Issues in Final Code Generation

- Final code generation is similar to intermediate code generation in some ways, but there are several issues that arise that do not occur in intermediate code generation:
  - Instruction Set
  - Memory Allocation
  - Register Allocation
  - Operating System Calls
Target Architecture

• Our target architecture is the Intel 8x86 family of processors.
• We first must consider:
  – Register Set
  – Flags
  – Floating Point Unit

16-bit Processor Architecture
General Purpose Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Bits</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>AX</td>
<td>0-15</td>
<td>AH, AL</td>
</tr>
<tr>
<td>BX</td>
<td>0-15</td>
<td>BH, BL</td>
</tr>
<tr>
<td>CX</td>
<td>0-15</td>
<td>CH, CL</td>
</tr>
<tr>
<td>DX</td>
<td>0-15</td>
<td>DH, DL</td>
</tr>
</tbody>
</table>

AX (Accumulator) - favored for arithmetic operations
BX (Base) - Holds base address for procedures and variables
CX (Counter) - Used as a counter for looping operations
DX (Data) - Used in multiplication and division operations.
Segment Registers

Segment registers are used to hold base addresses for program code, data and the stack.

- **CS (Code Segment)** - holds the base address for all executable instructions in the program
- **SS (Stack Segment)** - holds the base address for the stack
- **DS (Data Segment)** - holds the base address for variables
- **ES (Extra Segment)** - an additional base address value for variable.

Index Registers

Index Registers contain the offsets for data and instructions.

- **Offset** - distance (in bytes) from the base address of the segment.

- **BP (Base Pointer)** - contains an assumed offset from the SS register; used to locate variables passed between procedures.
- **SP (Stack Pointer)** - contains the offset for the top of the stack.
- **SI (Source Index)** - Points to the source string in string move instructions.
- **DI (Destination Index)** - Points to the source destination in string move instructions.
Status and Control Registers

IP (Instruction Pointer) - contains the offset of the next instruction to be executed within the current code segment.

Flags register contain individual bits which indicate CPU status or arithmetic results. They are usually set by specific instructions.

- **O** = Overflow
- **D** = Direction
- **I** = Interrupt
- **T** = Trap
- **x** = undefined
- **S** = Sign
- **Z** = Zero
- **A** = Auxiliary Carry
- **P** = Parity
- **C** = Carry

Flags

There are two types of flags: control flags (which determine how instructions are carried out) and status flags (which report on the results of operations).

- Control flags include:
  - **Direction** Flag (DF) - affects the direction of block data transfers (like long character string). 1 = up; 0 - down.
  - **Interrupt** Flag (IF) - determines whether interrupts can occur (whether hardware devices like the keyboard, disk drives, and system clock can get the CPU’s attention to get their needs attended to.
  - **Trap** Flag (TF) - determines whether the CPU is halted after every instruction. Used for debugging purposes.
Status Flags

- Status Flags include:
  - **Carry** Flag (CF) - set when the result of unsigned arithmetic is too large to fit in the destination. 1 = carry; 0 = no carry.
  - **Overflow** Flag (OF) - set when the result of signed arithmetic is too large to fit in the destination. 1 = overflow; 0 = no overflow.
  - **Sign** Flag (SF) - set when an arithmetic or logical operation generates a negative result. 1 = negative; 0 = positive.
  - **Zero** Flag (ZF) - set when an arithmetic or logical operation generates a result of zero. Used primarily in jump and loop operations. 1 = zero; 0 = not zero.
  - **Auxiliary Carry** Flag - set when an operation causes a carry from bit 3 to 4 or borrow (from bit 4 to 3). 1 = carry, 0 = no carry.
  - **Parity** - used to verify memory integrity. Even # of 1s = Even parity; Odd # of 1s = Odd Parity

Floating-Point Unit

<table>
<thead>
<tr>
<th>80-bit Data Registers</th>
<th>48-bit Pointer Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>ST(0)</td>
<td>FPU Instruction Pointer</td>
</tr>
<tr>
<td>ST(1)</td>
<td>FPU Data Pointer</td>
</tr>
<tr>
<td>ST(2)</td>
<td></td>
</tr>
<tr>
<td>ST(3)</td>
<td></td>
</tr>
<tr>
<td>ST(4)</td>
<td></td>
</tr>
<tr>
<td>ST(5)</td>
<td></td>
</tr>
<tr>
<td>ST(6)</td>
<td></td>
</tr>
<tr>
<td>ST(7)</td>
<td></td>
</tr>
<tr>
<td>Opcode Register</td>
<td></td>
</tr>
</tbody>
</table>

Tag Register
Control Register
Status Register
Tag Register

<table>
<thead>
<tr>
<th>tag</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>valid (finite nonzero number)</td>
</tr>
<tr>
<td>01</td>
<td>zero</td>
</tr>
<tr>
<td>10</td>
<td>invalid (infinite or NaN)</td>
</tr>
<tr>
<td>11</td>
<td>empty</td>
</tr>
</tbody>
</table>

Control Register

- The control register contains six exception masks and three control fields.
- If one of the exception masks is cleared and that exception occurs, the program is suspended and the an interrupt is generated, which will either correct the problem is terminate the program.
- The control fields control rounding and the type of infinity used.
Memory Layout

- Data Segment
- Code Segment
- Stack Segment

Addresses growing

DS
CS
SS

Coding The Stack Segment

DOSSEG  segment name  paragraph alignment

STACK SEGMENT  para stack 'stack'  uninitialized
dw  1000 dup(?)

STACK ENDS
define word  # of words allocated
Coding The Data Segment

_DATA SEGMENT word public 'data'
TestResult dw ?
x       dw ?
y       dw ?
t47     dw ?
t48     dw ?
t49     dw ?
t51     dw ?
t55     dw ?
t56     dw ?
_DATA ENDS

Generating the Stack Segment Code

void writestack(void)
{
    fprintf(ofp, "%s
%s
%s
%s

",
        "DOSSEG",
        "_STACK SEGMENT para stack \'stack\'",
        "dw 1000 dup(?)",
        "_STACK ENDS");
}

Generating the Data Segment Code

```c
void writedata(void)
{
    int i, datasize;
    float litvalue;
    char label[LABELSIZE];

    fprintf(ofp, "_DATA SEGMENT word public"
            "\'data\'\n");
    fprintf(ofp, "TestResult dw ?\n");

    for (i = NUMTOKENS+2; i < tablesize(); i++) {
        if ((symclass(i) == sttempvar ||
             symclass(i) == stvariable)
            && getproc(i) == NUMTOKENS+1) {
            getlabel(i, label);
            if (data_class(i) == dtinteger)
                fprintf(ofp, "%-10s     dw     ?\n", label);
            else
                fprintf(ofp, "%-10s     dd     ?\n", label);
        }
    }
}
```
else if (symclass(i) == stliteral
     && data_class(i) == dtreal) {
    getlabel(i, label);
    litvalue = getrvalue(i);
    fprintf(ofp, "%-10s dd %f
", 
            label, litvalue);
}

fprintf(ofp, "_DATA ENDS\n\n");

### Allocating the Stack

![Stack Diagram]

- **Addresses growing**
- **Bottom of stack**
- **Top of stack**
- **SS**
- **SP**
Activation Record

Addresses growing

Activation Record

Local Variables

Return Address

Parameters

BP

SP

Processing Assignments

; x := 33
    mov    ax, 33
    mov    x, ax

; x := y
    mov    ax, y
    mov    x, ax
Processing Integer Addition

; $5 := 3 + 4
    mov    ax, _t47
    add    ax, _t48
    jno    Jump3
    jmp    iovrflo

Jump3:
    mov    _t49, ax

Processing Integer Multiplication

; $4 := x * y
    mov    ax, x
    imul   y
    jno    Jump2
    jmp    iovrflo

Jump2:
    mov    _t50, ax
Processing Integer Division

; $3 := y / b

    cmp  b, 0
    jne  Jump0
    jmp  divby0

Jump0:

    mov  ax, y
    cwd
    idiv  b
    jno  Jump1
    jmp  iovrflo

Jump1:

    mov  _t51, ax

Processing Jumps

; if $6 != 0 goto _loop55

    cmp  _t51, 0
    je   Jump6
    jmp  _loop55

... ... ...

Jump6:

; goto _loop54
    jmp  _loop54
Processing Procedure Calls

; arg x
    mov ax, offset x
    push ax

; call test
    call test

Beginning the Procedure

- Beginning a new procedure requires:
  - Saving the base pointer (where the current activation record begins)
  - Setting the new stack pointer to the old base pointer (where the new activation record begins)
  - Allocating space on the stack (in the new activation record) for local variables by adjusting the stack pointer.
Pushing Parameters on the Stack

Pushing the Return Address on the Stack
Dynamic Allocation of Local Variables

Code For the Procedure’s Beginning

```asm
_TEXT SEGMENT
test:
push bp
mov bp, sp

; Allocate space for local variables
sub sp, 12
```
Local Variables In Assembler

; a := c
    mov bx, word ptr [bp+2]
    mov ax, [bx]
    mov word ptr [bp-2], ax
; b := 8
    mov ax, 8
    mov word ptr [bp-4], ax
; $0 := a + b
    mov ax, word ptr [bp-2]
    add ax, word ptr [bp-4]
    jno Jump9
    jmp iovrflo
Jump9:
    mov word ptr [bp-6], ax

Ending the Procedure

; Return space used by local variables
    mov sp, bp
    pop bp
    ret 2

_TEXT ENDS