TDDI11: Embedded Software

Min Bao

Outline

- Part 1
  - Lab organization
  - Brief introduction of the targeted problems
  - Tools
  - Assignments
- Part 2
  - X86 Review

Labs Organization

- 2 lab groups (divided in teams of 2 students)
- 24 hours/lab group (supervised), 12 lab sessions
  - 31/3 17-21: Only group A (GRA)
  - 2/4 17-21: Only group B (GRB)
  - Other times: Both GRA and GRB
- 5 lab assignments
- 2 points (3 ECTS points)

- http://www.ida.liu.se/~TDDI11
- Prepare before coming to the lab !!!
- Follow the instructions EXACTLLY !!!
- You have to show the demo and code in the lab !!!

Labs Organization(con't)

- Register in webreg for the labs
  - http://www.ida.liu.se/webreg
- Demo and Codes of 5 assignments have to be shown in 12 lab sessions (no other time!!)

Goals

- Understand data representation at machine level
- Master operations most frequently used in embedded systems
- Understand why and when programming in assembly is necessary/appropriate
- I/O programming
- Preemptive/Non-preemptive multithreaded programming

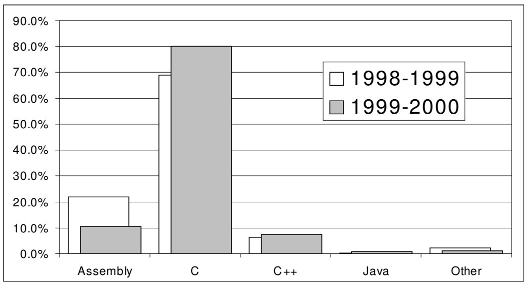
Embedded Software

- Embedded systems generally serve a single/specific purpose
- In our labs, the embedded software consists of one single program image that contains
  - The application software
  - A small real time kernel (labs 4-5)

Hardware Architecture: Intel x86

- Dominant architecture for PCs
- No need for specialized single board computers => cheap development platform
- Studied concepts are to a certain extent independent of the architecture
- Protected mode of Intel 386 is quite representative for modern architecture
- Easier transition from programming for general purpose systems to embedded software development

Programming Languages



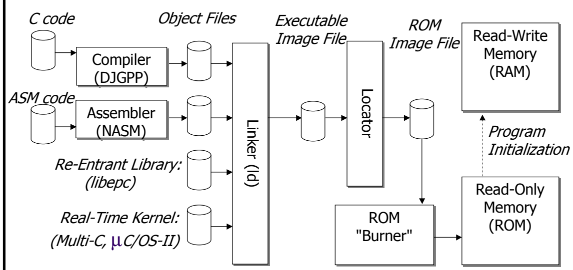
Embedded Software Tool Set

Tool
- Eclipse (programming IDE)
- DJGPP (Windows port of GNU C compiler)
- NASM (assembler)
- Simics (hardware simulator)

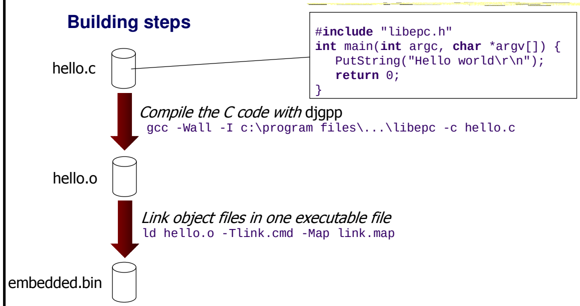
Library
- Libepc

Real-time kernel
- Multi-C (non-preemptive real-time kernel)
- µC/OS-II (preemptive real-time kernel)

Embedded Software Tool Set (cont'd)

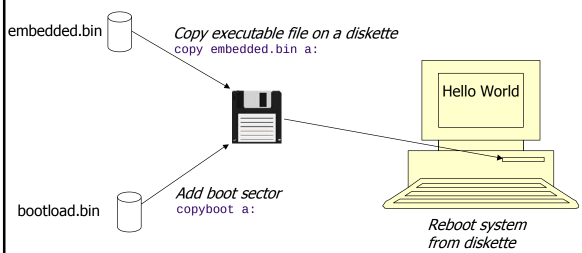


Hello World example



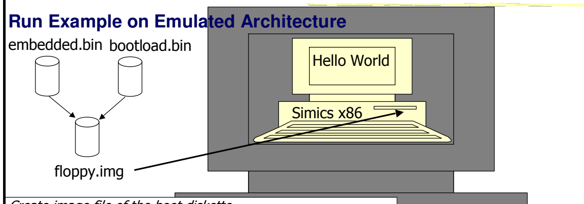
Running Hello World (1)

Run Example on Target (Non-Emulated) Architecture



Running Hello World (2)

Run Example on Emulated Architecture



Create image file of the boot diskette
```
copydisk -b bootload.bin -o floppy.img embedded.bin
```
Run Simics to emulate x86 architecture (bootable from diskette)
```
simics targets/x86-440bx/dredd-floppy-install.simics
```
Boot the emulated architecture from diskette image file
```
simics> flp0.insert-floppy A floppy.img
```


Assignment 1: Measuring processor speed

Goal: Write a program that computes and displays the clock rate of an Intel x86 CPU

Solution: Compute the number of CPU clock cycles that pass during a given time interval

- Use libepc library functions
  - `Now_Plus:` used for measuring time
  - `CPU_Clock_Cycles:` used for measuring cycles

Assignment 1: Measuring processor speed

- `Now_Plus`
  - **`Now_Plus(0)`** returns a 32-bit integer.
  - **`Now_Plus(int n)`** returns the 32-bit integer that **`Now_Plus(0)`** would return if called **n** seconds from the moment when **`Now_Plus(int n)`** is called.
- `CPU_Clock_Cycles`
  - Returns the value stored in a 64-bit counter that is incremented at the processor clock rate

ClockRate = $(\text{CPU\_Clock\_Cycles}_n - \text{CPU\_Clock\_Cycles}_0) / \text{time}_n$

Assignment 2: Mixing C and Assembly

Goals:
- Understand data representation at machine level
- Understand bit manipulation
- Learn to use C and assembly in the same program.
- Become aware of performance issues.

Assignment 2: Mixing C and Assembly

Problem:
- Multiplication of 64-bit values on a 32-bit architecture

Solution:
- Software emulation

```
void llmultiply(unsigned long long int A, unsigned long long int B,
                unsigned char* AmulB);
```

Requirements:
- ASM implementation
- C implementation, with compiler optimization
- C implementation, without compiler optimization

Assignment 2: 64-bit Multiplication

```
void llmultiply(
        unsigned long long int A,
        unsigned long long int B,
        unsigned char* AmulB) {
```

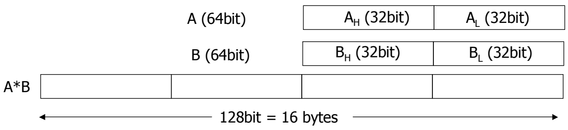
$A = A_H * 2^{32} + A_L$

$B = B_H * 2^{32} + B_L$

return $A * B = A_H * B_H * 2^{64} + (A_H * B_L + A_L * B_H) * 2^{32} + A_L * B_L$

```
}
```

Assignment 2: 64-bit Multiplication



$A * B = A_H * B_H * 2^{64} + (A_H * B_L + A_L * B_H) * 2^{32} + A_L * B_L$

- implemented using operations on 32 bits
  - Four integer multiplications (MUL)
  - register shift (e.g. SHL, SAR)
  - integer additions (ADD, ADC)

Assignment 2: 64-bit Multiplication

Compiling the C code with optimisation

```
gcc -Wall -Ic:\program files\...\libepc -O3 -c llmultiply.c
```

Compiling the C code without optimisation

```
gcc -Wall -Ic:\program files\...\libepc -O0 -c llmultiply.c
```

Makefile.common

```
CFLAGS   = -Wall -O0  $(INCLUDES)
CFLAGS   = -Wall -O3  $(INCLUDES)
```


Assignment 3: I/O Programming

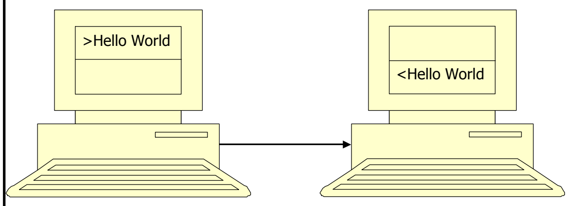
Goal:

- Become aware of the various ways to communicate with the peripherals. Understand the advantages and disadvantages of each.

Problem: write a program that

- reads characters typed on the keyboard and sends the characters on the serial interface.
- displays characters read remotely from the serial interface

Assignment 3: I/O Programming



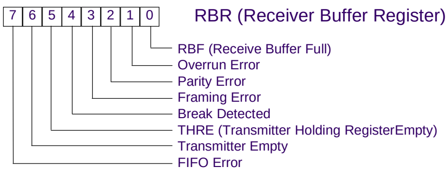
WRITE on serial interface: **Polling** READ from serial interface: **Interrupt**

Assignment 3: I/O Programming

Polling:

- continuously checks the peripheral to detect if it has changed state.

LSR (Line Status Register) THR (Transmitter Holding Register) RBR (Receiver Buffer Register)



Assignment 3: I/O Programming

Interrupt Driven Communication:

- Instead of the processor interrogating the peripheral, the peripheral notifies the processor if its state has changed.
  - The peripheral asserts a signal that interrupts the execution of the processor.
  - The execution jumps to a predefined address where the *Interrupt Service Routine (ISR)* resides. The ISR implements the response to the interrupt.

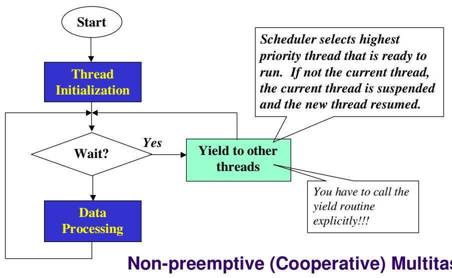
Advantage: the processor works with the peripheral only when needed.

Assignment 4: Non-preemptive Multithreaded Application

Goals:
- Work with multi-threaded programs.
- Understand non preemption

Problem: Split the implementation of lab 3 into 3 threads
- Read keyboard, print input, send input on serial
- Read serial, print received data
- Display local timestamps

Assignment 4: Non-preemptive Multithreaded Application



Non-preemptive (Cooperative) Multitasking

Assignment 4: Non-preemptive Multithreaded Application

Multi-C: real-time non-preemptive kernel

Main
- MtCCoroutine(Thread_1());
- MtCCoroutine(Thread_2());
- MtCCoroutine(Thread_3());

Thread_1
- Read keyboard
- print input
- send input on serial

Thread_2
- Read serial line
- print received data

Thread_3
- Display timestamps

Assignment 5: Preemptive Multithreaded Application

Goal:
- Work with pre-emptive kernels.
- Understand the critical section concept.

Problem:
- Extend the problem in lab 4 to send local timestamps over the serial(in addition to sending characters)
- Allow preemption
- Fix packet corruption due to preemption
- Count and display how many characters has been received by remote machine

Assignment 5: Preemptive Multithreaded Application

Problem:
- Send both chat text and timestamps over the serial link

Solution:
- Create packets out of individual bytes

```
void SendPacket(int type, BYTE8 *bfr, int bytes);
```

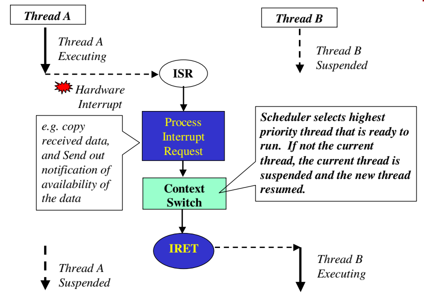
	Start Flag	Type	Byte Count	Data Bytes
Chat packet	0xff	0x01	n	$Byte_1...Byte_n$
Time packet	0xff	0x02	n	$Byte_1...Byte_n$

Assignment 5: Preemptive Multithreaded Application

Problem:
- Allow Preemption

Solution:
- Use µC/OS-II to implement the application
  - Event-driven application
  - Threads stay suspended until they are activated by an external event that triggers an interrupt
  - Thread communication and synchronisation (semaphores, queues, mutexes, mailboxes)

Assignment 5: Preemptive Multithreaded Application



Preemptive Multitasking

Assignment 5: Preemptive Multithreaded Application

Problem:

- Fix packet corruption due to preemption
  - keyboard interrupts can initiate transmission of a chat packet in the middle of transmitting a time packet

Solution:

- mark the SendPacket as a critical section

OS_EVENT *OSSemCreate(int count); allocates and initializes a semaphore data structure and returns a pointer to it. The parameter "count" is set to 1 for a Mutex.

OSSemPend(OS_EVENT *semaphore, int timeout, BYTE8 *err); returns when it acquires the semaphore; if the semaphore is currently owned by another thread, this function causes the current thread to be suspended while it waits for the semaphore to be released.

OSSemPost(OS_EVENT *semaphore); causes the current thread to relinquish ownership of the semaphore.

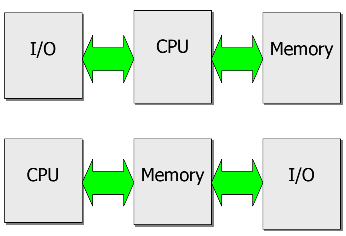
Part 2 X86 Review

- A Programmer's View of Computer Organization
- X86 Processor architecture
- Intel X86 assembly
  - Addressing Modes
  - Basic assembly
  - Mixed with C

Part 2 X86 Review

- A Programmer's View of Computer Organization
- X86 Processor architecture
- Intel X86 assembly
  - Addressing Modes
  - Basic assembly
  - Mixed with C

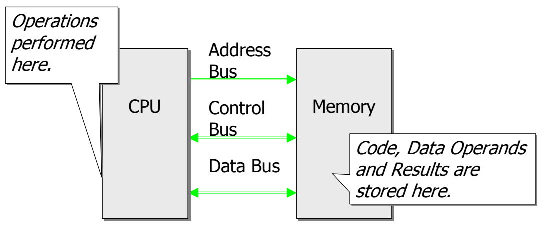
Input/Output Configurations



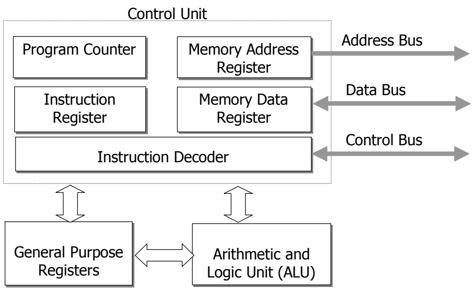
CPU coordinates transfer between I/O and memory.

Direct Memory Access (DMA).

CPU and Main Memory



The Central Processing Unit



Part 2 X86 Review

- A Programmer's View of Computer Organization
- X86 Processor architecture
- Intel X86 assembly
  - Addressing Modes
  - Basic assembly
  - Mixed with C

History of Intel x86 Architecture

Processor	Year	MIPS	CPU Frequency	Register Size	Data Bus	Address Space	CPU Cache
8086	1978	0.8	8.0 MHz	16	16	1 MB	None
286	1982	2.7	12.5 MHz	16	16	16 MB	None
386	1985	6.0	20 MHz	32	32	4 GB	None
486	1989	20	25 MHz	32	32	4 GB	8 KB L1
Pentium	1993	100	60 MHz	32	64	4 GB	16 KB L1
Pentium Pro	1995	440	200 MHz	32	64	64 GB	16 KB L1; 512 KB L2
Pentium II	1997	466	266	32	64	64 GB	32 KB L1; 512 KB L2
Pentium III	1999	1000	500	32	64	64 GB	32 KB L1; 512 KB L2

Images of x86 chips: `http://www.cpu-collection.de`

Operating Modes of Intel IA

- **Real-address Mode**

  Equals 8086 processor, the initial operating mode at start-up, limited feasture,1MB memory addressable.
- **Protected Mode**

  This mode was originally introduced with the Intel 286, and later enhanced in the Intel 386. Protected mode offers greater performance than real mode. All of the features of the processor are available and a much larger physical address space.
- **System Management Mode**

  Introduced for 386SL, Implement power management and system security (not deal with it here)

Instruction Format

CISC (Complex Instruction Set Computer)

Operand Fields	*Example*	*Description*
0	CLC	Clear the carry flag to 0.
1	INC AX	Increment contents of register AX
2	MOV AX,BX	Copy contents of BX into AX.

"Destination" operand

"Source" operand

Instruction Operands

Constant

- Immediate Mode
  - Embedded within representation of instruction.

Register

- Register Mode

I/O Port

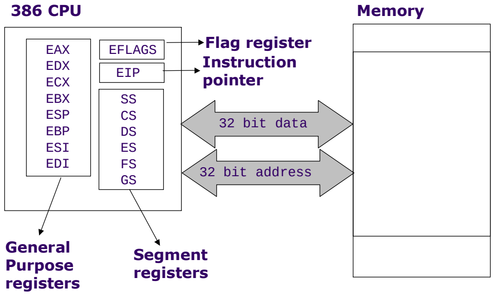
Memory Location

- Real Mode:

  Address = RB + RI + constant
- Protected Mode:

  Address = R1 + C1 × R2 + C2

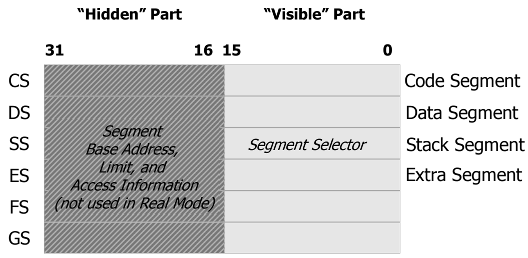
386 Registers



General Purpose Registers

	31	16 15	0
(E)AX: Accumulator	MSW of EAX	AH	AL
(E)BX: Base Register	MSW of EBX	BH	BL
(E)CX: Count Register	MSW of ECX	CH	CL
(E)DX: Data Register	MSW of EDX	DH	DL
(E)SP: Stack Pointer	MSW of ESP	SP	
(E)BP: Base Pointer	MSW of EBP	BP	
(E)SI: Source Index	MSW of ESI	SI	
(E)DI: Destination Index	MSW of EDI	DI	

Segment Registers



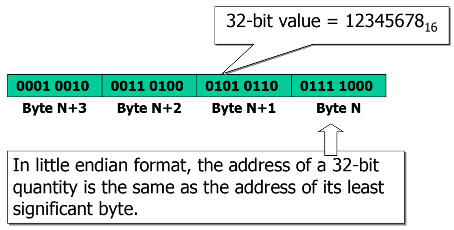
Flags Register

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
				OF	DF	IF	TF	SF	ZF		AF		PF		CF

Flag	*Bit*	*Description*
Overflow	11	Previous result caused arithmetic overflow.
Direction	10	1 = auto-decrement, 0 = auto-increment.
Interrupt Enable	9	Interrupts are enabled
Trap	8	Single step mode enabled
Sign	7	Previous result was negative
Zero	6	Previous result was zero
Auxiliary Carry	4	Previous result produced a BCD carry
Parity	2	Previous result had even parity
Carry	0	Previous result produced a carry put of MSB

Endianness

Byte ordering of 32-bit value



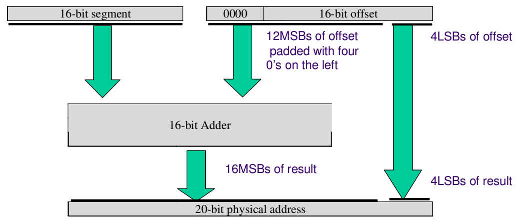
The Stack

Instruction sequence:	*Address*	*Memory contents*	
			High Address
PUSH EBX	SS:[ESP+10]	value from EBX(32 bits)	*Stack "grows" downward.* ↓
PUSH AX	SS:[ESP+8]	value from AX (16 bits)	
PUSH CS	SS:[ESP+4]	value from CS (32 bits)	
PUSH EDX	SS:[ESP]	value from EDX (32 bits)	← ***Top of stack***
			low Address

Part 2 X86 Review

- A Programmer's View of Computer Organization
- X86 Processor architecture
- Intel X86 assembly
  - Addressing Modes
  - Basic assembly
  - Mixed with C

Addressing in Real Mode

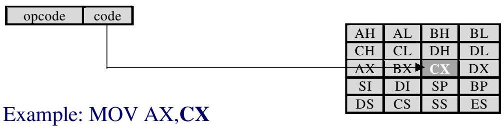


Immediate and Register Modes

opcode	16-bit operand

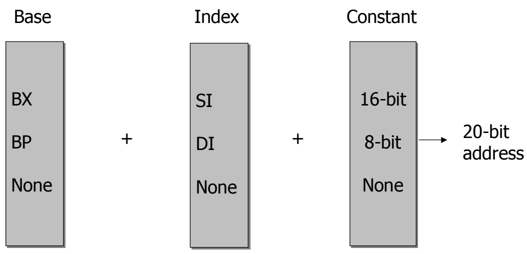
Operand is embedded within instruction representation.

Example: MOV AX,**12345**

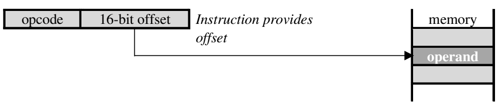


Example: MOV AX,**CX**

Memory Operands - Real Mode



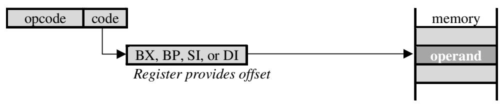
Direct Addressing



Address = ~~R_B~~ + ~~R_I~~ + constant

Example: MOV AX,**[TOTAL]**

Register Indirect Addressing

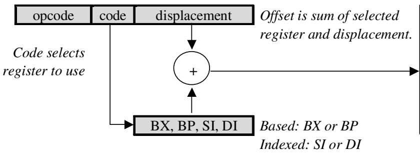


Address = R_B + ~~R_I~~ + ~~constant~~

or Address = ~~R_B~~ + R_I + ~~constant~~

Example: MOV AX,**[BX]**

Based and Indexed Addressing



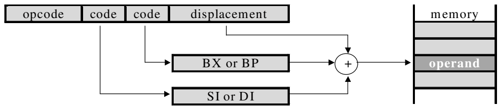
Address = R_B + ~~R_I~~ + constant

or

Address = ~~R_B~~ + R_I + constant

Example: MOV AX,**[BX+3]**

Based-Indexed Addressing



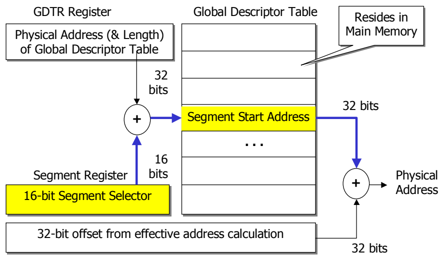
Address = R_B + R_I + constant

Example: MOV AX,**[BX+SI+3]**

Memory Addressing – Protected Mode

- Memory Address on 32 bits => 4 GB address space
- Generalized segmentation concept
- More GPRs can be used for Base,Iindex

Memory Addressing in Protected Mode



The Flat Memory Model

- GDT configured so that all segments start at physical address zero and have a size of 4GB. (so, e.g. CS loaded with offset of the descriptor for code segment)
- There's a one-to-one correspondence between physical addresses and the 32-bit offsets produced by effective address calculations.
- Memory looks like a single continuous space, called a *linear address space*.

Addressing Memory in Protected Mode

Constant

- Immediate Mode
  - Embedded within representation of instruction.

Register

- Register Mode

I/O Port

Memory Location

- Protected Mode:
  Address = $R_1 + C_1 \times R_2 + C_2$

Effective Address Calculation in Protected Mode

Base		Index		Scale Factor		Displacement		
EAX EBX ECX EDX ESI EDI EBP ESP None	+	EAX EBX ECX EDX ESI EDI EBP None	×	1 2 3 4	+	None 8-bit 16-bit 32-bit	⟶	32-bit address

I/O Port Addressing

An I/O port can be addressed with either an immediate operand or a value in the DX register.

As I/O port address bus is 12 bits wide, immediate operand < 4096, and the address > 4096 has to be preload to DX for addressing

Data Movement Instructions

MOV dst,src ; dst ← src

LEA reg_{32},mem ; reg_{32} ← $offset_{32}$ (mem)

MOVZX reg_{32},src ; reg_{32} ← zero extended src

MOVSX reg_{32},src ; reg_{32} ← sign extended src

XCHG dst,src ; temp ← dst
dst ← src
src ← temp

Stack Instructions

PUSH src_{16} ; ESP ← ESP-2, MEM[SS:ESP] ← src_{16}
PUSH src_{32} ; ESP ← ESP-4, MEM[SS:ESP] ← src_{32}
PUSHF ; ESP ← ESP-4, MEM[SS:ESP] ← EFlags
PUSHA ; Pushes EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI

POP dst_{16} ; dst_{16} ← MEM[SS:ESP], ESP ← ESP+2
POP dst_{32} ; dst_{32} ← MEM[SS:ESP], ESP ← ESP+4
POPF ; EFlags ← MEM[SS:ESP], ESP ← ESP+4
POPA ; Pops EDI, ESI, EBP, skip, EBX, EDX, ECX, EAX

Arithmetic Instructions

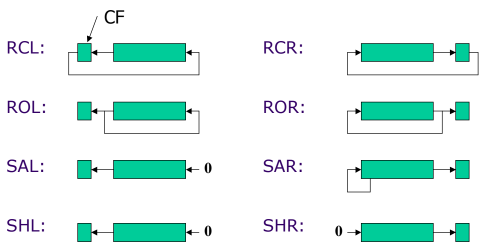
ADD dst,src
ADC dst,src
SUB dst,src
SBB dst,src
INC dst
DEC dst
NEG dst

MUL src ; unsigned
IMUL src ; signed
DIV src ; unsigned
IDIV src ; signed
CBW
CWD/CDQ
CMP dst,src

Bitwise Logical Instructions

AND dst,src ; dst ← dst & src
OR dst,src ; dst ← dst | src
XOR dst,src ; dst ← dst ^ src
NOT dst ; dst ← ~dst
TEST dst,src ; dst & src

Shift Instructions: opcode reg counter



Conditional Jump Instructions

Signed Tests:

JG/JNLE label
JGE/JNL label
JL/JNGE label
JLE/JNG label

Unsigned Tests:

JA/JNBE label
JAE/JNB label
JB/JNAE label
JBE/JNA label

Equality Tests:

JE/JZ label
JNE/JNZ label

Other Tests:

JC, JNC, JO, JNO, JS, JNS, JPO, JNP, JCXZ

Jump Instructions

Compare	*Mnemonic(s)*	*Jump if . . .*	*Determined by . . .*
equality	JE (JZ)	Equal (Zero)	ZF==1
	JNE (JNZ)	Not Equal (Not Zero)	ZF==0
unsigned	JB (JNAE)	Below (Not Above or Equal)	CF==1
	JBE (JNA)	Below or Equal (Not Above)	CF==1 || ZF==1
	JAE (JNB)	Above or Equal (Not Below)	CF==0
	JA (JNBE)	Above (Not Below or Equal)	CF==0 && ZF==0
signed	JL (JNGE)	Less than (Not Greater than or Equal)	SF!=OF
	JLE (JNG)	Less than or Equal (Not Greater than)	SF!=OF || ZF==1
	JGE (JNL)	Greater than or Equal (Not Less than)	SF==OF
	JG (JNLE)	Greater than (Not Less than or Equal)	SF==OF && ZF==0

Other Jump Instructions

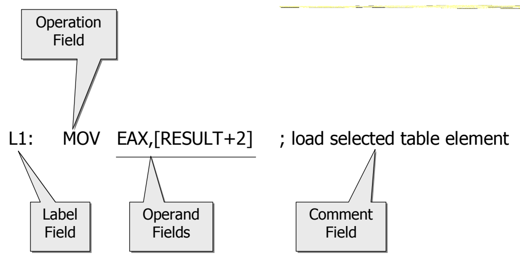
Unconditional:

JMP label
JMP regptr
JMP memptr

Loops (count in register ECX):

LOOP short-label
LOOPE/LOOPZ short-label
LOOPNE/LOOPNZ short-label

NASM syntax



Example: Break and End of Loop

```
for (;;)                top_of_for:  ...
  {
  ...                               ...
  if (...) break ;                  JMP  end_of_for
  ...                               ...
  }                                 JMP  top_of_for
                        end_of_for: ...
```

Examples: WHILE loop, IF-THEN-ELSE

```
while (x < 1000)          top_of_while:   CMP   DWORD [x],1000
 {                                        JNL   end_of_while ; >=
   ...                                    ...
 }                                        JMP   top_of_while
                          end_of_while:
```

```
if (x > y)            MOV   EAX,[x]  ; x > y ?
{                     CMP   EAX,[y]
      x = 0 ;         JNG   L1       ; x<=y jump
}                     MOV   DWORD [x],0   ; then: x = 0 ;
 else                 JMP   L2            ; skip over else
{                     L1:   MOV   DWORD [y],0 ; else: y = 0 ;
      y = 0 ;         L2:   ...
}
```


Example: Loop With JECXZ and LOOP

```
          MOV     ECX,[iteration_count]
          JECXZ   loop_exit     ; jump if ECX is zero.
top_of_loop:
          ...
          <Register ECX: N, N-1, ... 1>
          ...
          LOOP    top_of_loop   ; decrement ECX & jump if NZ
loop_exit:
```


Interfacing C and Assembly

Register(s)	*Usage in C functions*
EAX	Functions return all pointers and integer values up to 32-bits in this register.
EDX and EAX	Functions return 64-bit values (long long ints) in this register pair. (Note: EDX holds bits 63-32, EAX holds bits 31-0).
EBP	Used to access: (1) The arguments that were passed to a function when it was called, and (2) any automatic variables allocated by the function.
EBX, ESI, EDI, EBP, DS, ES, SS.	These registers must be preserved by functions written in assembly language. Any of these registers that the function modifies should be pushed on entry to the function and popped on exit.
EAX, ECX, EDX, FS, GS	"Scratch" registers. These registers may be used without preserving their current content.
DS, ES, SS, EBP, ESP	Used to reference data. If modified by a function, the current contents of these registers must be preserved on entry and restored on *return*.

Function Calls and Return

- CALL instruction used by caller to invoke the function
  - Pushes the return address onto the stack.
- RET instruction used in function to return to caller.
  - Pops the return address off the stack.

Examples of Functions in Assembly

Function Call with no Parameters and No Return Values

C prototype:	void Disable_Ints(void) ;
Example usage:	Disable_Ints() ;
Generated code:	CALL _Disable_Ints
NASM source code for the function:	_Disable_Ints: CLI ; Disables interrupt system RET ; Return from function

Examples of Functions in Assembly

Function Call with no Parameters and 8-bit Return Values

C prototype:	BYTE8 LPT1_Status(void) ;
Example usage:	status = LPT1_Status() ;
Generated code:	CALL _LPT1_Status ; returns status in EAX MOV [_status],AL
NASM source code for the function:	_LPT1_Status: MOV DX,03BDh ; Load DX w/hex I/O adr XOR EAX,EAX ; Pre-Zero rest of EAX IN AL,DX ; Get status byte from port. RET ; Return from function.

Parameter Passing for Function Calls

- Parameters are pushed onto stack prior to CALL.
  - gcc pushes parameters in reverse order.
  - 8/16-bit parameters are extended to 32-bits
- Caller should remove parameters after function returns.

Examples of Functions in Assembly

Function Call with 2 Parameters and No Return Values

Function call w/parameters:	Byte2Port(0x3BC, data) ;

Code generated by the compiler:	PUSH DWORD [_data] ; Push 2nd param
	MOV EAX,03BCh ; Push 1st param
	PUSH EAX
	CALL _Byte2Port ; Call the function.
	ADD ESP,8 ; Remove params

Examples of Functions in Assembly

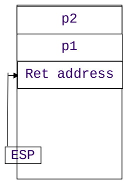
Function Call with a 64-bit Parameter

C	*Assembly*
/* signed or unsigned */ long long data ; ... Do_Something(data) ; ...	PUSH DWORD [_data+4] PUSH DWORD [_data] CALL _Do_Something ADD ESP,8

Retrieving Parameters from Stack

- Must access parameters without actually removing them from the stack!
- Can't use POP instructions to access parameters.
  - Parameters expect to be removed from the stack later by the caller
  - RET instruction expects return address to be on top of the stack

```
_Swap:
    MOV   ECX,[ESP+4] ; Copy parameter p1 to ECX
    MOV   EDX,[ESP+8] ; Copy parameter p2 to EDX
    MOV   EAX,[ECX]   ; Copy *p1 into EAX
    XCHG  EAX,[EDX]   ; Exchange EAX with *p2
    MOV   [ECX],EAX   ; Copy EAX into *p1
    RET               ; Return from this function
```



Polled Serial Input

```
_Serial_Input:
      MOV   DX,02FDh       ; DX  Status Port Address
SI1:  IN    AL,DX          ; Read Input Status Port
      TEST  AL,00000001B   ; Check the "Ready" Bit
      JZ    SI1            ; Continue to wait if not ready
      MOV   DX,02F8h       ; Else load DX with Data Port Address
      XOR   EAX,EAX        ; Pre-clear most significant bits of EAX
      IN    AL,DX          ; Read Data Port
      RET                  ; return to caller with data in EAX
```


Interrupt Service Routine for Serial Input

```
_Serial_Input_ISR:
   STI                          ; Enable higher priority interrupts
   PUSH   EAX                   ; Preserve contents of EAX and EDX.
   PUSH   EDX
   MOV DX,02FDh                 ; Retrieve the data and clear the request.
   IN   AL,DX
   MOV [_serial_data],AL        ; Save the data away
   MOV AL,00100000b             ; Send EOI (end-of-interrupt)command to
   OUT 20h,AL                   ;   Programmable Interrupt Controller
   POP  EDX                     ; Restore original contents of the registers
   POP  EAX
   IRET                         ; Restore EIP and EFlags.
```

Review

- Lab material available at
  - http://www.ida.liu.se/~TDDI11
- Register in webreg for the labs
  - http://www.ida.liu.se/webreg
- Be Well Prepared for the labs
- Demo and Codes of 5 assignments have to be shown in 9 lab sessions (no other time!!)