,,r8



CRAPS Load Instructions

- **LODBH**

Name:	*Load Byte High*
Mnemonic:	*LODBH*
Format:	*lodbh address, [src], dest*
Function:	*dest (0-7) := M[address+src],*
	dest (8-15) untouched

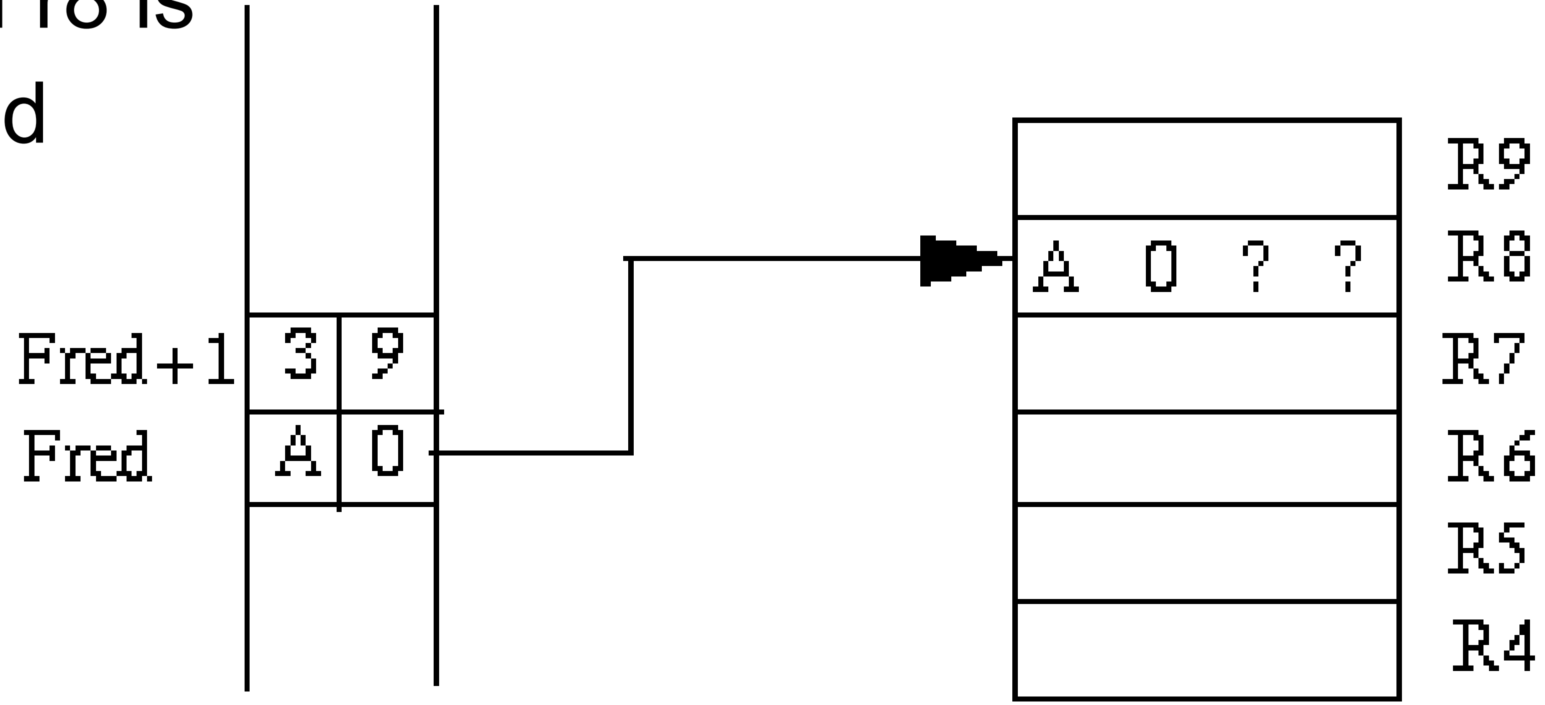
  Flags: None

  The *high order bits* (0-7) of the destination register are loaded with the contents of the 8-bit byte whose location is found by adding the specified *address* and the specified source register.

LODBH Example

lodbh Fred,,r8

Note that the low half of r8 is unchanged



CRAPS Load Instructions

- **LODBL**

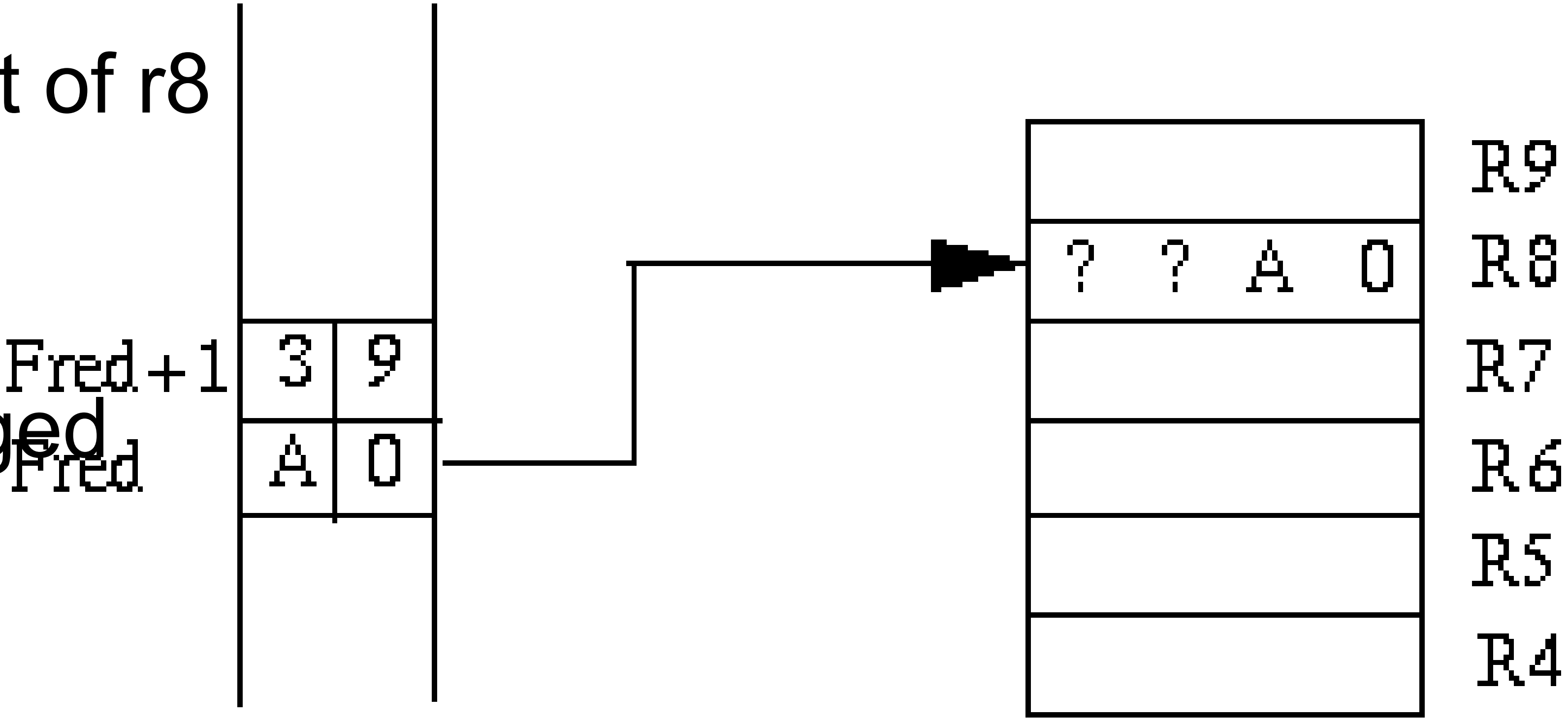
Name:	*Load Byte Low*
Mnemonic:	*LODBL*
Format:	*lodbl address, [src], dest*
Function:	*dest := M[address+src]*
Flags:	*None*

The *low order bits* (8-15) of the destination register are loaded with the contents of the 8-bit byte whose location is found by adding the specified *address* and the specified source register. This register is optional.

LODBL Example

lodbl   Fred,,r8

Note that the high part of r8 is unchanged



CRAPS Load Instructions

- **LODBU**

Name:	*Load Byte Unsigned*
Mnemonic:	*LODBU*
Format:	*lodbu address, [src], dest*
Function:	*dest (8-15) := M[address+src]*
	dest (0-7) := 0
Flags:	*None*

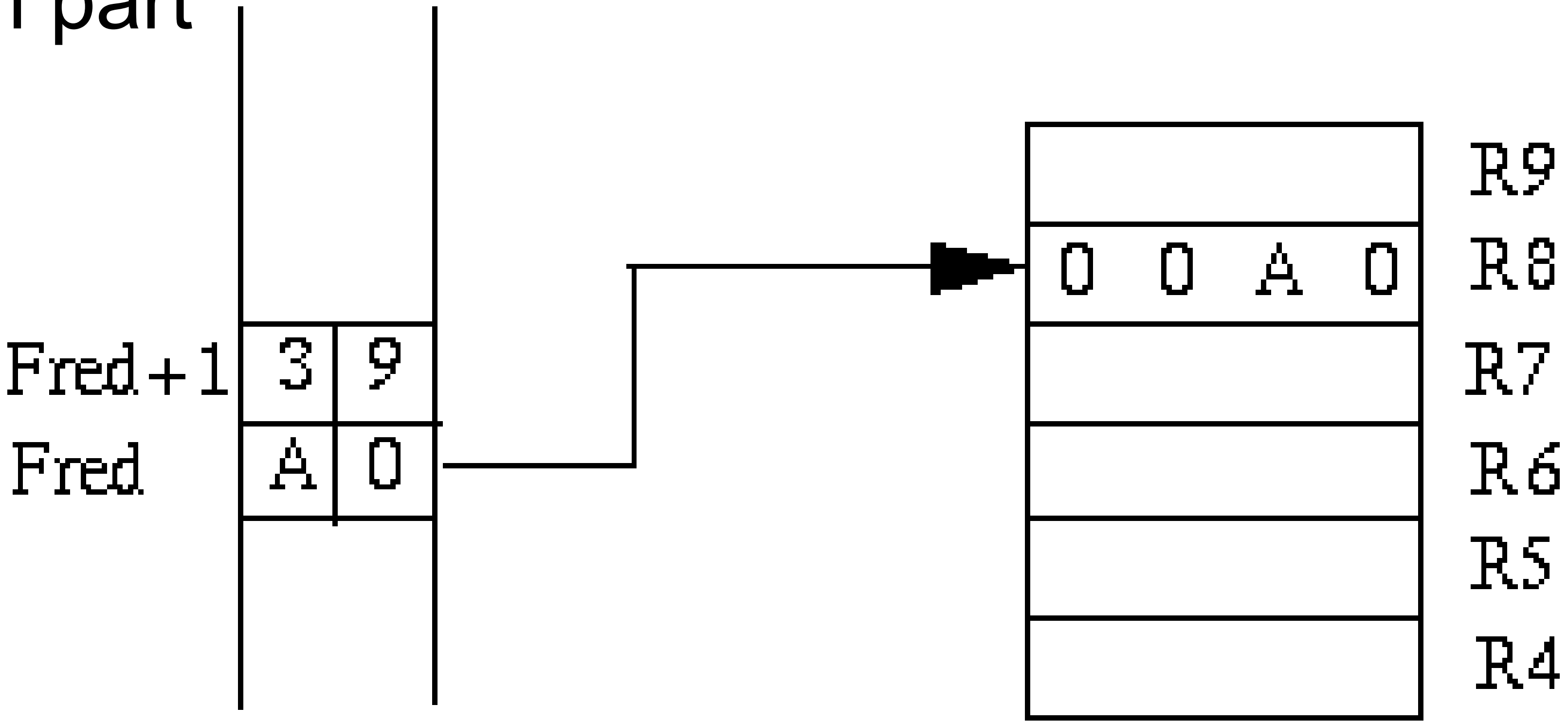
The *low order bits* (8-15) of the destination register are loaded with the contents of the 8-bit byte whose location is found by adding the specified *address* and the specified source register. This register is optional.

The *high order* bits are set to 0; thus the destination register is set to the 16-bit ***unsigned*** numerical value equal to that held in the 8-bit byte in memory

LODBU Example

lodbu fred,,r8

Note the zeros in the high part of r8



CRAPS Load Instructions

- **LODBS**

Name:	*Load Byte Signed*	
Mnemonic:	*LODBS*	
Format:	*lodbs address, [src], dest*	
Function:	*dest (8-15)*	*:= M[address+src]*
	dest (0-7)	*filled with a copy of bit 8 (the "sign" bit)*
Flags:	*None*	

The *low order bits* (8-15) of the destination register are loaded with the contents of the 8-bit byte whose location is found by adding the specified *address* and the specified source register. This register is optional.

The *high order* bits are set to a copy of the "sign" bit in the original byte; thus the destination register is set to the 16-bit ***signed*** numerical value equal to that held in the 8-bit byte in memory.

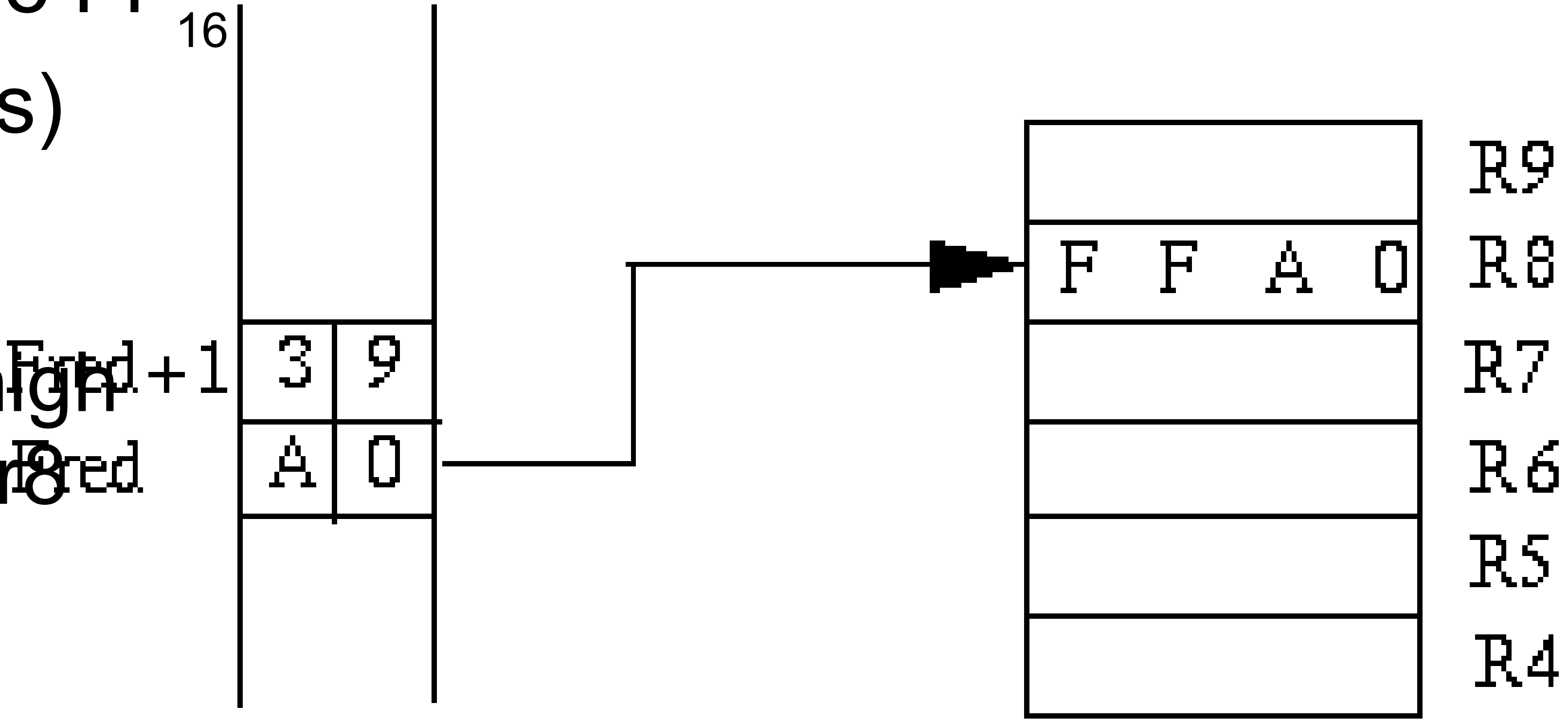
LODBS Example

lodbs fred,,r8

Note the FF$_{16}$

(8 1-bits)

in the high part of r8



CRAPS Store instructions

- **STORE instructions**
  - There are 3 instructions for storing data into memory from a register. (Why 3 and not 5?)
    - STOW
    - STOBH
    - STOBL

CRAPS Store instructions

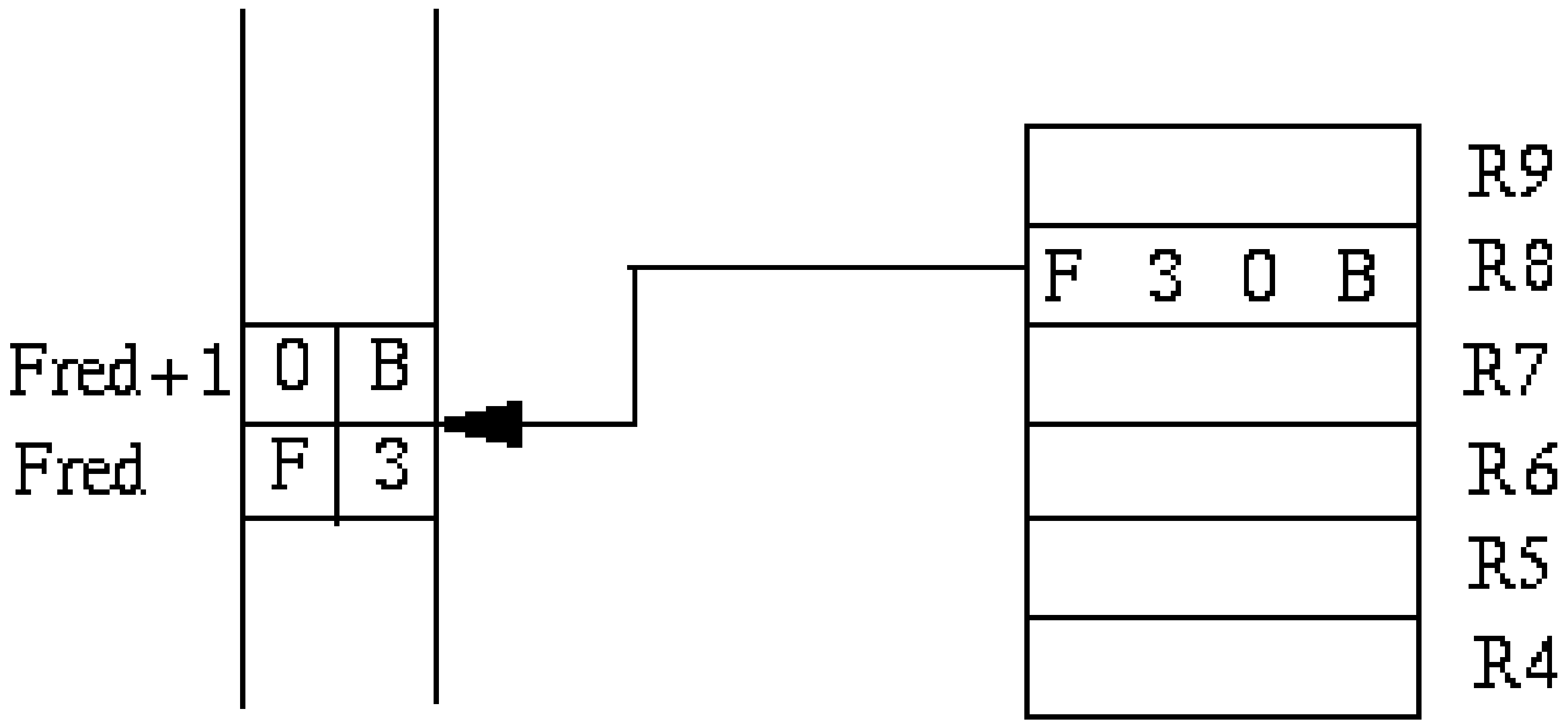
- **STOW**

Name:	*Store Word*
Mnemonic:	*STOW*
Format:	*stow address, [src1], src2*
Function:	*M[address+src1] := src2 (bits 0-7)* *M[address+src1+1] := src2 (bits 8-15)*
Flags:	*None*

The contents of the specified (**src2**) register are stored into the 16-bit word whose location is found by adding the specified *address* and the specified source register. This register is optional.

STOW Example

stow fred,,r8



CRAPS Store instructions

- **STOBH**

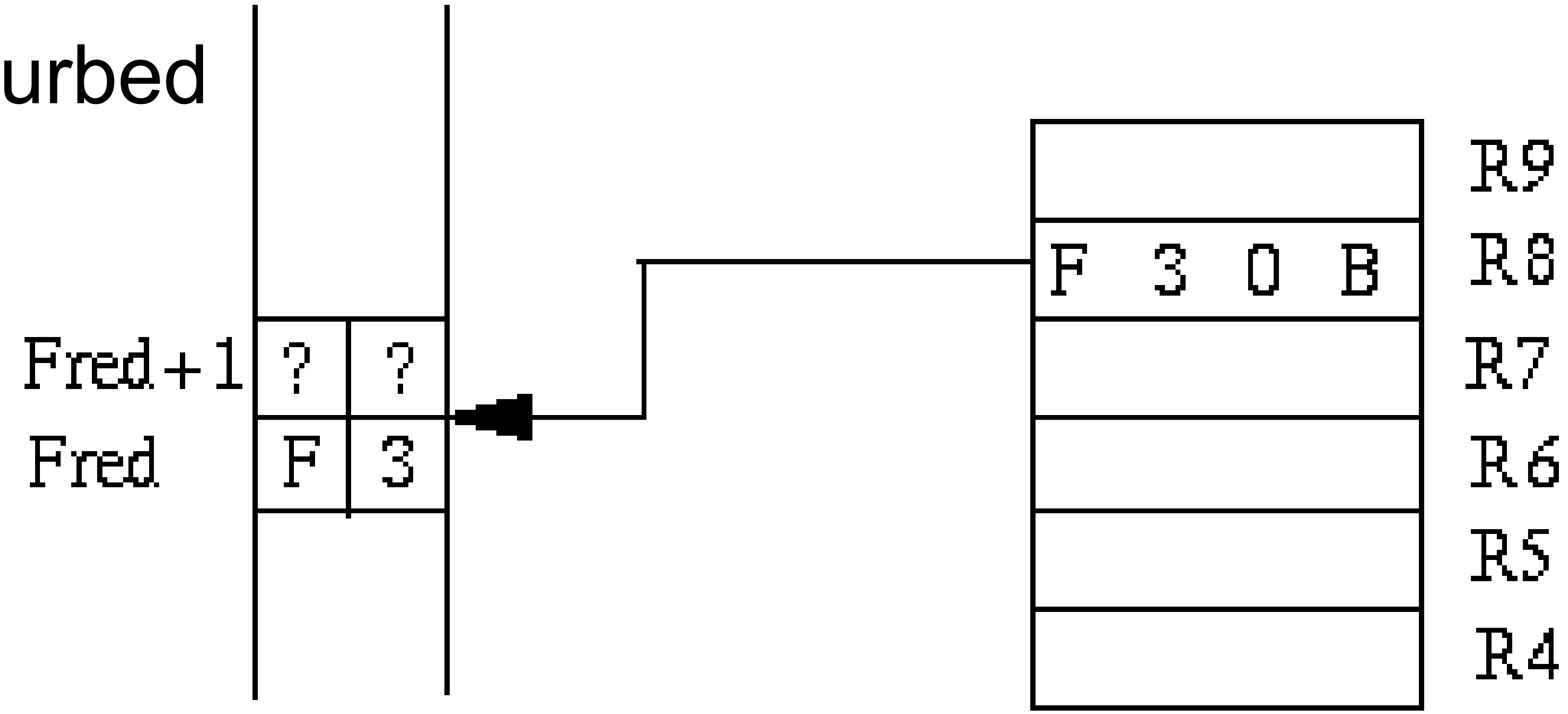
Name:	*Store Byte High*
Mnemonic:	*STOBH*
Format:	*stobh address, [src1], src2*
Function:	*M[address+src1] := src2 (0-7)*
Flags:	*None*

The *high order bits* (0-7) of the src2 register are stored in the 8-bit byte whose location is found by adding the specified *address* and the specified src1 register. This register is optional.

STOBH Example

stobh fred,,r8

Note that Fred+1 is undisturbed



CRAPS Store instructions

– **STOBL**

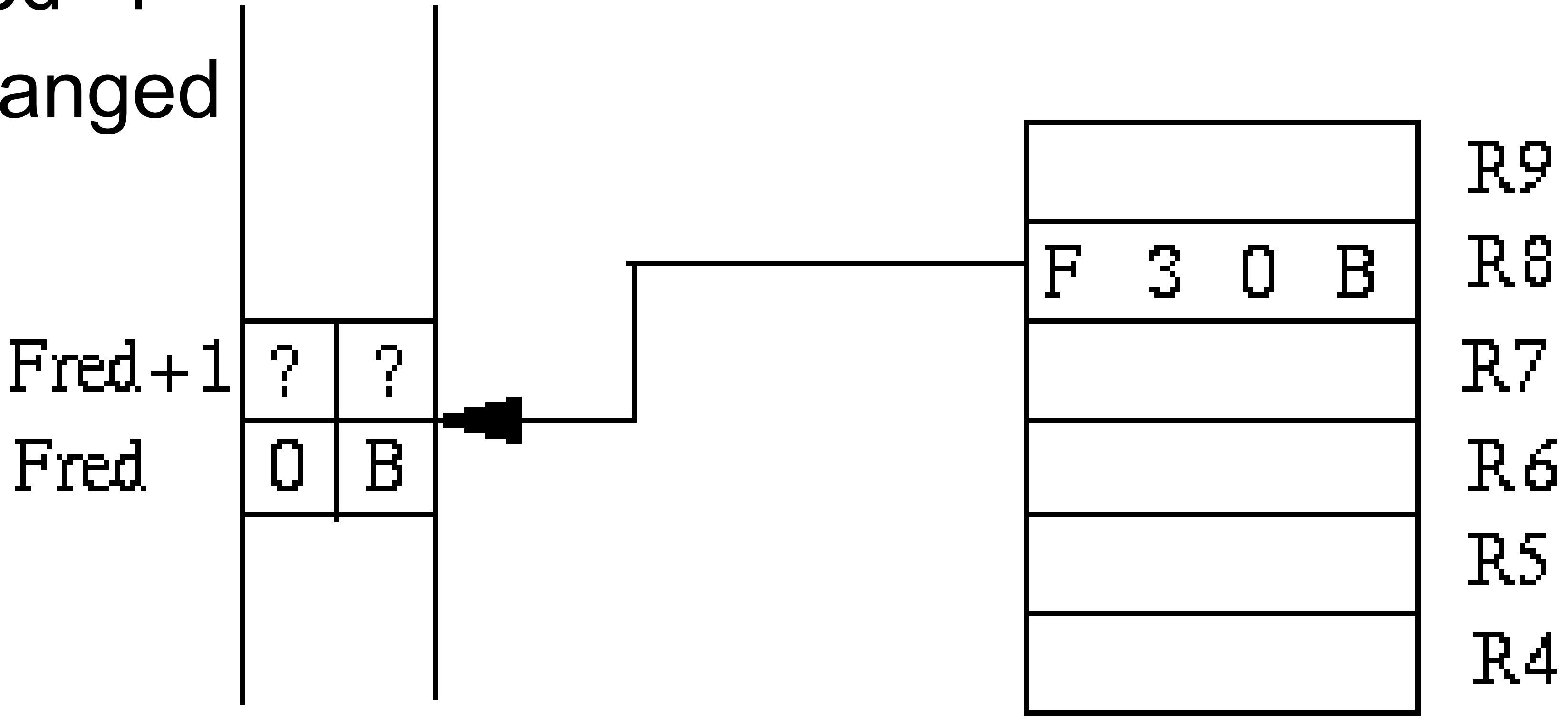
Name:	*Store Byte Low*
Mnemonic:	*STOBL*
Format:	*stobl address, [src1], src2*
Function:	*M[address+src1] := src2 (8-15)*
Flags:	*None*

The *low order bits* (8-15) of the src2 register are stored in the 8-bit byte whose location is found by adding the specified *address* and the specified src1 register. This register is optional.

STOBL Example

stobl fred,,r8

Again notice that Fred+1 is unchanged



CRAPS Arithmetic Instructions

Addition

Name: Add Mnemonic: ADD Format: add src1, src2, dest Function: dest := src1 + src2 Flags: OSZC set	Name: Increment by 1 Mnemonic: INC Format: inc dest Function: dest := dest + 1 Flags: OSZC set
Name: Add with carry Mnemonic: ADC Format: adc src1, src2, dest Function: dest := src1 + src2 + carry flag Flags: OSZC set	

CRAPS Arithmetic Instructions

Subtraction

Name: Subtract Mnemonic: SUB Format: sub src1, src2, dest Function: dest := src1 - src2 Flags: OSZC	Name: Decrement by 1 Mnemonic: DEC Function: dest := dest - 1 Flags: OSZC
Name: Subtract with borrow Mnemonic: SBB Format: sbb src1, src2, dest Function: dest := src1 - src2 – carrry flag Flags: OSZC	

Flow of Control

- **Unconditional branch**

Name:	*Branch*
Mnemonic:	*BR*
Format:	*br target[, src]*
Function:	*PC := target [+ src]*
Flags:	*None*

Example:

```
br      Target          ;jump to instruction at Target
```

Flow of Control

- **Compare**

Name:	*Compare*
Mnemonic:	*CMP*
Format:	*cmp src1, src2*
Function:	*Compare src1 to src2, set flags*
Flags:	*OSZC*
Note:	*A 16-bit immediate value may be used for src2*

- The compare instruction is not strictly flow of control, but it is used extensively to set **flags** which are then tested by conditional branches
- Examples:

```
cmp        r1, r2     ; compare r1 to r2 and set flags
cmp        r1, 10     ; compare r1 to 10 (decimal), set flags
```

Flow Of Control

Name	Branch if condition is met
Mnemonic	Bcc
Function	If appropriate condition is true pc := target + src (Condition flags will have been set by a previous instruction)
Flags	None

Examples

BC	branch if carry flag on	BNC	branch if carry flag off
BO	branch if overflow flag on	BNO	branch if overflow flag off
BS	branch if sign flag on	BNS	branch if sign flag off
BZ	branch if zero flag on	BNZ	branch if zero flag off
BE	branch if equal	BA	branch if abov (unsigned)
BAE	branch if above or equal (unsigned)	BB	branch if below (unsigned)
BBE	branch if below or equal (unsigned)	BG	branch if greater (signed)
BGE	branch if greater or equal (signed)	BL	branch if less (signed)
BLE	branch if less or equal (signed)		

Note: The src register may be omitted (r0 is assumed)

Branching

- There is also the *call* instruction for invoking procedures and subroutines. See below for more details.
- The ***above*** and ***below*** conditions are used for *unsigned* comparisons, *greater* and *less* for *signed.* To see why both are necessary, consider the two numerical orderings of the 8 3-bit patterns:

Signed and Unsigned values

Unsigned	Decimal value	Signed	Decimal value
111	7	011	3
110	6	010	2
101	5	001	1
100	4	000	0
011	3	111	-1
010	2	110	-2
001	1	101	-3
000	0	100	-4

Note: that for unsigned comparisons 111_2 is numerically larger than 000_2, whereas for signed comparisons it is smaller.

Defining Data

- Data is defined (in the *data region)* and initialised using one of three directives.

Name:	Define Bytes
Mnemonic:	DB
Format:	DB init [, init [, init....]]
Function:	Initializes run-time memory to the values specified.

Name:	Define Words
Mnemonic:	DW Format: DW init [, init [, init....]]
Function:	Initializes run-time memory to the values specified.

Name:	Define Doublewords
Mnemonic:	DD Format: DD init [, init [, init....]]
Function:	Initializes run-time memory to the values specified.

Data Initialisation

- The operands for the D? directives consists of a comma-separated list of *initial values*.
- Numeric data:
  - Decimal: any string of decimal digits beginning with a *non-zero* digit, e.g.:

    123, 5094
  - Octal (base 8): any string of octal digits (0-7), *beginning* with 0, e.g.:

    0123
  - Hexadecimal (base 16): any string of hexadecimal digits (0-9, A-F, a-f) preceded by the string *0x, e.g.:*

    0xffff
  - Binary (base 2): any string of zeros and 1s preceded by the string *0b, e.g.*:

    0b10011

Data Initialisation

- Character (string) data:
  - ASCII (DB only): any string of characters enclosed by *either* single or double quotations marks, e.g;

    'abcd'

    "hello sailor!"
- Unitialised storage:
  - ? - uninitialised: this special symbol tells the assembler to reserve one byte (word, doubleword) but not to place an initial value into it
- Repetitions:
  - It is also possible to specify that an initial value should be repeated a certain number of times. The value is placed between [ ] and followed by an asterisk (*) and a repeat count.

Data Definition Examples

```
msg      db   'Hello world!'
```

This directive initializes the 12 bytes *beginning* with the one which will be labeled *msg* (note this name only applies to the *first* byte) to the ASCII string 'Hello world!'

```
bytes    db   ['a'] * 10, ?, [15] * 5
```

This initialises the area beginning with *bytes* to: 10 copies of the ASCII character 'a', a reserved but uninitialised byte, 5 copies of the decimal number 15.

```
words    dw   99, 100, 101, [?] * 10
```

This initialises 3 *words* (16 bits) to 99, 100, 101 (decimal) then reserves, but doesn't initialise, a further 10 words (20 bytes).

```
dwords   dd   0xaa11ff00, ?
```

This initializes a doubleword (4 bytes) to the hexadecimal value AA11FF00 and reserves a further doubleword without initialising it.

Note that the assembler forces words to begin on an even address and double words to begin on a multiple-of-4 address.

CRAB - The CRAPS BIOS

- Interfacing to the outside world (e.g. doing I/O) can only be achieved by calling upon the services of *CRAB* (the CRAPS BIOS).
- This is done using the *SYS* instruction:

Name:	System call
Mnemonic:	SYS
Format:	SYS operand
Function:	The BIOS is called and the function family specified by **operand** is performed. See individual functions for more details.
Flags:	None

- The current version of CRAB contains three functional groups:
  - CONSOLE
  - FSTREAM
  - EXIT

Console I/O

- This family is used to read input from the keyboard and write output to the screen
  - The operand specified in the SYS instruction is 1.
  - There are four currently-defined functions within this family; each is specified by placing a sub-function code into the lower half (8-15) of register R1 before executing the SYS instruction
  - GETCHAR (0)
    - Read a single character from the keyboard. The character is placed into the R2 register (bits 0-7 = 0, bits 8-15 = character)
  - Example:

```
GETCHAR        EQU    0
....
mov            GETCHAR, r1
sys            CONSOLE
; R2 now contains character from keyboard
```

Console I/O

- GETSTR (1)
  - Read a line (string) from the keyboard. R2 must contain the address of the first character of the buffer area. This area is formatted as follows:

– Byte 0	Maximum input length (in bytes) including CR
– Byte 1	Set to actually input length (in bytes) excluding CR
– Byte 2-	An area sufficient to contain the maximum input string (the CR is converted to a NUL – 0x00 - byte)

Console I/O

- Example:

```
                include     System ; useful EQUates
MAXLINE         EQU         80
                data
input           db          MAXLINE+1, [?] * MAXLINE+2
                ...
                code
                ...
                mov         GETSTR, r1
                lea         input, r2
                sys         CONSOLE
```

- After the SYS instruction is complete, the byte ***input+1*** will contain the input character count. (Less the CR - so an empty input will have a length of 0).

Console I/O

- PUTCHAR (2)
  - write a single character to the screen. The character must be placed into R2 (bits 8-15) before the SYS instruction.
    - *Example:*

```
include  System
...
mov      'D', r2             ; want to print a D
mov      PUTCHAR, r1         ; just print one char
sys      CONSOLE             ; print it
```

Console I/O

- PUTSTR (3)
  - write a **NUL-terminated** string to the screen. The address of the *first* byte of the message must be placed into R2 before the SYS instruction. The message must end with a **NUL** (i.e. zero) byte. (This is often known as an ***ASCIIZ*** *string*)
    - Example:

```
          include         System
          data
message db                "hello world!", 0
          ...
          code
          ...
          mov             PUTSTR, r1
          lea             message, r2
          sys             CONSOLE
```

File Stream I/O

- This family is used for reading and writing files
- The value specified in the SYS instruction is 2.
- There are three currently-defined functions within this family; each is specified by placing a sub-function code into the lower half (8-15) of register R1 before executing the SYS instruction

File Stream I/O

- OPEN (0)
  - R2 contains the address of an ASCIIZ string which is the UNIX path of a file to be opened.
  - On return from the SYS instruction R1 will contain 0 if the open succeeded, and -1 if it failed.
  - On success R2 contains a number called the *file* ***handle*** which is used in all other FSTREAM calls.
  - On failure R2 contains an error code.
- CLOSE (1)
  - R2 contains the file handle returned by the OPEN call.
  - On return from the SYS instruction R1 will contain 0 if the close succeeded, and -1 if it failed.
  - On success R2 contains the status returned by the UNIX system call.

File Stream I/O

- READ (2)
  - R2 contains the file handle returned by the OPEN call.
  - R3 contains the address of a buffer into which the input data will be placed.
  - R4 contains the length of the buffer in bytes.
  - The next line is read from the specified file and put into the buffer. The ending linefeed character is ***not*** removed by the call; the terminating ASCII NUL is added to the string.
  - On return from the SYS instruction R1 will contain 0 if the read succeeded, and -1 if it failed.
  - On success R2 contains the number of characters read; if this is zero it means end of file. On failure it contains the error code.

Example FSTREAM Program

```
program   tfstream
          STACK 20
          INCLUDE    System
MAXLINE   EQU   80              ; maximum line length
          DATA
infile    db    'testdata.in', 0 ; NUL terminator
buffer    db    [?] * MAXLINE + 1     ; allow for NL
handle    dw    ?

          CODE
tfstream
          mov   OPEN, r1
          lea   infile, r2
          sys   FSTREAM
          stow  handle,,r2
copyline
          mov   READ, r1
          lodw  handle,,r2
          lea   buffer, r3
          mov   MAXLINE, r4
          sys   FSTREAM         ; read line
          cmp   r2, 0
          be    EOF             ; nothing there
          mov   PUTSTR, r1
          lea   buffer, r2
          sys   CONSOLE
          br    copyline
EOF
          mov   CLOSE, r1
          lodw  handle,, r2
          sys   FSTREAM
          sys   EXIT
          END
```

SYSTEM

- The system group includes only one function
  - **Program Termination**
    - The value specified in the SYS instruction is 65535 (0xFFFF)
    - The program is terminated

Flow of Control (revisited)

- The conditional and unconditional branch instructions give us all we really need in the way of control flow constructs.
- The reason for this certainty is simple: in the 1960s the *Böhm-Jacopini theorem,* proved that any program can be constructed out of just three control flow types:
  - **Sequence**
    - the "normal", one instruction follows the previous, flow. As used in most major programming languages
  - **Selection**
    - A choice between two alternatives. Corresponds to *if ... then ... else* in most languages
  - **Iteration / Repetition**
    - Test-at-the-top form of repetition. Equivalent to *while* in most languages.
- All of these constructs can easily be achieved in CHASM

Sequence

- The normal program flow in the CPU's processing cycle (*Fetch-Decode-Execute*) is sequential

```
mov     2, r2
add     r2, r2, r2
sub     r1, r2, r3
```

  - These three instructions will be executed one after another.

Selection

- A selection can be programmed as follows, the equivalent of the C/C++/Java:

```
if (a == b)
        // something interesting goes here
else
        // something else interesting
        // common thread resumed here
```

can be achieved in CHASM as follows:

Selection

```
    lodw    a,,r1
    lodw    b,,r2
    cmp     r1,r2       ; compare a to b
    bne     Else        ; !=, skip to else
                        ; here if a == b
    ...                 ; interesting stuff
    br      Common      ; skip else part
Else
    ...                 ; other stuff
Common
```

Note the way in which we have to branch (jump) over the code we don't want to execute

Iteration / Repetition

- The equivalent of the C/C++/Java *test-at-the-top* loop:

  ```
  for (r1=0; r1 <= 15; r1++)
  {
    // loop body
  }
  ```

  can be achieved in CHASM as:

Iteration / Repetition

```
      mov 0, r1       ; initial value
Again                 ; Top of loop
      cmp r1, 15      ; finished yet?
      ba  Done        ; r1 > 15, so yes
      ...             ; something useful here
      inc r1          ; r1++
      br  Again       ; check if we're done
Done
```

Note the way the loop is implemented using two branch instructions:

> a conditional forward jump at the top of the loop
> an unconditional backward jump at the bottom

Iteration / Repetition

- Many languages also provide a *test-at-the-bottom* loop; the C/C++/Java:

  ```
  do
  {
     // loop body
     r1++;
  } until (r1 > 10)
     // Code after loop
  ```

  is achieved in CHASM as:

Iteration / Repetition

```
Again
    ...                 ; loop body
    inc     r1          ; add 1
    cmp     r1, 10      ; > 10 yet?
    bbe     Again       ; not yet
    ...                 ; code after loop
```

Note that only one (backward, conditional) jump is required

Addressing

- **Based addressing**
  - Consider copying N words of memory from one location to another

```
      DATA
A     dw          [?] * 3      ; Array 3 words (6 bytes)
B     dw          [?] * 3      ; And another
      ...
      CODE
      ...                      ; A gets initialised here
      lodw       A,,r1         ; Copy first word to r1
      stow       B,,r1         ; Store in first word
      lodw       A+2,,r1       ; Copy second word to r1
      stow       B+2,,r1       ; Store in second word
      lodw       A+4,,r1       ; Copy third word to r1
      stow       B+4,,r1       ; Store in third word
```

Observations

- This uses two instructions (both code size and execution time) per word copied.
- What if the array size changes?
  - To 10?
  - To 100?
  - To 1000?
- Evidently this method soon becomes unwieldy and error-prone - the program code expands linearly with the size of the array.
- The solution: *based* or *pointer* addressing

Based/Pointer Addressing

```
        ...
ArrSize  EQU 100 ; Makes it easier to change later
        ...
        DATA
A       dw    [?] * ArrSize ; ArrSize Words
B       dw    [?] * ArrSize ; Ditto
        ...
        CODE
        ...
        lea   A, r1          ; Address first in word to R1
        lea   B, r2          ; Address first out word to R2
        mov   ArrSize, r3 ; Loop count to R3
Copy
        cmp   r3, 0          ; exhausted yet?
        be    Done           ; yup
        lodw  0, r1, r4      ; Get in word to R4
        stow  0, r2, r4      ; Store in out word
        add   r1, 2, r1      ; Next in word
        add   r2, 2, r2      ; Next out word
        dec   r3             ; count down
        br    Copy           ; Loop until ArrSize copied
Done
```

Observations

- Although this is 11 instructions long, if our array is larger than 5 words, this version will be physically smaller than the simple-minded method above.
- It will be slightly slower, because each word copied now takes 8 instructions instead of 2, *but* those two access memory which is far slower than accessing registers and so this algorithm will probably perform almost (and possibly just) as fast.
- Recall that the description of the function of LODW was:

  dest (bits 0-7) := M[address+src]

  dest (bits 8-15) := M[address+src+1]

  in other words, the address in memory from which we load the data is calculated by adding the value specified in the first operand, to the contents of the named register in the second. If this operand is omitted (as in our initial examples) it defaults to R0 which, you will recall, is always zero.

Observations

- The same *effective address* calculation is done by the store instructions.
- So, the first time through the loop, the address calculated by the LODW is

  0 + R1

  and we have loaded the *address* of A into R1 already
- Similarly, the address calculated by the STOW is

  0 + R2
- At the end of the loop we add 2 to **both** R1 and R2, so R1 now has the address of (we often say ***points to***) the byte which is 2 *after* A and R2 the address of the byte 2 after B.
- Using registers in this way, i.e. putting the memory address of the desired byte in the register and using *based addressing* is how *pointers* are implemented in high level languages like C and C++ (and how *references* are implemented in C++ and Java, among others).

Indexed addressing

- The previous example wastes one of the scarcest resources in the CPU - a register.
- The key to this is to notice that both pointer registers stay "in step" as the code moves through the arrays in parallel. If only we could use a single register to keep track of how far into each array we have got...
- Here is a first pass using ***indexed addressing***:

Indexed addressing

```
         ...
ArrSize  EQU 100
         ...
         DATA
A        dw    [?] * ArrSize ; ArrSize Words
B        dw    [?] * ArrSize ; Ditto
          ...
         CODE
         ...
         mov  0, r1        ; First word of A is
                           ; 0 bytes in
                           ; Likewise first word of B
         mov  0, r2        ; Loop counter
Copy
         cmp  r2, ArrSize  ; Done yet?
         bae  Done         ; yes
         lodw A, r1, r3    ; Get in word to r3
         stow B, r1, r3    ; Store in out word
         add  r1, 2, r1    ; Next word
         inc  r2           ; count this iteration
         br   Copy
Done
```

Observations

- Note that this version uses registers R1, R2 and R3 - we have saved R4 (which the based version used) for some other purpose and two instructions, making the program slightly smaller and marginally faster - but this is precisely the reason for programming in assembler in the first place, to squeeze the last ounce (or gram) of performance out of the CPU.
- But we can still improve on this - and in two ways.

Observations

- First, we note that the array size is always non-zero (not much point in moving zero-length arrays around in memory - how could you tell anyway?), which means that we will *always* go through the loop body at least once, so we should really use a test-at-the bottom loop.
- This will save two instructions worth of CPU time, because we shall not perform the *CMP* and *BAE* instructions before the first time through the loop.

Observations

- Even better though, we can save *another* register.
- Note that we count the number of times we go through the loop (in R2) by adding 1, starting from zero, until it gets to *ArrSize*.
- But we also initially set R1 to zero and add 2 to it each time through the loop. So when we have finished looping the value in R1 will be *Arrsize * 2*

Second version

```
                ...
ArrSize         EQU    100
                ...
                DATA
A               dw  [?] * ArrSize ; ArrSize Words
B               dw  [?] * ArrSize ; Ditto
                ...
                CODE
                ...
                mov     0, r1                     ; First word of A is
                              ;  0 bytes in
                                                  ; Likewise first word
                              ;  of B
Copy
                lodw    A, r1, r2           ; Get in word to r2
                stow    B, r1, r2           ; Store in out word
                add     r1, 2, r1           ; Next word
                cmp     r1, ArrSize * 2     ; Done yet?
                bb      Copy                      ; no
```

Comparison of based and indexed addressing

Method	Registers	Instructions
Based	4	11
Indexed_1	3	9
Indexed_2	2	6

Note how careful choice of addressing technique and algorithm can reduce the resource usage of the program: 50% of registers and 45% of instructions.

Logic

- Computers are constructed using combinations of simple logical functions, following the laws established by English mathematician George Boole.
- The fundamental operations are:
  - and
  - inclusive or
  - exclusive or (xor)
  - not
- The operations are indeed fundamental: all other operations, including arithmetic, can be built from these. (In fact it is possible to construct all of them using only NAND (not and) or NOR (not or) gates, but that's another story...)
- Given a thorough understanding of these operations it is possible to test and set individual bits or groups of bits within a byte.

Logic

A	B	AND	OR	XOR	NOT A
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	0	0

AND

Name: *Logical AND*
Mnemonic: *AND*
Function: *dest := src1 AND src2*
Flags: *OC set to zero, SZ set*
Example:

```
mov    0x3f46, r1
mov    0x179a, r2
and    r1, r2, r3
```

R1	0011111101000110
R2	0001011110011010
R1 AND R2 => R3	0001011100000010
Flags	S = 0, Z = 0

And

- The AND instruction is frequently used to isolate certain bits in a word or byte. This technique is usually referring to as *masking* or *masking out*.

```
Bit15       EQU     0b01
Bit14       EQU     0b10
            ...
            lodw    Fred,,r1
            and     r1, Bit14+Bit15, r1
            bnz     On              ;either b14 or b15 (or both) is on
```

- Another using of masking is to ensure that certain bits in a word or byte are off - i.e set to 0.

```
NotBit14    EQU   0b1111111111111101
            ...
            and   r1, NotBit14, r1          ;Ensure Bit14 off
```

Test

Name:	*Logical compare*
Mnemonic:	*TEST*
Function:	*r0 := src1 AND src2*
Flags:	*OC set zero, SZ set*
Example:	

```
mov  0x3f46, r1
mov  0x179a, r2
test  r1, r2
```

R1	0011111101000110
R2	0001011110011010
R1 Test R2	0001011100000010
Flags	S = 0, Z = 0

Test

- The TEST instruction is useful in those circumstances when we need to check whether one or more bits within a word or byte are set, but don't want to keep the result, which is of no further use

```
Bit12 EQU       0b1000
      ...
      lodw      Fred,, r9
      test      r9, Bit12
      bnz       Bit12On     ; Bit 12 is set
```

Test

- Note that the following code will also test whether bit 12 is set in the word Fred, but will consume a register with a modified copy of the word, whereas TEST leaves the original untouched

```
Bit12 EQU       0b1000
        ...
        lodw        Fred,, r12
        and         r12, Bit12, r13
        bnz         Bit12On               ; Bit 12 is set
```

OR

Name:	*Logical inclusive OR*
Mnemonic:	*OR*
Function:	*dest := src1 OR src2*
Flags:	*OC set to zero, SZ set*
Example:	

```
mov  0x3f46, r8
mov  0x179a, r9
or   r8, r9, r10
```

R8	0011111101000110
R9	0001011110011010
OR R8, R9 => R10	0011111111011110
Flags	S = 0, Z = 0

OR

- The commonest use of the OR instruction is to ensure that one or more bits in a word or byte are set on - i.e. to 1.

```
Bit11 EQU       0b10000
Bit10 EQU       0b100000
        ...
        OR      r2, Bit10+Bit11,r2 ; Turn bits 10 and 11
                                   ; of r2 on
```

OR

- A frequent sub-case is that of converting an uppercase alphabetic character to lowercase. In the ASCII character set each uppercase character differs from its lowercase equivalence in a single bit position.

```
Bit10 EQU      0b100000
          ...
          mov      'A', r8
          or       r8, Bit10, r8        ; Turn 'A' into 'a'
```

- Reversing this process involves turning bit 10 off, which requires an AND (or XOR) instruction.

XOR

Name: *Logical exclusive or*
Mnemonic: *XOR*
Function: *dest := src1 XOR src2*
Flags: *OC set zero, SZ set*
Example:

```
mov  0x3f46, r13
mov  0x179a, r14
xor   r13, r14, r10
```

R13	0011111101000110
R14	0001011110011010
OR R13, R14 => R10	0010100011011100
Flags	S = 0, Z = 0

XOR

- The XOR instruction is frequently used to 'flip' a bit - i.e. turn it from 0 to 1 or 1 to 0.

```
Bit8    EQU     0b10000000
        ...
        lodbl   Jim,,r1
        xor     r1, Bit8, r1       ; Flip bit 8
        stobl   Jim,,r1
```

- Occasionally the following code is seen in programs - why?

```
        xor     r1, r1, r1
```

- While the following, of essentially curiosity value, exchanges the contents of two registers *without* the use of any intermediary storage.

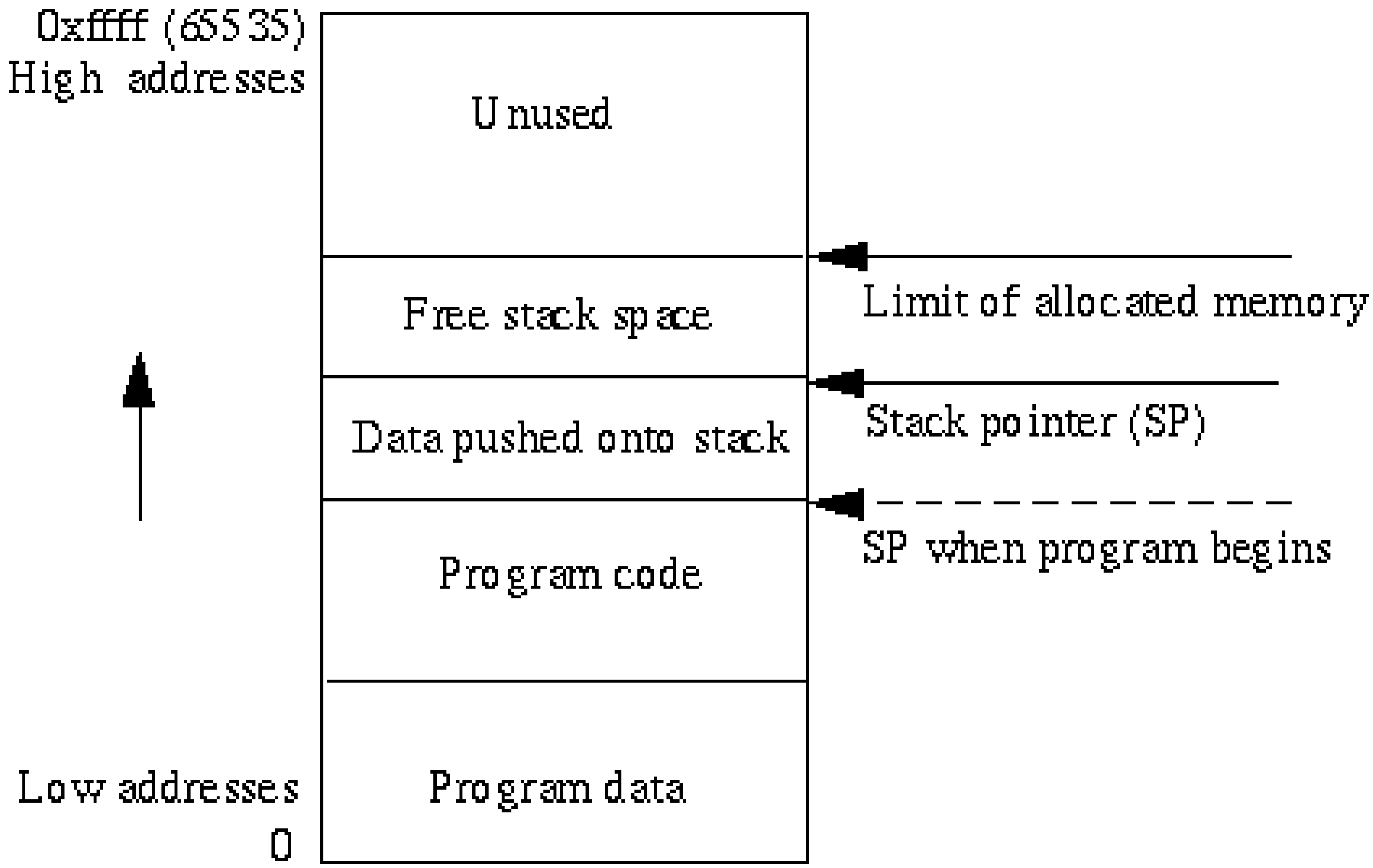
```
        xor     r1, r2, r1
        xor     r1, r2, r2
        xor     r1, r2, r1
```

Stacks

- One of the Fundamental Structures of Computer Science
- The stack is classically defined as a *LIFO* (Last In First Out) data structure, with two main operations:
  - Push
    - Put an item onto the top of the stack.
  - Pop
    - Remove the top item from the stack.
- Stacks come in various flavors; some CPUs provide little or no support for stack-based addressing, others - such as the CRAPS - provide direct support in the level 2 machine code.

Stacks

- Memory layout (including stack) of program in CRAPS memory



Stacks

- Note that when the image file is loaded into memory, the CRAPS *loader* sets the SP register (R15) to the address of the last word *before* the stack. This is because the *push* instruction first increments SP before storing in memory. Read on.

Stack Manipulation Instructions

PUSH

Name:	*Push operand onto stack*
Mnemonic:	*PUSH*
Function:	*sp := sp + 2; M[sp] := src*
Flags:	*None*

POP

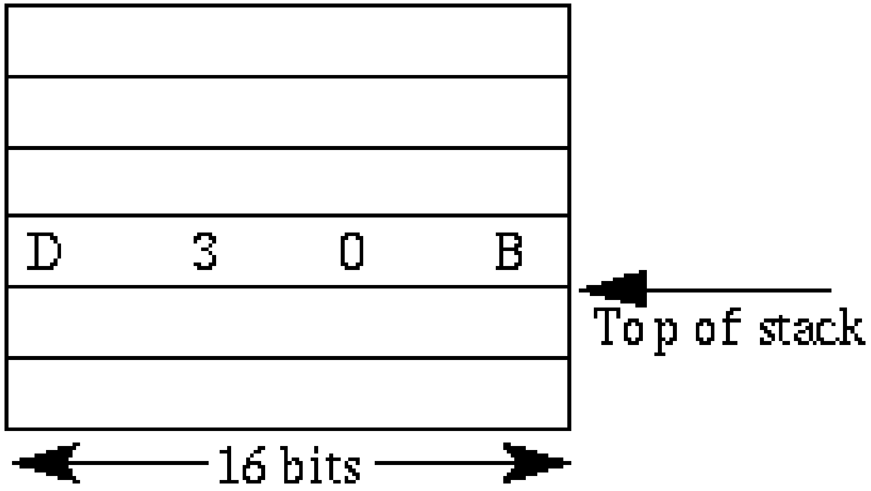
Name:	*Pop word from stack*
Mnemonic:	*POP*
Function:	*dest := M[sp], sp := sp - 2*
Flags:	*None*

Stack Manipulation

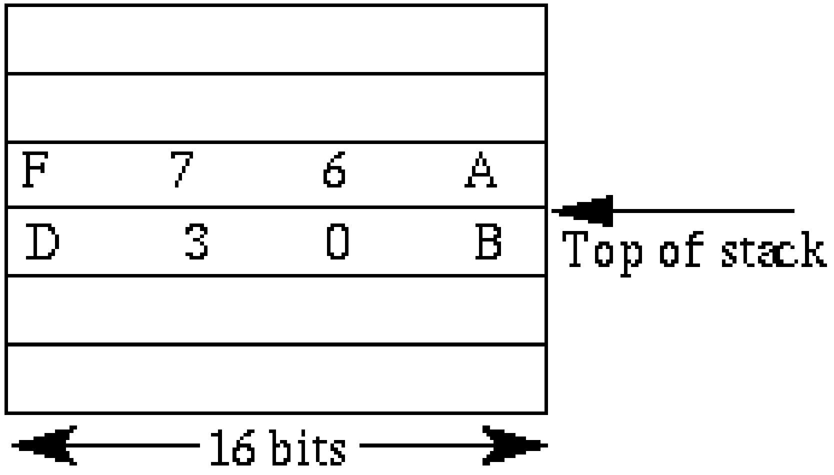
- Note that only 16-bit quantities can be pushed onto or popped off the stack.
- Note that both PUSH and POP are two-stage operations
- Note also the symmetry between the instructions: PUSH decrements SP ***before*** storing data onto the stack; POP retrieves the stack top and ***then*** increments SP.

Stacks

The Effect of a Push on the Stack



Before PUSHing



After PUSHing

```
mov   0xf76a, r1
push  r1
```

Subroutines and Stacks

- The stack is also used by the CALL and RETurn instructions.
- When a separate procedure is called, the flow of control is interrupted, but in a controlled fashion. Unlike the various branch instructions, call must *remember* where it came from.

```
        lodw    Fred,,r1
        call    Julie
        mov     r1, r3
        ...
Julie                   ; Beginning of procedure Julie
        sub     r2, r4, r5
         ...
         ret             ; Go back ...
```

Subroutines and Stacks

- The result of the CALL instruction is that the next instruction to be executed is *not* the one immediately succeeding the CALL, but the one at location *Julie*, i.e. the SUB instruction.
- Moreover, when the *Julie* procedure is completed, control should now return to the MOV instruction located immediately after the CALL.
- Although, as we shall see later, there are several different ways to preserve this *Return Address*, the most useful - and the way CRAPS does it - is to save it onto the stack.

CALL and RETurn

- Subroutine Linkage

  The stack is used by the CALL and RETurn instructions for saving and restoring the *return address* (i.e Program Counter register) of the caller.

Name:	*Call procedure*
Mnemonic:	*CALL*
Function:	*PC pushed onto stack, PC := target*
Flags:	*None*

Name:	*Return from procedure*
Mnemonic:	*RET*
Function:	*PC popped from stack*
Flags:	*None*

Procedures and Parameter Passing

- When we call a procedure we shall frequently wish to pass parameters to that procedure. Which naturally raises the question:

  Where do we put the parameters?

- A number of solutions have been used over the years, such as:
  1. A Dedicated Register
     - This is the method by the various flavors of the SYScall instruction, when calling on the BIOS.
     - *Problem:* there is only a limited number of registers; this method is not practical for procedures which require a larger number of parameters than there are registers, or for high-level languages, where there is no restriction on the number of parameters.

Procedures and Parameter Passing

2. Dedicated Memory Location
   - A feasible solution, but prevents the easy construction of *recursive* procedures.

3. The Stack
   - If parameters are pushed onto the stack, they can be easily - well, reasonably easily - retrieved by the called procedure. Moreover, the use of the stack ensures that a recursive call will push fresh values onto the stack in a different physical location.

(Note that the dedicate register and dedicated memory locations schemes have also been used in real CPUs for storing the return address)

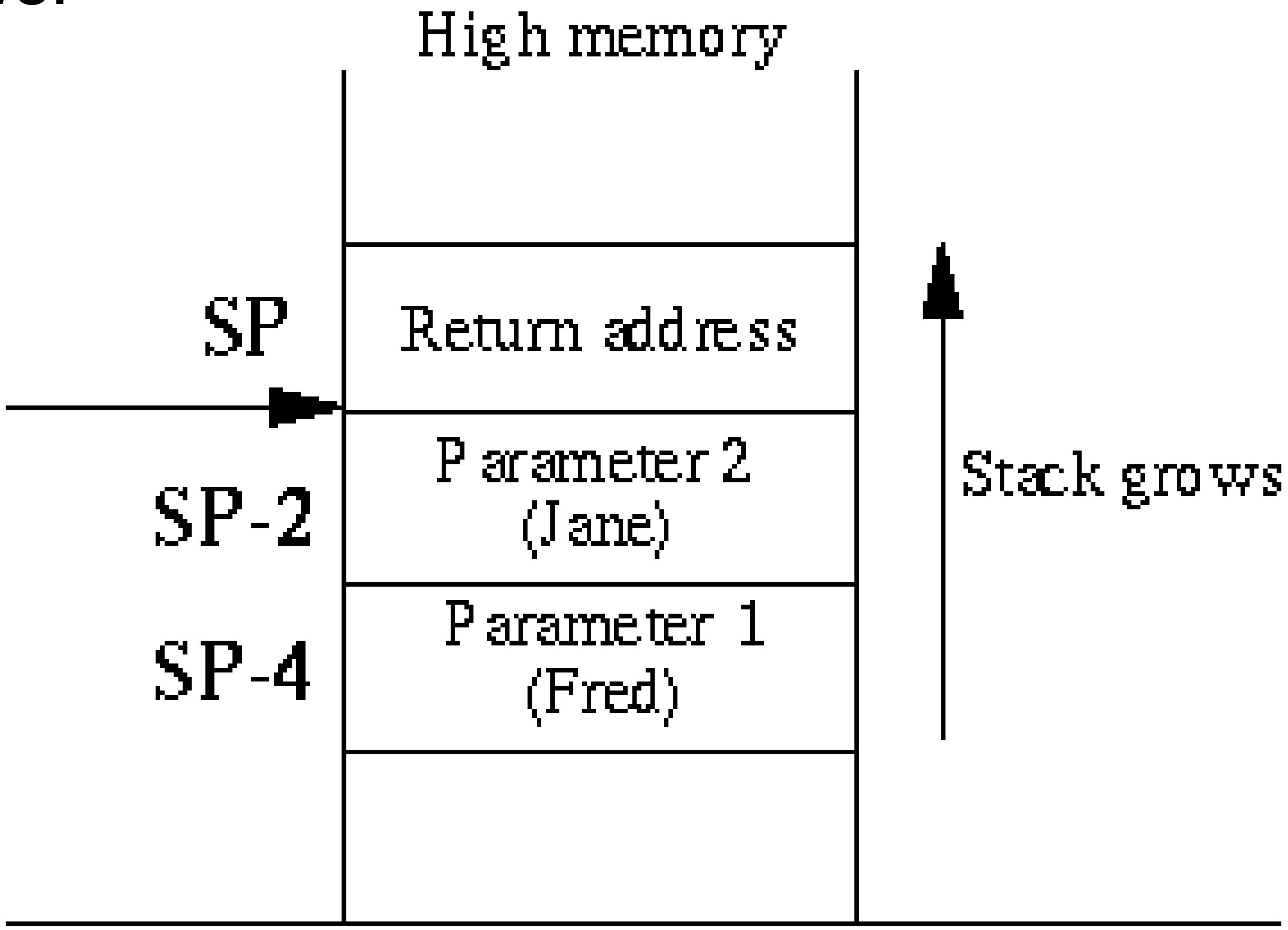
Retrieving Parameters from the Stack

- Pushing parameters onto the stack is simple, but how do we retrieve them?

```
lodw    Fred,,r1
push    r1          ; push first parameter
lodw    Jane,,r2
push    r2          ; push second parameter
call    Proc1
```

Stack

- When we enter the procedure *Proc1* the stack looks as follows:



Hmmm…..

- If we attempt to get at the parameters by POPping the stack, we shall lose our return address - or, at the very least, have to write code to push it back again, like this:

```
Proc1
        pop     r1 ; return address
        pop     r2 ; get parameter 2
        pop     r3 ; get parameter 1
        push    r1 ; put return address back
        ....
        ret
```

which is hardly what we might call elegant – or efficient

What to do

- The solution is found in the register set. We can use the SP (Stack Pointer) register (actually R15) to access the stack.
- We can now access the parameters by observing that:
  - The return pointer is at the location 'pointed at' by SP, (we shall write 'at SP' from now on)
  - Parameter 2 is at SP-2
  - Parameter 1 is at SP-4

The Solution?

```
Proc1
    lodw        -4, sp, r1 ;first parameter to r1
    lodw        -2, sp, r2 ;second parameter to r2
    ....
    ret
```

which is fine (as far as it goes) but violates the “the only unnamed constants should be 0 and 1” rule, so...

A Better Solution

```
Param1 EQU -4
Param2 EQU -2
...
Proc1
          lodw  Param1, sp, r1 ;first param to r1
          lodw  Param2, sp, r2 ;second param to r2
          ...
          ret
```

Note that this leaves the return address in the stack.

Keeping track of SP

- Unfortunately the previous "solution" ignores some “inconvenient truths.” (You can call me Al)
  - The first is that we may wish to use the stack ourselves during the called procedure. Each time anything is PUSHed onto the stack, the value in the SP register changes.
  - So, at procedure entry parameter one can be found in SP-4; after a single PUSH, it would now be in SP-6.
- Does this mean that we simply cannot use the stack inside a procedure?
  - No! The solution is to use the FP (Frame Pointer) register (aka R14) - this is what it's for.
  - So, the obvious solution is simply to copy SP into FP, then any changes in SP won't affect FP and so we have a fixed point of reference in the stack.

Keeping Track of SP

```
Param1  EQU     -4
Param2  EQU     -2
            ...
Proc1
        mov     sp, fp              ; copy sp
        ...                         ; other instructions
                                  ; perhaps a PUSH or 2
        lodw    Param1, fp, r1      ; first param to r1
        lodw    Param2, fp, r2      ; second to r2
        ...
       ret
```

Quis custodiet ipsos custodes?

- But if our procedure is using FP to access parameters, perhaps the procedure which called *us* was doing the same. If we simply copy SP into FP we destroy the caller's "fixed point of reference" into the stack, so after our procedure returns our caller won't be able to find its own parameters anymore.
- Oh ****!

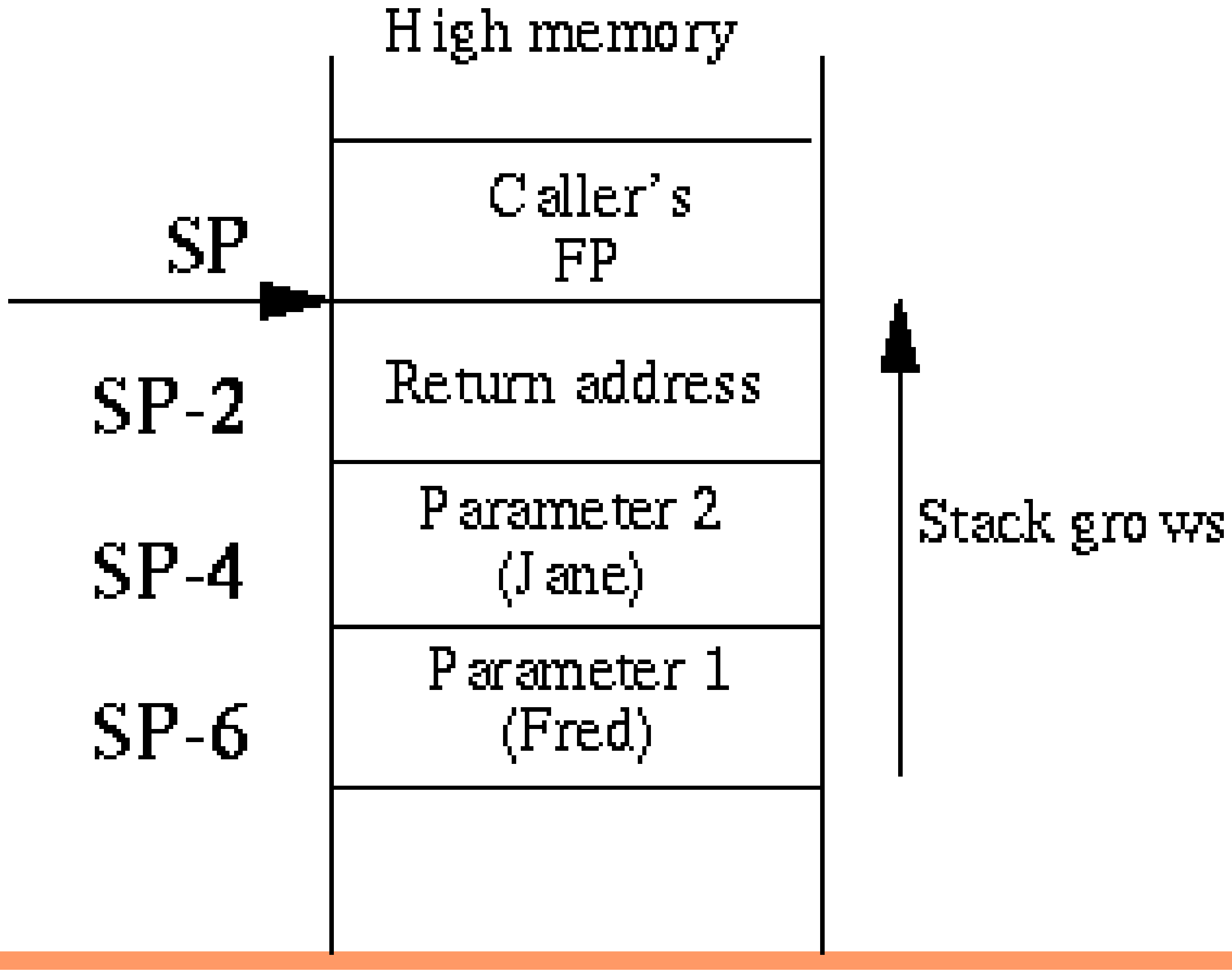
Fixing it For FP

- **Saving the Caller's FP**
  - We must therefore save our caller's FP and what better place than on the stack?

```
Proc1
        push  fp              ; save caller's FP
        mov   sp, fp          ; copy sp
        ...
        pop   fp              ; restore caller's FP
        ret
```

Fixing it For FP

- After the two entry instructions (push, mov) – often called the preamble - the stack looks like this:



Interfacing to High Level Languages

- **Left and Right Pushers**
  - High Level Languages almost all use the stack to pass parameters to subprocedures (aka subroutines, aka functions). Notable exceptions are FORTRAN and COBOL, although individual implementations may use the stack anyway, even though recursion is not part of the language definition.
  - Generally speaking we can classify languages by the **order** in which they push parameters onto the stack

Left and Right Pushers

- Consider this subroutine call:

  test (p1, p2);

  In Pascal/Modula-2/Modula-3/Oberon this will generate something like:

```
lodw p1,,r1
push r1
lodw p2,,r1
push r1
call test
```

  Whereas in C/C++/Java the same code will generate:

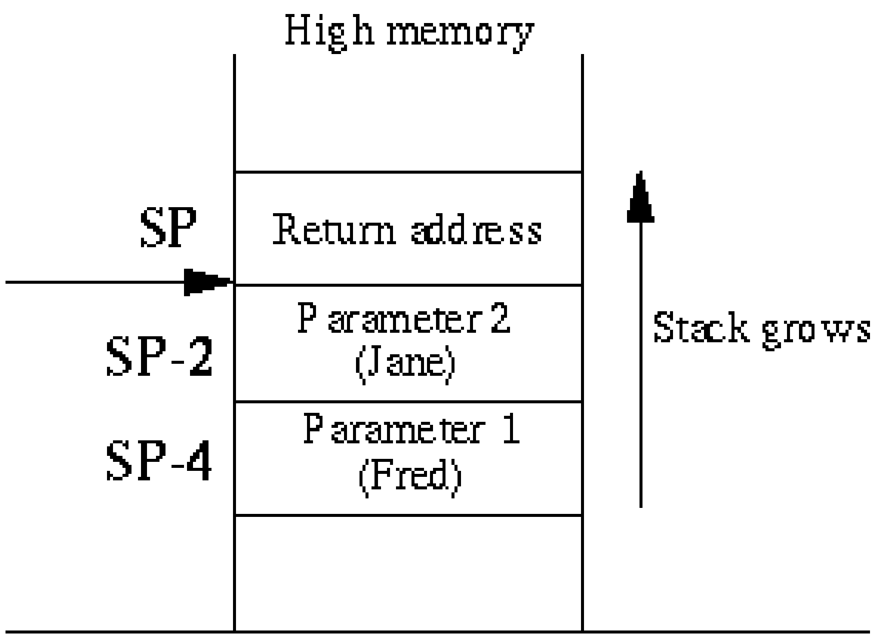
```
lodw p2,,r1
push r1
lodw p1,,r1
push r1
call test
```

Left and Right Pushers

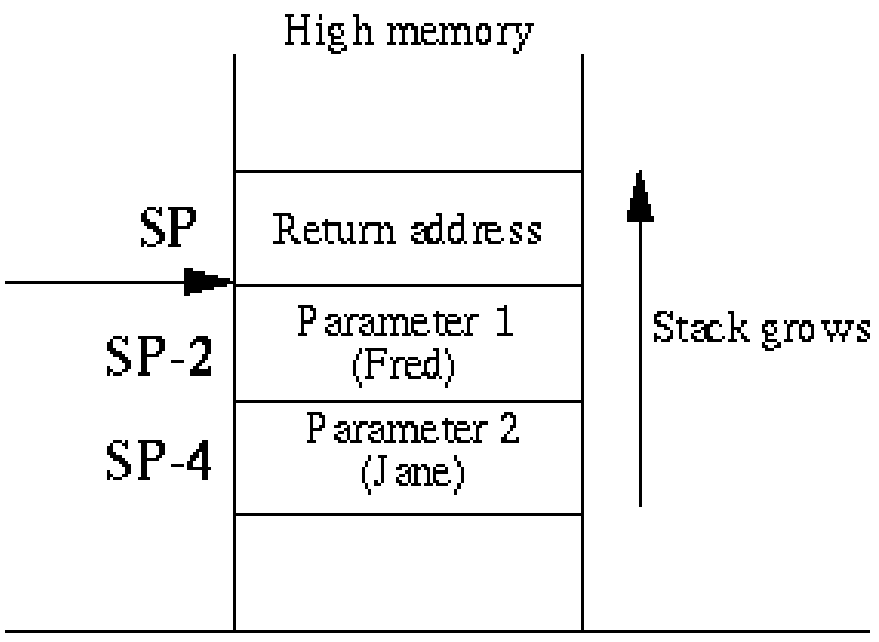
- Pascal and Modula-2 are known as *left-pushers* and C and C++ as *right-pushers*.
- These names stem from the order in which the languages push their parameters onto the stack.
  - Left-pushers push parameters from left to right
  - right-pushers do so from right to left.
- Why does it matter?
  - Obviously - this *should* be obvious - the assembler programmer writing a procedure which is to be called from a high-level language needs to know whether the language is a left- or right-pusher.
  - Why? Because the stack will look different.

Left and Right Pushers

- Consider our two-parameter (Fred and Jane) example from before. Depending on whether our HLL is a left- or right-pusher, we have two possible stack layouts:



Left-pushed stack



Right-pushed stack

Left and Right Pushers

- Obviously, our EQUates for Param1 and Param2 will have to be different in each case.
- Note, BTW, that this is, for once, not a detail forced on us by the choice of any particular CPU chip. The differences between left- and right-pushers affect **all** assembly languages on **all** CPUs.
- But is there any advantage to using left-pushing over right or vice-versa?

Left and Right Pushers - Advantages

- C and C++ are two of the few languages which allow the programmer to construct procedures (or functions) which take a *variable* number of parameters.
- Firstly, note that with a right-pushing language (which C/C++ are), the *first* parameter is the *last* to be pushed onto the stack. It is, therefore, *always* the last thing on the stack immediately before the return address. This is regardless of how many parameters there are. (With a left-pusher the first parameter's distance from the return address in the stack depends on the number of parameters). So:

  Push from right to left

  Put the parameter count (directly or indirectly) into the first parameter.
- In this way the called procedure can determine the exact parameter count by looking in a known location: namely the first parameter.

Printing Any Number of Strings

```
printer
        push    FP                  ; save caller's
        mov     SP, FP              ; frame pointer
        push    r1                  ; save all the
        push    r2                  ;    registers this
        push    r3                  ;       procedure
        push    r4                  ;         consumes
        lodw    COUNT,FP,r3         ; get string count
        mov     PUTSTR, r1          ; only need do this once
        add     FP, STRING1, r4     ; point r4 to first string pointer
        mov     r3, r3              ; force Z flag for next instruction
again
        bz      done                ; done them all
        lodw    0, r4, r2           ; point r2 to next string for sys
        sys     CONSOLE             ; and print it
        sub     r4, 2, r4           ; point r4 2 down (previous param)
        dec     r3                  ; count time through
        br      again               ; go back - try again
done
        pop     r4                  ; restore
        pop     r3                  ; all the registers
        pop     r2                  ; we previously
        pop     r1                  ; saved
        pop     FP                  ; restore FP
        ret
```

Printing Any Number of Strings

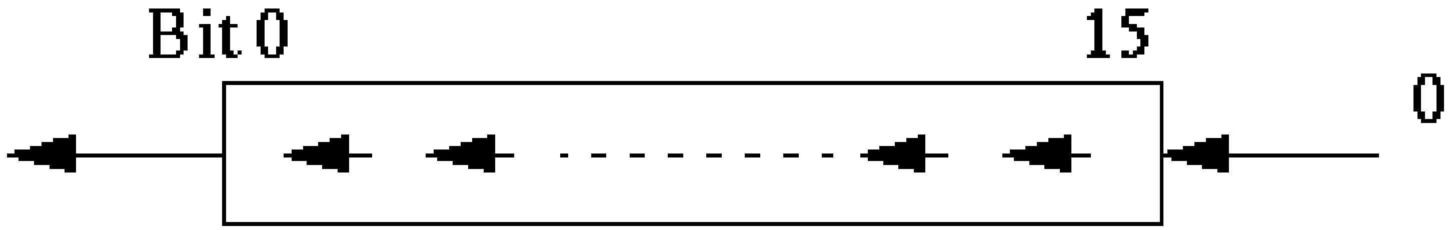
- There are several points of interest about this code, e.g. the main loop is a variant of test-at-the-top.
- Note in particular the way the loop is controlled: the ***be*** instruction tests the flags set by the ***dec*** at the bottom of the loop (the ***br*** does not affect the flags). For this reason the loop is preceded by the unusual ***mov r3, r3*** instruction which simply sets the **Z** flag off (unless the specified count is zero, in which case the code still works).

Shifts and Rotates

- **Shifts**

Name:	Shift left
Mnemonic:	SHL
Function:	src1 is shifted left src2 bits and stored in dest vacated right bits are filled with zero
Flags:	SZC set, O unchanged

- Shift instructions, as their name implies, move an entire register along by one or more bit positions.



SHift Left

- In the case of the left shift, zeros are shifted in from the right and the bits shifted out to the left are discarded. The **last** bit shifted out will be used to set the carry flag.
- Shifting left by **n** bits is equivalent to multiplying by 2^n, just as shifting left in decimal arithmetic is equivalent to multiplying by powers of 10:
- E.g. 123_{10} shifted left by 2 positions becomes 12300_{10}, which is 100 ($=10^2$) times greater.

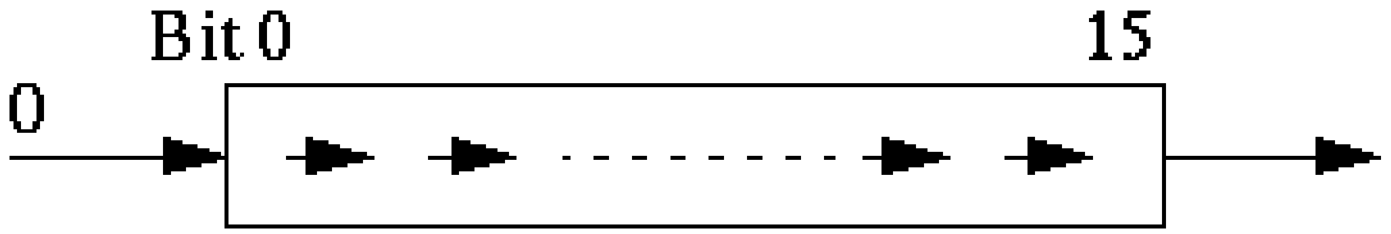
```
mov     22, r1
shl     r1, 3, r1
```

R1 before the shift contains $16_{16} = 22_{10} = 00010110_2$

After the shift, R1 contains $b0_{16} = 176_{10} = 10110000_2$ (= 22 * 8)

SHift Right

Name:	Shift right
Mnemonic:	SHR
Function:	src1 is shifted right src2 bits and stored in dest vacated left bits are filled with zero
Flags:	SZ set, C set off, O unchanged



SHift Right

- In the case of the right shift, zeros are shifted in from the left and the bits shifted out to the right are discarded.
- Shifting right by **n** bits is equivalent to dividing by 2^n, just as shifting right in decimal arithmetic is equivalent to dividing by powers of 10:
- E.g. 14768 shifted right by 2 positions becomes 147, which is 100 (=10^2) times smaller (note this is purely integer division).

```
mov     96, r1
shr     r1, 3, r1
```

R1 before the shift contains $60_{16} = 96_{10} = 01100000_2$

After the shift, R1 contains $0c_{16} = 12_{10} = 00001100_2$ (= 96 / 8)

SHift Right

- In the case of the right shift, zeros are shifted in from the left and the bits shifted out to the right are discarded.
- Shifting right by **n** bits is equivalent to dividing by 2^n, just as shifting right in decimal arithmetic is equivalent to dividing by powers of 10:
- E.g. 14768 shifted right by 2 positions becomes 147, which is 100 (=10^2) times smaller (note this is purely integer division).

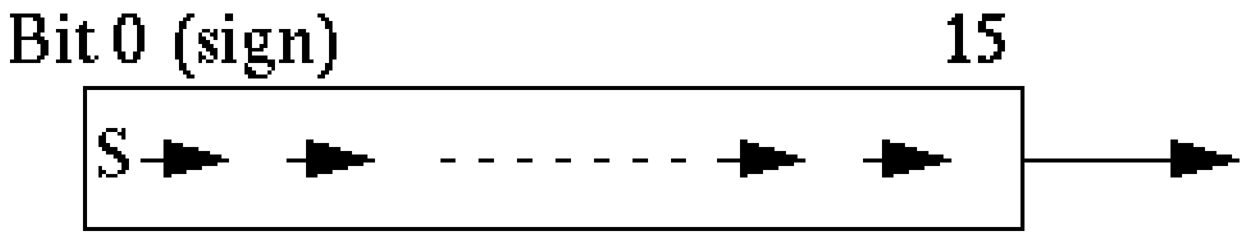
```
mov     96, r1
shr     r1, 3, r1
```

R1 before the shift contains $60_{16} = 96_{10} = 01100000_2$

After the shift, R1 contains $0c_{16} = 12_{10} = 00001100_2$ (= 96 / 8)

Shift Arithmetic Right instruction

Name:	Shift arithmetic right
Mnemonic:	SAR
Function:	src1 is shifted right src2 bits and stored in dest vacated left bits are filled with copies of src1 bit 0
Flags:	SZ set, C set off, O unchanged



Signed and Unsigned

The SHR instruction works as a powers-of-two divide for **unsigned** numbers only. If the original value is signed and is negative then SHR gives incorrect results:

```
mov     -2, r1
shr     r1, 3, r1
```

The 16-bit representation of -2 is 1111111111111110; SHRing this 1 bit gives 0111111111111111 (=32767_{10}): hardly the desired answer

The SAR instruction is provided for this; it fills the vacated bit position(s) with *copies of bit* 0: it works for positive and negative values

```
mov     -2, r1
sar     r1, 3, r1
```

The resulting value in r1 is 1111111111111111 (-1)

```
mov     96, r1
sar     r1, 3, r1
```

Still gives the correct answer - the original bit0 in R1 is 0

A Cautionary Note

- *Warning*
  - The limiting value when dividing positive integers by using shift is 0
  - When dividing negative integers it is -1.

You Have Been Warned