CS220 – Logic Design
AS04-Beginning Assembly

• Outline
  − Defining Data Elements
  − Moving Data
  − The Stack
  − Unconditional Branches
AS04-Beginning Assembly
Defining Data Elements

- The following assembler directives define basic data elements in the data section:

  - `.ascii` Text string
  - `.asciz/.string` Null-term. text string
  - `.byte` 8 bit integer
  - `.short` 16 bit integer
  - `.long/.int` 32 bit integer
  - `.quad` 64 bit integer
  - `.single` 32 bit floating point
  - `.double` 64 bit floating point
  - `.tfloat` 80 bit float
Here are some examples:

```assembly
.section .data
msg:   .asciz  "Enter a number:  
factors: .double  37.45, 45.33, 12.30
height: .int     54
pi:    .float    3.14159
bigval: .double  -45.123E+35
```

Data elements are placed in memory in the order in which they are defined.
• The **fill** directive can be used to set aside space for **arrays**:

```assembly
.section .data

# Define an 80 byte buffer of zeros
buffer:
    .fill 80

# Define an 100 element array of ints
# (4 bytes). Each int is set to 12345.
iarray:
    .fill 100, 4, 12345
```
AS04-Beginning Assembly
Defining Data Elements

- The **bss** section is used for storing data that does not have an initial value. All bytes in the **bss** section are cleared (set to 0).

- The same directives that were used in the **data** section can be used to allocate space in the **bss** section. Any initialization is ignored:

  ```
  .section .bss
  xarr: .fill 100, 4, 0
dval: .double 0
  ```
• A common mistake is to omit the expression after `.int`, `.float`, etc directives. No space is allocated in that case!

```
a: .int
b: .int 0  # a is an alias for b
```

• Space for data in the `bss` section is allocated at run-time and so is not included in the executable program while data in the `data` section is included in the program.
The `equ` directive can be used to define a symbol (named constant). The symbol is replaced by its value during assembly:

```
.equ  BUFSIZE, 100
.section.data
buffer:  .fill  BUFSIZE
.section.text
   movl  $BUFSIZE, %eax
```

It is good practice to place `equ` directives at the top of the file (outside of any section).
• The **mov** instruction copies data from a source to a destination:

\[
\text{mov}S \text{ source, destination}
\]

where \( S \) is b (8 bit), w (16 bit) or l (32 bit). \( S \) can be omitted in many cases in practice (when an operand is a register). The source can be constant (immediate) data, a register or a memory location. The destination can be a **register** or a memory location.
AS04-Beginning Assembly
Moving Data

- Moving **constant** data to registers/memory.

  # Use $ for an immediate operand.
  # C++ equivalents are in comments
  movl $0, %eax  # EAX = 0;
  movw $0xffff, %bx
  movb $0b10101010, %al
  movl $'!', %eax
  movw $-50, value  # value = -50;
  # Move ADDRESS into EDX reg
  movl $value, %edx  # EDX = &value;
• Moving data between registers

```assembly
movl %eax, %ecx  # EAX = ECX;
movw %bx, %cx    # BX = CX;

# You can not move data between
# different size registers.
movb %al, %bx     # NO, NO
# Do this instead
movw $0, %bx      # YES
movb %al, %bl     # YES
```
AS04-Beginning Assembly Moving Data

- Moving data between registers and memory

```assembly
movw %bx, count    # count = BX;
movl value, %eax   # value = EAX;
# If count labels a 16-bit
# location, do not do this!
movl %ebx, count   # NO, NO
```

# You can not move data directly
# between memory locations.
```assembly
movl xval, yval     # NO, NO
```
So far we have used only direct addressing to reference memory. In general, a memory address (reference) has the form:

\[ \text{disp}(\text{base}, \text{index}, \text{scale}) \]

where \( \text{disp} \) is an optional displacement, \( \text{base} \) and \( \text{index} \) are optional 32-bit base and index registers, and \( \text{scale} \) may be either 1 (default), 2, 4, or 8. The actual address referenced is:

\[ \text{disp} + \text{base} + \text{index} \times \text{scale} \]
Here are a few examples (assume `xarr` labels the first element in an array of ints):

```assembly
# Direct addressing
# Move 1\textsuperscript{st} element into EAX
movl xarr, %eax  # EAX = xarr[0];
```

This also moves the first element into EAX but uses \textit{indexed} addressing:

```assembly
movl $0, %edi
movl xarr(,%edi,4), %eax
```
Indexed addressing gives us access to elements other than the first. Direct addressing does not.

# Move 2nd element into EAX
incl %edi
movl xarr(,%edi,4), %eax

# Move 11th element into EAX
movl $10, %edi
movl xarr(,%edi,4), %eax
• With **indirect** addressing, a register holds the starting address:

# Move array address into EBX
movl $xarr, %ebx  # EBX=&xarr[0]

# Move 1st element into EAX
movl (%ebx), %eax # EAX=*EBX

# Also move 1st element into EAX
movl $0, %edi
movl (%ebx,%edi,4), %eax
• We can still easily access other elements:

  # Move 11\textsuperscript{th} element into EAX
  movl $10, \%edi

  # EAX = *(EBX + 10);
  movl (%ebx,\%edi,4),\%eax

• Negative displacements are permitted (as are negative index values):

  # EAX = *(EBP - 4);
  movl -4(%ebp), \%eax
AS04-Beginning Assembly
Moving Data

- **Indirect** addressing **must** be used when referencing unlabeled data. Data on the stack is unlabeled and can only be referenced through an address in the stack pointer. Routines that receive pointers (addresses) to data as arguments must also use indirect addressing to reference the data.

- All instructions that can access memory can use this general form of memory reference (not just the **mov** instruction).
The `cmov` instructions do a conditional move depending on the settings of flags in the EFLAGS register. The `cmp` instruction is often used to compare two values and set flags accordingly.

In the following example, register EAX is loaded only if EBX contains 100:

```assembly
cmp $100, %ebx
cmovzl xvalue, %eax
```
There are several instructions that can be used to **exchange** data: `xchg`, `bswap`, `xadd`, `cmpxchg`, `cmpxchg8b`.

- **xchg** exchanges the values of two registers, or a register and a memory location:
  
  - `xchgb %al, %bl`
  - `xchgl %eax, -4(%ebp)`
  - `xchgw val, %bx`
AS04-Beginning Assembly
Exchanging Data

- **bswap** exchanges the *endian* order of bytes in a register. It is useful when translating binary files from a big endian machine to little endian format.

- **xadd** exchanges two values, adds the values and stores the sum in the destination.

- **cmpxchg** compares the destination with the EAX/AX/AL register. If equal, the source is moved to the destination, if not it is moved into the EAX/AX/AL register.
The stack is an area of memory that is automatically reserved for your use when storing data. (It is also holds arguments and the return address in a function call.)

The stack pointer register (ESP) contains the address of the last item stored on the stack.

The stack “grows” downward, that is, the address in the stack pointer decreases as items are added to the stack.
The programmer must remember the size and location of items on the stack!

Here are some examples that illustrate how to add items to the stack:

```
subl $4, %esp     # Dec first!!
movl val, (%esp)  # val -> stack
subl $4, %esp
movl $val, (%esp) # &val -> stack
subl $2, %esp
movw %bx, (%esp)  # BX -> stack
```
AS04-Beginning Assembly

The Stack

- You would remove the items like this:

```assembly
movw (%esp), %bx  # Move first!!
addl $2, %esp     # then increment
movl (%esp), %eax
addl $4, %esp
movl (%esp), val
addl $4, %esp
```

- It is not necessary to *pop* data back into the same register or memory location that was *pushed* onto the stack.
The push instruction simplifies adding items to the stack. ESP is automatically decremented before the move:

```
pushl val        # val -> stack
pushl $val       # &val -> stack
pushw %bx        # BX -> stack
```

Similarly, data is easily removed with pop. ESP is incremented after the move:

```
popl %eax        # stack -> EAX
popl val         # stack -> val
popw %bx         # stack -> BX
```
AS04-Beginning Assembly
The Stack

- Only 16-bit and 32-bit pushes and pops are allowed. (There is no `pushb` or `popb`.)
- `pusha` (or `pushal`) pushes all 8 general purpose registers on to the stack. `popa` (or `popal`) will restore the registers from the stack.
- These instructions are often used near the start and end of functions (see `skeleton.s`).
• Labels can also be used to reference addresses within the **text** section. The **jmp** instruction can then be used to load the instruction pointer (EIP) with a new address and change the normal flow of execution:

```plaintext
loc01: movl $100, %eax
       jmp loc03
loc02: movl %eax, %ebx
       jmp loc01
loc03: addl %eax, %ebx
```
AS04-Beginning Assembly
Unconditional Branches

• A **call** instruction also changes the standard flow of execution. The **call** instruction pushes the EIP (which contains the address of the next instruction after **call**) onto the stack and jumps to the operand address.

• A **ret** instruction pops an address off of the stack into EIP. The next instruction to be executed will be the instruction at the address popped from the stack.
Here is a simple example illustrating the use of the `call` and `ret` instructions:

```assembly
_asm_main:
    call foobar
    ret

foobar:
    movl $1200, %eax
    call print_int
    ret
```
interrupts are another type of unconditional branch. Hardware interrupts are generated by hardware external to the processor. They can (typically) occur at any time.

An interrupt allows the current instruction to complete. The EIP, CS, and EFLAGS registers are pushed on the stack and the EIP is loaded with an address from the interrupt vector table (IVT).
AS04-Beginning Assembly
Unconditional Branches

- The IVT contains addresses of system and interrupt service routines (ISR).

- The `int` instruction generates a software interrupt. The operand is an index into the IVT. The EIP, CS, and EFLAGS registers are pushed onto the stack and the EIP is loaded with the address from the IVT.

- The `iret` instruction is used at the end of the system routine or ISR to restore the EIP, CS, and EFLAGS registers from the stack.