Outline

Conditional Branches
High Level Decision/Looping Constructs
Basic Mathematical Operations
Unlike the unconditional branch instructions (jmp, call/ret, and int) conditional branches are not always taken. Whether a branch is taken depends on the state of flags in the EFLAGS register.

The five status flags are: carry flag (CF – bit 0), parity flag (PF – bit 2), zero flag (ZF – bit 6), sign flag (SF – bit 7), and the overflow flag (OF – bit 11).
The status flags are affected by several of the instructions (add, sub, mul, div, etc). The cmp (compare) instruction is most often used with conditional branches.

The cmp instruction compares two integers by subtracting its first operand from its second one. Bits in the EFLAGS register are set based on the result, but the result is not stored.
Here are the **conditional** branch instructions that use the EFLAGS flags individually:

- **JZ**  branch if ZF is set (1)
- **JNZ** branch if ZF is clear (0)
- **JO**  branch if OF is set
- **JNO** branch if OF is clear
- **JS**  branch if SF is set
- **JNS** branch if SF is clear
- **JC**  branch if CF is set
- **JNC** branch if CF is clear
- **JP**  branch if PF (parity flag) is set
- **JNP** branch if PF is clear
The previous branch instructions are useful, but we often want to compare two numbers. This requires examination of multiple flags and differs for signed and unsigned numbers. Fortunately, additional branch instructions are provided.

When comparing unsigned integers, only the zero (ZF) and carry (CF) flags are important. ZF is set if the numbers are equal (the result is zero). CF is set if a borrow is required.
Consider the instruction

cmp ufirst, usecond

where ufirst and usecond are unsigned. Flags are set based on the result of (usecond - ufirst):

ZF = 1, CF = 0  If usecond = ufirst
ZF = 0, CF = 0  If usecond > ufirst
ZF = 0, CF = 1  If usecond < ufirst
When the `cmp` operands are unsigned we can use the following `branch` instructions:

- **JE**  branch if `usecond = ufirst`
- **JNE** branch if `usecond ≠ ufirst`
- **JB, JNAE** branch if `usecond < ufirst`
- **JBE, JNA** branch if `usecond ≤ ufirst`
- **JA, JNBE** branch if `usecond > ufirst`
- **JAE, JNB** branch if `usecond ≥ ufirst`

(Read B as below, A as above, E as equal, N as not. `usecond` is compared to `ufirst`.)
When comparing two signed integers three flags are important: the zero (ZF), overflow (OF) and sign (SF) flags. (Recall that the carry flag (CF) is ignored.)

The zero flag is set if the two numbers are equal. The overflow flag is set if an overflow occurs. For example, if the operands are byte-sized, (-120) - (100) = (-220) would overflow. The sign flag is set if the result is negative.
Consider the instruction

```assembly
cmp sfirst, ssecond
```

where `sfirst` and `ssecond` are signed. Flags are set based on `(ssecond - sfirst)`:  

- `ZF = 1` If `ssecond = sfirst`
- `ZF = 0, SF = OF` If `ssecond > sfirst`
- `ZF = 0, SF ≠ OF` If `ssecond < sfirst`
If the `cmp` operands are signed we can use the following branch instructions:

- **JE**: branch if `ssecond = sfist`
- **JNE**: branch if `ssecond ≠ sfist`
- **JL, JNGE**: branch if `ssecond < sfist`
- **JLE, JNG**: branch if `ssecond ≤ sfist`
- **JG, JNLE**: branch if `ssecond > sfist`
- **JGE, JNL**: branch if `ssecond ≥ sfist`

(Read L as less, G as greater, E as equal, N as not. `ssecond` is compared to `sfist`.)
Here is a simple example:

```
cmpl %eax, %ebx
jge  BIGEBX
# Code for EBX < EAX here
jmp  NEXTAA
BIGEBX:
# Code for EBX >= EAX here
NEXTAA:
```

Note: The programmer determines whether numbers are signed or unsigned.
The following pseudo-code:

```
if (EAX == 8) EBX = 1;
else          EBX = 2;
```

can be coded in assembly as:

```
cmpl $8, %eax  # Compare EAX to 8
je   thenblock # jump if EAX == 8
movl $2, %ebx  # ELSE part of IF
jmp   next     # jump over THEN
thenblock:
   movl $1, %ebx  # THEN part of IF
next:
```
Alternatively, it can be coded as:

```assembly
    cmpl $8, %eax  # Compare EAX to 8
    jne  elseblock;  jump if EAX != 8
    movl $1, %ebx  ; THEN part of IF
    jmp  next    ; jump over ELSE

elseblock:
    movl $2, %ebx  ; ELSE part of IF

next:
```

The first form preserves the comparison in the pseudo-code while the second form preserves the order of the then/else blocks.
Loops

The loop instructions (there are five) can be used to implement for-like loops. Each instruction has a label (address) as an operand. The loop instructions decrement ECX and compare the new value with 0.

- LOOP --ECX, branch if ECX≠0
- LOOPE --ECX, branch if ECX≠0, ZF=1
- LOOPZ An alias for LOOPE
- LOOPNE --ECX, branch if ECX≠0, ZF=0
- LOOPNZ An alias for LOOPNE
AS05-Intermediate Assembly
Loops

We can use **loop** to translate the following *for*-loop pseudo-code:

```plaintext
sum = 0;
for (i=10; i>0; i--)
    sum += i;
```

to assembly as:

```asm
movl $0, %eax ; Use EAX as sum
movl $10,%ecx ; Set loop count
loop_start99:
    addl %ecx, %eax
loop loop_start99
```
loope and loopne are useful when searching for a value in an array:

```assembly
  # Look for element containing # 1234 in 50 element int array
  movl  $49, %ecx
  movl  $xarr, %eax
  begin01:
  cmpl  $1234, (%eax,%ecx,4)
  loopne begin01
```
A C++ **while** loop can be implemented in assembly like this:

```
movl $0, %eax  # int cnt = 0;
begwhile:     # while (cnt<50) {
cmpl $50, %eax
jge endwhile
    // WHILE BODY HERE
incl %eax    #   cnt++;
jmp begwhile # }
endwhile:
```
Instead of using the **loop** instructions for **for** loops it is often easier to translate the **for** loop to a **while** loop and implement the **while** loop in assembly as on the previous slide:

```
for(i=0; i<25; i++)
    { y = y+i; }
```

would become

```
i=0;
while (i<25)
    {y = y+i; i++;}
```
The CPU has no idea what a particular byte (or word or long) is supposed to represent. Assembly does not have the idea of types that a high level language has.

You (the programmer) must keep track of whether an integer is signed (two's complement) or unsigned. You must also keep track of the size of each integer (byte, word, or long).
For example, if the word `0xFFFF` is meant to be unsigned it represents 65535, if it is supposed to be signed it represents -1.

For many operations there are different assembly instructions that are used for signed and unsigned integers. However, addition and subtraction are identical operations for both signed and unsigned numbers and use the same instructions.
Here are a few examples of integer addition and subtraction:

```
addl $5, %eax           # EAX = EAX + 5
addb %ah, bval          # BVAL=BVAL+AH
addw wval,%bx           # BX=BX+WVAL
subw $-10, %bx          # BX = BX - -10
subl (%eax),%ebx         # EBX=EBX-*EAX
incl %ecx               # ECX++
incl (%ecx,%edi,4)       # DL--
```
Use **mulS** to do unsigned int multiplication and **imulS** for signed integer multiplication.

Two different instructions are needed because unsigned multiplication is different from signed multiplication. (0xFF)*(0xFF) unsigned (byte) is FE01 (+65025) while (0xFF)*(0xFF) signed (byte) is 0001 (+1).
\textbf{mulS} takes only a single operand, either a register or memory reference. The other operand and destination are \textit{implicit}.

An 8-bit operand is multiplied by AL and the product stored in AX.

A 16-bit operand is multiplied by AX and the product stored in (DX:AX).

A 32-bit operand is multiplied by EAX and the product stored in (EDX:EAX).
Here are some `mul` examples:

```
mulb %al            # AX = AL*AL
mulw xval           # DX:AX = AX*xval
```

`imul` has one, two and three operand forms. Here are some examples:

```
imulb %al           # AX = AL*AL
imulw xval          # DX:AX = AX*xval
imulw %dx, %ax      # AX = AX*DX
imull $100,num,%eax # EAX = 100*NUM
```
**divS** and **idivS** are used similarly for unsigned and signed division. Each take only a single operand (the divisor).

If the operand, Q, is 8-bit, then AL = AX/Q and AH = AX%Q (remainder).

If the operand is 16-bit, then AX = (DX:AX)/Q and DX = (DX:AX)%Q.

If the operand is 32-bit, then EAX = (EDX:EAX)/Q and EDX = (EDX:EAX)%Q.
There are no special multiple-operand versions of `idiv` as for `imul`. Here are some examples:

```
divb  %bl       # AL=AX/BL, AH = AX%BL
divw  x         # AX=(DX:AX)/x
              # DX=(DX:AX)%x
idivl  %eax     # EAX=(EDX:EAX)/EAX
              # EDX=(EDX:EAX)%EAX
```
To increase the size of an unsigned number we just add zeros. For example, the byte value FF (255 unsigned) should become the word 00FF (255 still).

To increase the size of a signed number we must extend the sign bit. The byte FF (-1) should become FFFF (-1) or FFFFFFFF (-1) when increased to word or double word size. The byte 5A (+90) should become 005A (+90) or 0000005A (+90 still).
To increase the size of unsigned data we just need to zero out the upper bits:

```
movb $0,%ah  #extend AL to AX
```

We can't extend AX to EAX this way because there is no way to refer to the upper 16 bits of EAX. The `movzss` instructions are intended for just this problem:

```
movzbw %al,%ax  #extend AL to AX
movzwl %ax,%eax #extend AX to EAX
movzwl %ax,%ebx #extend AX to EBX
```
The `movsSS` instructions are intended for extending the sign bit in signed numbers:

- `movsbw %al,%ax` #extend AL to AX
- `movswl %ax,%eax` #extend AX to EAX
- `movswl %ax,%ebx` #extend AX to EBX

There are also some special sign extension instructions. `cbw` (convert byte to word) sign extends AL to AX. `cwde` sign extends AX to EAX. `cdq` even sign extends EAX into EDX:EAX (64 bits).
The **negS** (negate) instruction computes the two's complement of its operand in-place. The operand may be either a register or memory location.

The **addS** and **subS** instructions set the carry flag in the EFLAGS register if a carry or borrow is generated. The **adc** (add with carry) and **sbb** (subtract with borrow) can then be used to add or subtract larger integers (64, or 128 or ??? bit integers).