Announcement

- Copies of relevant sections on Intel Architecture and assembly have been made available in the Lab. Please check them out. The available textbooks on Intel architecture teaches Intel assembly.

- However, since Linux community uses ATT assembly and Linux has become popular, the latest version of RTOS in the lab supports only ATT assembly for Intel. Thus, you need to translate Intel assembly to ATT assembly when you do lab exercise.

- Useful references for ATT assembly vs Intel assembly include
- http://www.delorie.com/djgpp/doc/brennan/brennan_att_inline_djgpp.html (intro)
- http://www-106.ibm.com/developerworks/linux/library/l-ia.html?dwzone=linux (best)

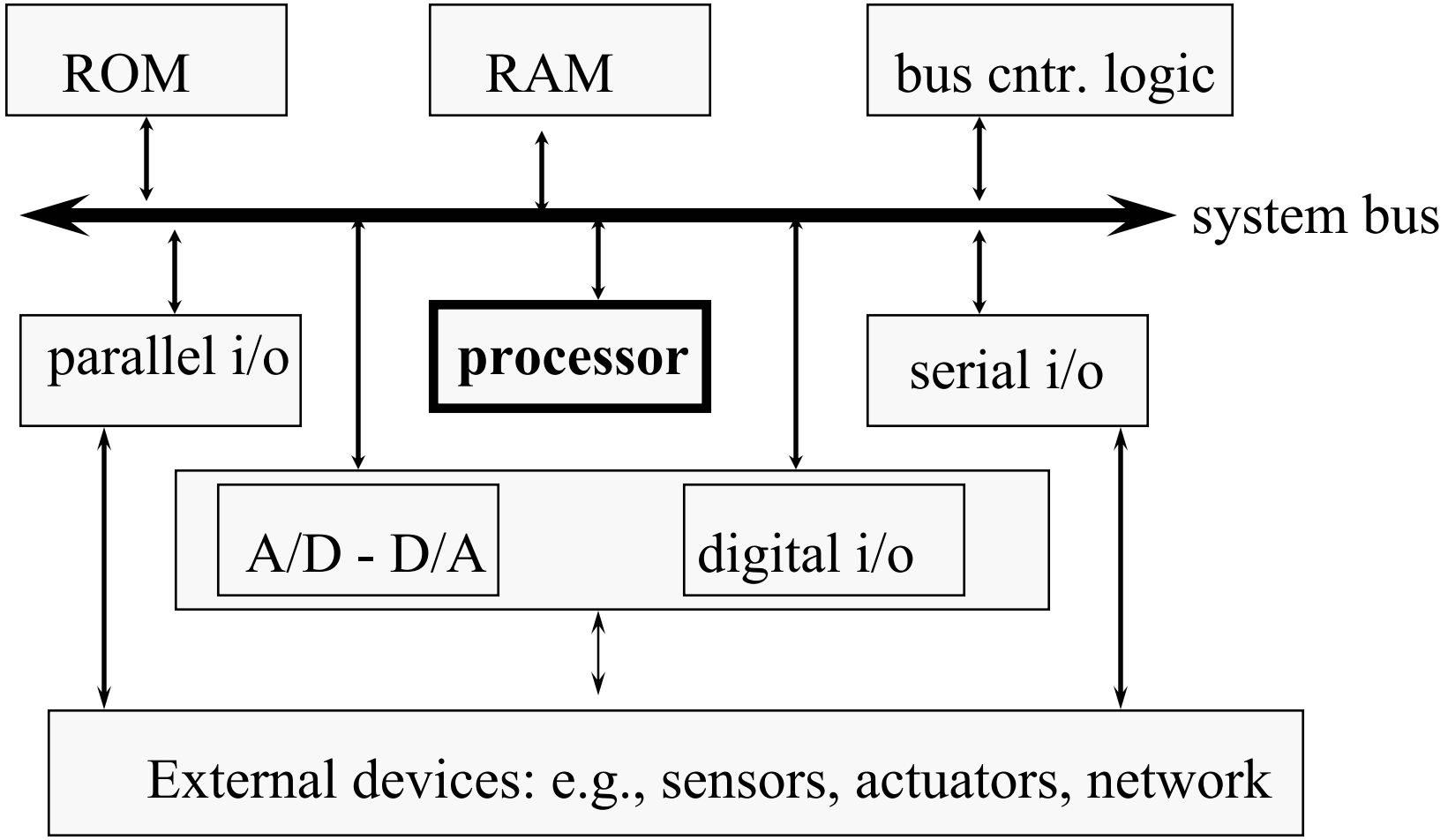
- http://linuxassembly.org/resources.html#tutorials (links to many tutorials)

Lecture 2

- Key message of last lecture:
  - Don’t passively listen, think actively.
  - Exercise your thinking at higher levels.

- Objective of Lectures 2, 3 and 4
  - Understand the programming environment
  - Master the basic assembly, in particular inline assembly and data transfer instructions needed for device I/O
  - Able to use inline assembly to talk to A/D-D/A card in lab1
- Today
  - Embedded software characteristics
  - Programming languages and programming environment
  - Review Intel assembly language

A Typical Embedded System



Programming Environment

- Open standard RTOS: POSIX RT extension, e.g., Lynx OS and now Solaris and various real time Linux.
- Commercial RTOS: Vxworks, Windows CE
- Language specific runtimes: embedded Java runtime (being developed), Ada 95 runtime
- Real time scheduling analysis tools, e.g. PERTS and TimeSys
- Distributed system integration tools: CORBA real time extensions

- Self-hosting: development and target machine is the same one
- Host+target: develop and compiled at the host, download and executed at the target.

- In our lab, we will use the “self-hosting” environment. In real world embedded applications, you often develop and test in a self-host environment and then download it to a target machine for real applications.

- Since downloading is rather mechanical and time consuming, we will skip this step in the lab.

Desirable Characteristics of Embedded System Software

- Efficiency: fast and compact.
- Reusability: encapsulate specific hardware related code, modular and easy to read
- Reliability: use simple and well understood algorithms and structures.
- Timing predictability: you can assure that deadlines are met.

Major Programming Language

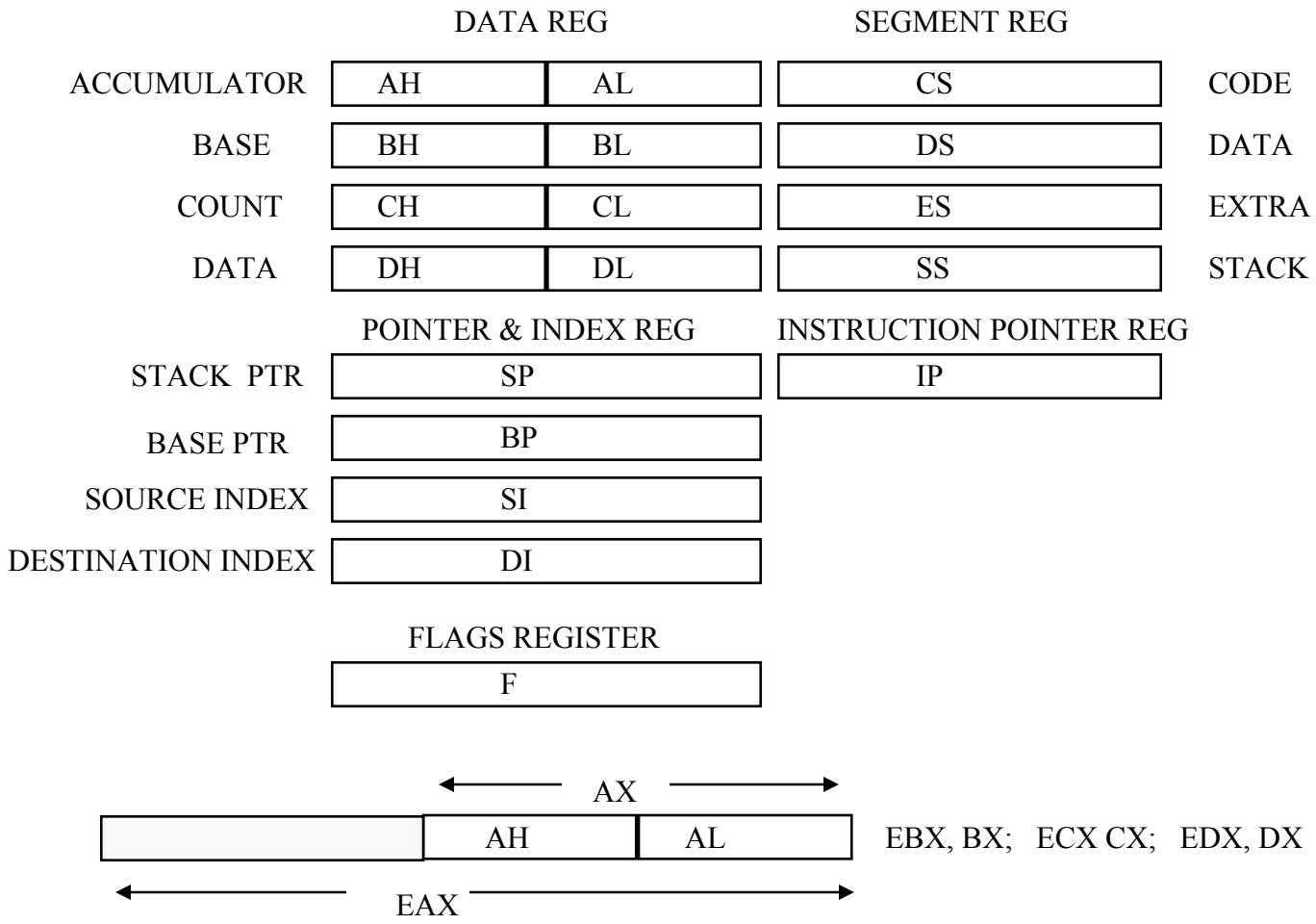
- Ada
  - Was mandated by DoD for many of their projects.
  - Chosen by companies for safety critical applications, e.g., Airbus and Boeing 777 flight control.
  - Try to help enforce good software engineering practices.
  - Not widely supported.
- Java
  - Simple, portable, multi-threaded.
  - Get rid of some of worst “features” of C.
  - Real time garbage collection are being solved
  - Available soon?

Major Programming Languages

- C/C++
  - Popular and C's efficiency close to assembly
  - Supports but does not enforce good software engineering practice (backward compatibility to C)
  - Widely supported. We will use C/C++ until real time JAVA is widely supported by RTOS

- Assembly
  - Processor specific.
  - Small and fast (compared to high level languages)
  - High development time and cost
  - **Still needed when interface with hardware.**
  - (We will teach Intel since PCs are cheap, widely used and supported.)

Programming View of Intel Processor Registers



Intel Assembly Language

- Types of Statement:
  - executable instruction
    - translated into machine code
  - assembler directive
    - tell the assembler how to do the translation
- Categories of Instructions
  - Data Transfer: e.g., IN, OUT, MOV
  - Arithmetic: e.g., ADC (add with carry)
  - Logic: e.g., AND, OR, NOT
  - Execution flow: e.g., JMP, JMZ (jump if zero)
  - Processor control: e.g., STC (set carry)

Protected Mode Addressing

- Protected mode: 32 bit linear addresses divided into 8,192 segments.
  - virtual memory paging support
  - protection:
    - type check e.g. write into read only data segments,
    - 20 bit segment address limit check,
    - addressable page domain check,
    - procedure entrance point check,
    - privileged level check

Real Mode Addressing

- Legacy of 20 bit address (00000 to FFFFF) the 1st 1 M byte (1, 048,576) memory.
  - divided into 16 64k blocks called segment
  - A segment can start at any 16 bit word boundary
  - Segment's starting address stored in segment registers
  - Addresses are generated by shifting segment register content to the left and then adding a 16 bit offset to address within the 64k segment.

  - Limited capability but easy to use for assembly codes.
  - It is adequate for most device I/O operations.

- We want to keep assembly to the minimum. We will use them only when it is unavoidable such as writing new device I/O code. As a result, we will limit ourselves to real mode assembly in this class.

Register and Immediate Addressing

- Register addressing: MOV AX, BX ; <destination> , <source>
- Immediate addressing: MOV AX, 100H
- Example 1:
  - AX = 1000H
  - MOV AX 01H
  - AX = _____
- Example 2:
  - AX= 1000H
  - MOV BX 01H
  - MOV AL, BL
  - AX = _____

- 01 H
- 1001H

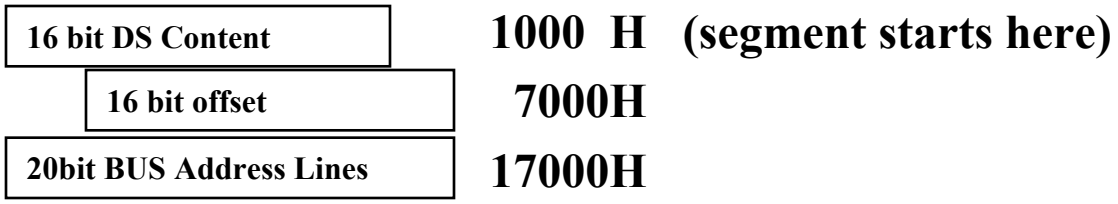
Direct (memory) Addressing

The physical address used in real mode is 20 bits, but the data segment register has only 16 bits. How can we create 20 bit address using 16 bit DS register?

Intel's solution is: 1) shift the DS value 4 bits to the left (16 + 4 = 20)
2) add a 16 bit offset to address 64k locations

How to move 10H to memory location 17000H? Suppose that we would like the segment starts at 10000H.

- MOV AX, 1000H (remember: hardware will turn 1000H to 10000H by shifting 4 bits to left)
- MOV DS, AX (MOV DS,1000H is not allowed)
- MOV AX, 10H
- MOV [7000H], AX (segment starts at 10000H, 7000H is offset)



Summary and Next Class

- We have reviewed the software development for embedded systems.
- Most of assembly codes can be replaced by high level language codes, and we should do so.
- Data movements between processors and external devices still need assembly codes.
- We shall study addressing modes and data movement assembly codes in more details next class.

Appendix: Intel to ATT Mapping

To prepare for the Lab, please read

http://www.delorie.com/djgpp/doc/brennan/brennan_att_inline_djgpp.html
Here are some examples:

•Reference a register
- •AT&T: %eax
- •Intel: eax

•Source/Destination Ordering:
In AT&T syntax, the source is *always* on the left, and the destination is *always* on the right. Under Intel, it is the other way.

•So let's load ebx with the value in eax:
- •AT&T: movl %eax, %ebx
- •Intel: mov ebx, eax

Also in AT&T: the register size is often made explicit:
"l" is for 32-bits, "w" is for 16 and "b" is for 8.

Appendix: Some More Common AT&T Assembly

- a eax
- b ebx
- c ecx
- d edx
- s esi
- d edi
- I constant value (0 to 31)
- r let compiler to pick any register
- ...

- Inline programming
- asm (…)
  - When refer to eax, write it as %%eax
  - %0 is the first variable, %1 the second.
  - Outputs go from %0...%k, and inputs go from %k+1...%n.