CS 241
Computer Architecture and Organization

Professor Richard Eckhouse

Office: W-2-013

Phone: 617-287-5776

email: eckhouse@cs.umb.edu

Introduction

- **Particulars**
  - **Information sheet**
  - **Syllabus**
  - **Textbooks and reference books**
  - **Laboratories**
  - **Machine problems**
  - **Examinations**
- **What this course is about**
  - **Distinction between computer architecture, organization, and implementation**
  - **Experience with standard PCs both in the laboratory and remotely**

Course Objectives

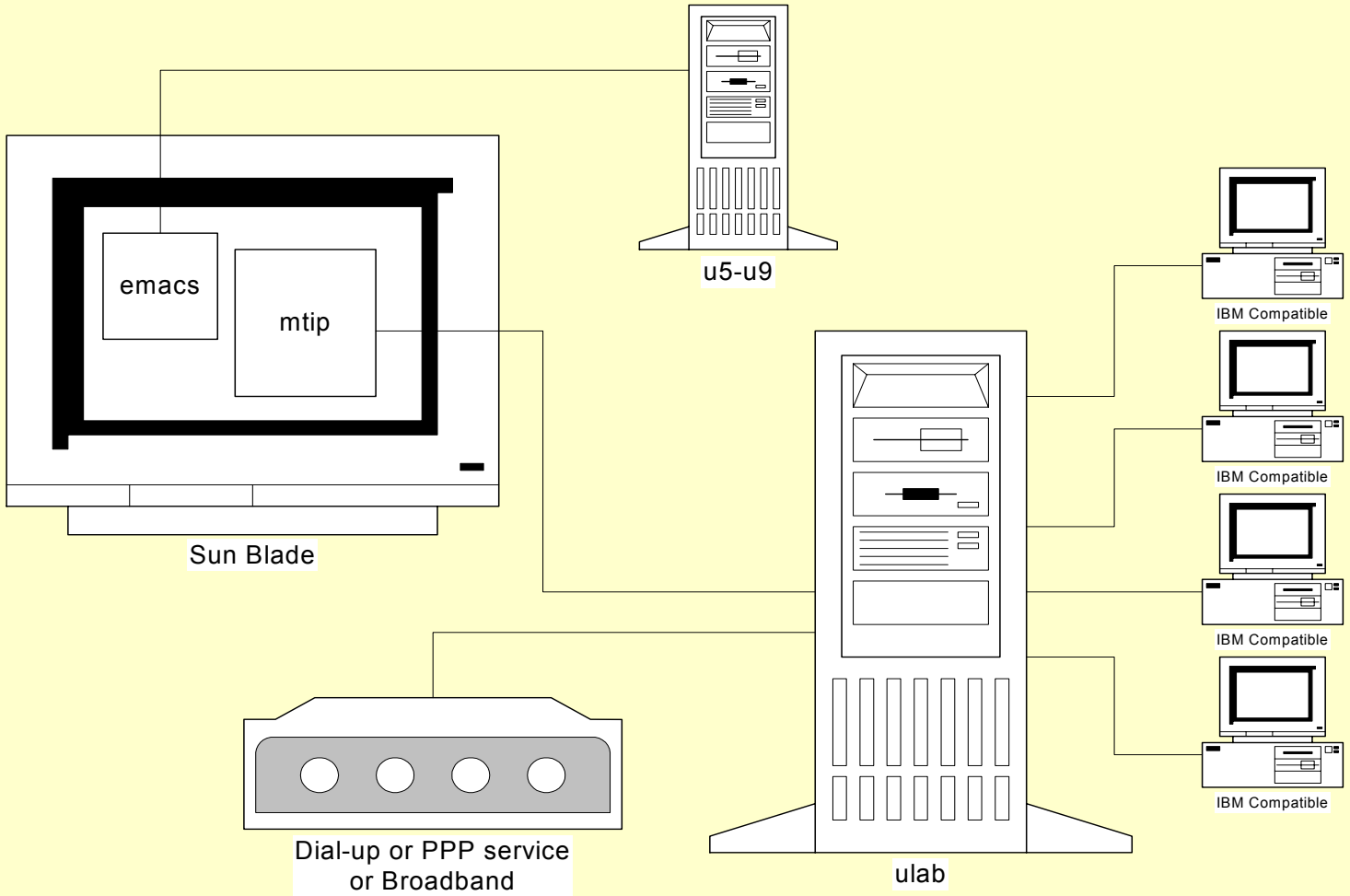
- **Provide hands-on experience with assembly language programming**
- **Understand the concepts of instruction sets architecture, interrupts, serial/parallel ports, digital logic, and memory/cache designs**
- **Gain some insight into how microcomputers work - the good, the bad, and the ugly - with the usual associated trade-offs**

Note: *the textbook serves as a reference to fill in the lectures -- may not be covered in detail, but you are responsible for the material*

Getting Started

- **Lab:**
  - **Currently room 142 (has push-button lock)**
  - **Lab instructors: John Lentz (jlentz@cs.umb.edu) and Andrew Davis (adavis@cs.umb.edu)**
- **Modules:**
  - **Adding a new module in .cshrc**
    - » **"module load standard ulab"**
  - **Defines environment variables**
  - **Makes it possible to compile and run programs, fetch examples, get necessary files**
- **First assignment:**
  - **Get mp1.txt from course directory along with *.c and *.h files**
  - **Printout and read $pcex/pc.handout and $pcex/test.c**

The "Big" Picture



Example

- Want to compile and run test.c (bad name for a program) on Unix and the SAPC ("stand alone PC")
  - Use Makefile
  - Produce both Sun and PC objects (.lnx extension)
    - » ex. `gcc -o utest test.c`
  - Uploading to the SAPC
  - Running
  - Exiting
- Getting help, type: "help mtip"
- How about running it in the lab?

Basic PC Hardware

- CPU, memory, cache, bus slots on motherboard
- Serial and parallel ports, floppy and hard disk controllers on I/O board
- Video card
- Keyboard and mouse
- Variations do not change the architecture

PC Software

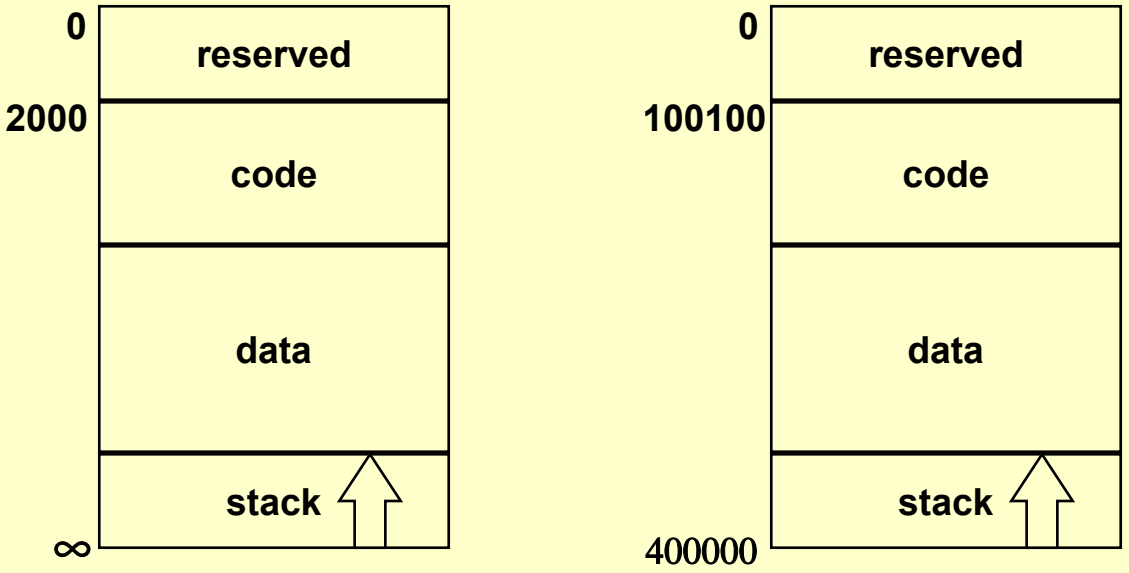
- 16-bit “real mode” and 64K segments
  - BIOS - Basic Input/Output Services
  - DOS - Disk Operating System
  - Various EMS (Expanded Memory Systems) extensions
  - Windows
    - » Using extended memory
- Modern operating systems using protected mode
  - OS/2
  - Linux
  - Windows 95, 98, ME, NT, 2000, and XP

C Programming Environment

- What do we know about C, makefiles, and Unix?
- Understanding SAPCs on Unix
  - Portability - limited if we directly access the hardware
  - Operating Systems - separate users from hardware; a great advantage!
  - Program test.c (again, a bad choice of names)
    - » Why is it portable?
    - » What do we do to run it under Unix?
    - » Use the -v switch (i.e., gcc -v) to see the preprocessor, compile, assemble, and link stages
      - How do you find out about such things?
    - » Use the -E switch to run only the C preprocessor
    - » Other switches: -S and -c

C Programming Environment (Cont'd)

- **Cross-compiling**
  - **Replace gcc with i386-gcc**
  - **Whether Unix or SAPCs, process is the same!**
- **How is our program loaded into memory under Unix? SAPC?**



Differences Between Unix and SAPC

- **No kernel on SAPC**
  - **Use Tutor**
  - **No files**
- **C compiler for SAPC**
  - **No malloc**
  - **No predefined string arrays**
  - **Can gain access to hardware**

Machine Project 1

- Need to make mp1 subdirectory under cs241
- Examine *cmds.c* and *makefile*
- Modify *cmds.c* to add commands
- Compile, download, execute
  - Remember to use scriptfiles
- Have a Tutor to use in testing
  - Important differences in md on Tutor and tutor
- Is your code portable?
  - Can it run on both the Unix machines and the SAPCs?

Machine Project 1 (Cont'd)

- **Key to understand the problem is the typedef:**

```
typedef struct {
  char *cmdtoken;  /* md or whatever--char string to invoke cmd */
  int (*cmdfn)();  /* function to call when you see this cmd */
  char *help;      /* helpstring for cmd */
              } cmd;
```

- **which defines the commands in the command table:**

```
cmd cmds[] = {{"md", mem_display, "Memory display: MD <addr>"},
              {"x",  xit,         "Exit" },
              {NULL, NULL,         NULL}}; /* null cmd to flag end of table */
```

- **and a parsed 'md' command calls:**

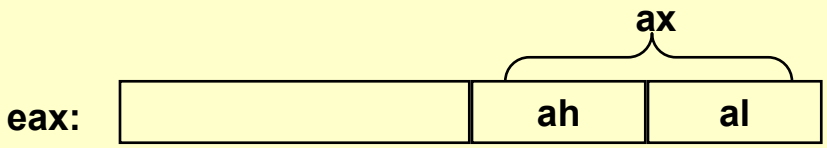
```
mem_display(cmd *cp, char *arguments)
```

Machine Project 1 (Cont'd)

- so your job entails
  - converting the character string to an integer
  - using the binary number as an address
  - finding the contents of the address
  - displaying those contents
- plus
  - adding other commands 'ms' and 'h' to the command table
  - writing code to make it all work!
- Are your commands case sensitive?

Accessing Hardware

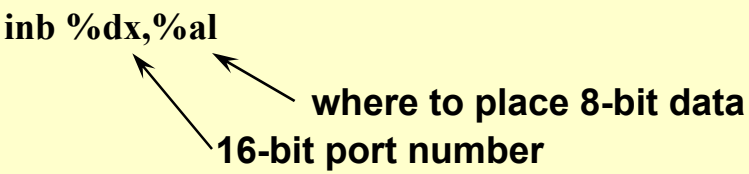
- Can't do it under Unix
  - Why not?
- Have the SAPCs to allow us to learn about hardware from "hands-on" experience
- Must start with some PC basics (486 machines)
  - Have eight 32-bit registers (note the missing "general purpose")
  - Example:



  - Same is true for ebx, ecx, and edx (esp is always 32-bits) plus:
    - » A program counter (PC) known as eip
    - » And a status register (SR) known as eflags
    - » Some segment registers fixed at boot time

Accessing Hardware (Cont'd)

- Can we examine the contents of these registers?
  - Yes, use Tutor with rd command; use 'all' to see more than base set
- What is the instruction set like
  - Familiar 'mov', 'add', 'inc', 'jmp', …
- I/O instructions using a 'port'
  - 'in' and 'out' format for 64K different devices (but total is less because some devices use many ports)
  - Works even if nothing is connected to the port!
  - Form is:

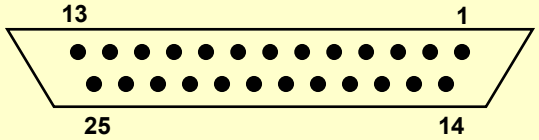


Accessing Hardware (Cont'd)

- **More I/O information**
  - Format for output is: `outb %al, %dx`
  - Could also use 'pd' in Tutor, ex. `Tutor> pd 200`
  - And 'ps', ex. `Tutor> ps 378 FF`
    - » Which sets all the data pins on that port to a logic 1
    - » What does '`ps 378 0`' do?
- **A real port is the parallel port, LPT1, which we will examine in Lab 3**

Parallel Port

- Visually is a DB25 connector on back of computer



  - Data appears on pins 2-9, control/status on pins 1 & 10-17; pins 18-25 are ground
  - "TTL" signals
    - » v ≈ 0-1volts is considered low and a logic 0
    - » v ≈ 3-5volts is considered high and a logic 1
  - IBM defines up to three parallel port addresses but we will use 378h as base address (see S&S, page 628)
    - » **base** used to send data to printer
    - » **base+1** used to get status
    - » **base+2** used for control

Parallel Port (Cont'd)

- Can test PP by using 'ps' command of Tutor
  - `ps 378 FF` to set to all ones
  - `ps 378 0` to set to all zeros
- Very simple interface
  - Provides access to backplane bus
  - No transformation of data; simple protocol to use
  - Can be accessed from C
- Port access
  - Could use those built into libraries specific to the PC
  - We have our own (cpu.h)

Accessing the Parallel Port

- **All the necessary information is in cpu.h**

  void outpt(int port, unsigned char outbyte);

  unsigned char inpt(int port);

  - **port < 64K and outbyte is the 8-bit character**
  - **Example: `outpt(0x378, 0xFF);`**
- **Don't want wired in numbers so look at lp.h**

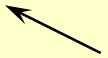
```
#define  LPT1_BASE  0x378
#define  LPT2_BASE  0x278
#define LP_DATA  0        /* out, in: 8 bits of data */
#define LP_STATUS 1       /* in: status bits */
#define LP_CNTRL  2       /* in, out: control bits */
```

- **Examine testlp.c**

Accessing the Parallel Port (Cont'd)

- Note that status is a read only port
- Some examples

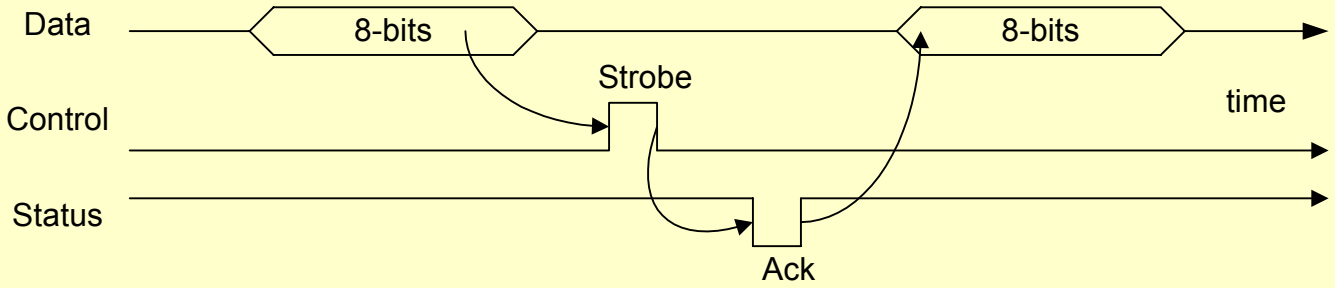
```
pd 378
  0378 00 7F E0 . . .
ps 378 55
pd 378
  0378 55 7F E0 . . .
ps 379 66
pd 378
  0378 55 7F E0 . . . {no effect}
                1110 0000
```



LP_PINTEN ≡ 0 {inits to off}

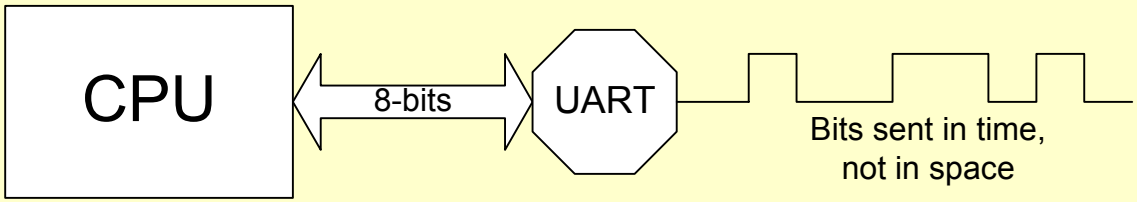
Parallel Port Printing

- "Handshaking" protocol
- Data byte sent to parallel data port
  - All bits sent at once (i.e., parallel transfer)
  - No parity bits
  - Printer "strobed" through control port to say data byte is ready
  - Printer "acks" or acknowledges data received or gives an error indication on status port



Serial Ports (COM Ports)

- Single wire for data out and single wire for data in plus return path (i.e., ground)
- Bits are passed one at a time in sequence
  - Internally, the data is passed in parallel from the CPU over the bus to the serial port interface
  - When the sequence starts and stops has to be specified
  - How the bits are serialized has to be specified



Serial Ports (Cont'd)

- **Signal levels for serial ports come from RS232 specification and are not 0 to 5 volts**
  - **RS232 signal levels are -15 to 15 volts, nominally**
- **PC specification allows up to four serial ports**
  - **COM1: base address is 3F8**
  - **COM2: base address is 2F8**
  - **Both have up to eight port addresses**
    - » **Base: receiver buffer on read / transmit buffer on write**
    - » **Base+1 and Base+2: interrupts and FIFO buffer**
    - » **Base+3: line control set up by Tutor**
    - » **Base+4: modem control**
    - » **Base+5: line status**

Serial Ports (Cont'd)

- **Examples:**

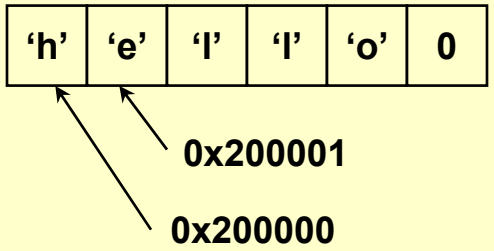
  `ps 2F8 41` will send an 'A' out on com2, the online console and you will see **ATutor>**

  `pd 2F8` will produce **02F8 0d . . .**

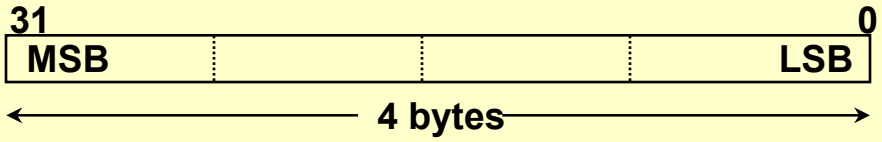
- **Many examples on-line**
  - Take a look at serial.h
  - Find things in $pcinc and $pcex
  - Use echo.c to read a character by polling until data is ready
- **Next homework on assembler but will look at this**

Strings

- How are strings stored?
  - Assume string ‘hello’ is stored in memory starting at location 0x200000
  - Since memory is “byte addressable” each character (i.e., byte) has an address



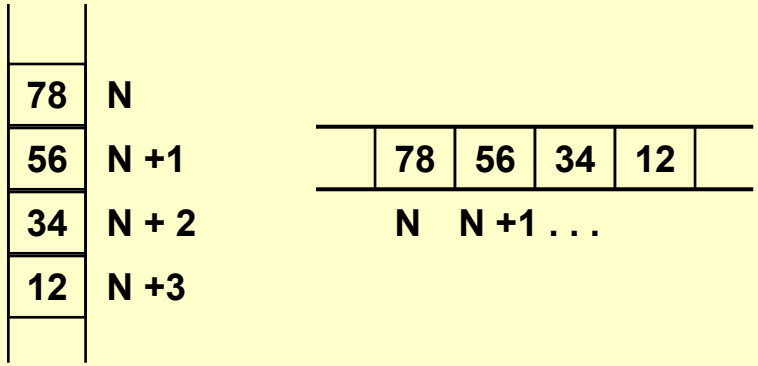
  - But numbers are stored a 32-bit integers



  - So how is the register stored in memory with a movl?

Copying Bytes

- **Little-endian used with the PC**
  - **Example: 0x12345678 stored in memory**



  - **So what is big-endian? Motorola and IBM use it.**
  - **What happens when you move from a register to memory?**
    - » **Same thing; MSB goes to highest addressed byte**

The Assembler for the PC

- Need to read "chap2.txt" found in $pcbook
  - We use the Gnu assembler which has a different syntax

Gas		Intel syntax	
`movw`	`$8, %ax`	`mov`	`ax, 8`
`addw`	`$3, %ax`	`add`	`ax, 3`
`movl`	`%eax, 0x200`	`mov`	`[200], eax`

  - Normally use full 32 bits for numbers and refer to the 32-bit registers like `%eax`
  - In Gas we have source on the left and destination on the right
- More convenient to let C do as much work as possible
  - Will thus make our Gas routines callable from C

Sum Two Numbers

- Assembly code with supporting C syntax

```
#sum2.s -- Sum of two numbers
.text
.globl  _sum2
_sum2:  movl   $8, %eax
        addl   $3, %eax
        ret                    #number in eax
```

- C "driver" code to execute sum2.s is called sum2c.c

```
extern int sum2(void)
void main(void) {
  printf("Sum2 returned %d\n", sum2());
}
```

Assembly and Driver Pair

- The makefile in $pcbook expects a ‘matched pair”
  - The assembler file is xxx.s and the C driver file is xxxc.c
  - The form of the make is:

    ```
    make A=myprog
    ```

  - Pick up the makefile from $pcbook
- Always read the makefile that goes with the program

Some Initial Conventions

- We will only use `%eax`, `%ecx`, and `%edx` to keep good relations with our "C caller" routine
- General syntactic construction for assembly code is:

```
                .globl  _mycode
                .text
_mycode:
                . . .
                ret
                .data
mydata:         .long  17
```

Forms of Addressing

- Have used % for registers and $ for immediate addressing using Gas
  - Intel does not use the $
  - But what is an immediate address?
- Direct addressing
  - Intel uses [ ] pair while Gas does not use anything
  - But what is a direct address?
  - Example:

```
        movl   %eax, total
        movl   total, %edx
          . . .
total: .long  0
```

Forms of Addressing (Cont'd)

- **Direct addressing**
  - Why can't we write something like:

```
movl   first, second
```

- **Symbolic addressing**
  - Of the form 'SYMBOL = value'
  - Example:

```
NCASES = 8
movl   $NCASES, %eax
```

- **Add instruction with immediate and direct addressing**

```
addl   $2, %eax
addl   item, %eax
addl   %edx, sum
```

Assembly Language Programming

- **Strategy**
  - **Move quantities into registers, compute, and move results back to memory**
  - **Registers like a scratchpad**
  - **Registers are local memory**
  - **Registers are part of the memory hierarchy**
    - » **What else is in the hierarchy?**
- **Why use registers rather than memory?**
- **What else are registers good for but holding intermediate results?**

mp2 – Passing Arguments

- How are you going to pass arguments, if you pass them at all?
- Option 1
  - Could write:

    ```
    cnt = counta("aabbccabcabc");
    ```

  - The called routine would receive a constant pointer to a string that is pushed on the stack
  - To remove the pointer from the stack the code would be:

    ```
    movl  4(%esp),%edx
    ```

  - Why the offset of 4? Why place the results in %edx

Passing Arguments (Cont'd)

- **Option 2**
  - Value stored in counta subroutine:

```
string: .asciz "aabbccabcabc"
```

  - Still need to fetch the string using the %edx register

```
movl  $string,%edx
```

Lewis Carroll, *Through the Looking Glass*

"The name of the song is called 'Haddocks' Eyes.'"
"Oh, that's the name of the song, is it?" Alice said trying to feel interested.

"No, you don't understand," the Knight said, looking a little vexed. "That's what the name is ***called.*** *The name really* ***is*** *'The Aged Aged Man****.'"***
"Then I ought to have said 'That's what the ***song*** *is called'?" Alice corrected herself.*

"No, you oughtn't: that's quite another thing! The ***song*** *is called 'Ways and Means': but that's only what it's* ***called, you*** *know!"*
"Well, what ***is*** *the song, then?" said Alice, who was by this time completely bewildered.*

"I was coming to that," the Knight said. "The song really ***is*** *'A-sitting On A Gate': and the tune's my own invention."*

Loops - The Heart of Programming?

- Many problems require repetitious operations
- How do you break out of a loop?
  - Use conditional branches (or jumps)
    - » Simplest loop instructions are: jne and jnz
    - » These instructions jump to another location in our code based on the z bit in the eflags register
  - Other conditional branches include jge, jle, je, jl, and jg
  - These are called “signed” conditional branches
    - » Are there “unsigned” conditional branches
  - What is the difference between signed and unsigned conditional branches?

Countdown Loop

- Sum up first ten integers

```
_sum10:     movl    $10, %ecx       # set up count
            movl    $0, %eax        # initialize sum
addint:     addl    %ecx, %eax      # add next value
            decl    %ecx            # decr and set z flag
            jnz     addint          # loop on z flag
            movl    %eax, sum       # if not a function
            ret                     # returning a value
```

- Loop instruction combines decrement, test, and branch conditional, so `decl/jnz` pair replaced by

```
            loop    addint          # decr and loop
```

Mixed Language Support

- Use most appropriate code to get the job done
  - Want to go back and forth between C and assembler
  - Can use printf to help debugging in assembler
  - Already have some rules to follow
    - » Use only %eax, %ecx, and %edx as "scratch" registers
    - » Assume C functions "clobber" the contents of these registers
    - » Assume the caller knows that the contents of these registers are clobbered
    - » If the values are to be preserved, then we need a save/restore mechanism to get through the call
    - » Most modern machines have a stack to support such a save/restore mechanism -- built into the x86 architecture

Push and Pop

- The **pushl** operation decrements %esp by 4 while the **popl** operation increments it by 4
- Also have **pushw** with corresponding pops
- Examples:

```
pushl  $7
pushl  %eax
pushl  x
popl   %edx
popl   x
popl   $7      # this won't work
```

Preserving Registers

- Want to save the contents of a register when we make a call to a C function

```
pushl  %eax
call   a_C_routine   #clobbers eax
popl   %eax
```

- What's missing here? Most C function/library calls take arguments

```
printf("%d", x);
```

a string pointer

a 32-bit integer

or, two 32-bit arguments

- So how do we call printf from our assembler code?

Calling a C Function

- Want to use something like printf inside some assembly code:

```
                pushl   x               # x is a 32-bit integer
                pushl   $format         # pointer to format string
                call    _printf         # the C printf routine
                addl    $8, %esp
                 . . .
                .data
x:                      .long           0x341256
format:         .asciz  “%d”
```

- Have to watch out for `%ecx` -- it too may get clobbered as the next example shows

Printing From Assembler

- **Assembly code to print Hello twice within a loop:**

```
            .globl  _dhello
            .text
_dhello:    movl    $2, %ecx        # set up count
doline:     pushl   %ecx            # save count
            pushl   $hellostr       # the string constant
            call    _printf         # clobbers eax, ecx, edx
            addl    $4, %esp        # restore stack
            popl    %ecx            # restore count
            loop    doline          # decr, test, and branch
            ret
            .data
hellostr:   .asciz  “Hello\n”
```

Printing From Assembler (Cont'd)

- The C calling routine (call it dhelloc.c according to our convention) to get things going is:

```
extern void dhello(void);
  void main()
    {
         dhello();
    }
```

Call/Return Use of Stack

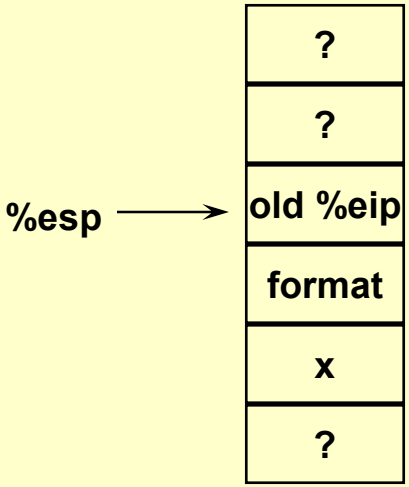
- Suppose the assembler code wants to be equivalent to the C code:

```
printf("%d", x);
```

  the actual assembler code would look like:

```
pushl  x
pushl  $format
call   _printf
```

- Then the stack would look like:
  - Which way is the stack growing?
  - Where was the %esp pointing to before the first pushl?

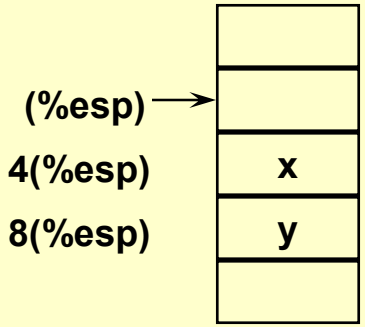


Call/Return Use of Stack (Cont'd)

- **In general, when you reach a function**
  - **The "old" `%eip` (i.e., the return address) is at the "top of the stack" and `%esp` holds its address**
  - **The first argument is next in memory at the address corresponding to the contents of `%esp+4`**
    - » **We would write this as `4(%esp)`**
  - **The next argument is ….**
- **Note that this last form of addressing is called "indirect"**
  - **It uses the contents of the register as an address and not data**
  - **We use the contents of the register to reach into memory for an operand**
  - **Examples: `movl $1,%edx` versus `movl $1,(%edx)`**

More Examples

- `z = add2(x, y)` where both x and y are integers
  - The caller pushs y and then x on the stack
  - Then add2 is called
  - The stack looks like
  - But why is y pushed first?



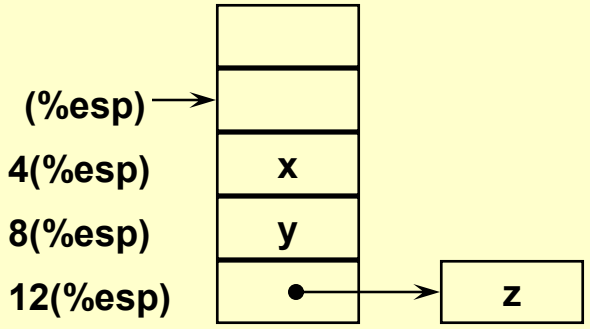
- What does add2 look like?

```
_add2:    movl   4(%esp), %eax # get x
          addl   8(%esp), %eax # add in y
          ret
```

More Examples (Cont'd)

- This time we pass an address when we call using `add3(x, y, &z)`
- The code looks like:

```
_add3:    movl   4(%esp), %eax
          addl   8(%esp), %eax
          movl   12(%esp), %edx
          movl   %eax, (%edx)
          ret
```



  - Note the use of the pointer in the last two move instructions
  - This is the dereferencing we speak about in C

Scanning Pointer Problem

- Want to sum up the elements in an array of N elements

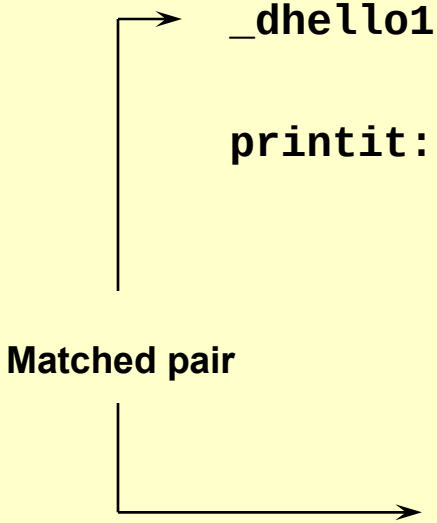
```
            .data
  iarray:   .long  1, 4, 9, 16, …
```

- The code might look like:

```
_sumarray:  movl    $0, %eax        # initial sum
            movl    $N, %ecx        # initial count
            movl    $iarray,%edx    # initial pointer value
add1:       addl    (%edx), %eax    # add in next element
            addl    $4,%edx         # bump pointer
            loop    add1            # test and loop
            ret
```

Loose Ends

- What if we want to use a “non-scratch” register such as %ebx?
- Recall our “hello” example now modified;




```
_dhello1:   pushl   %ebx            # save register contents
            movl    $2, %ebx        # put something into %ebx
printit:    pushl   $hellostr       # string pointer
            call    _printf         # go print it
            addl    $4, %esp        # restore stack
            decl    %ebx            # count down
            cmpl     $0, %ebx       # at end?
            jnz     printit         # if not, loop
            popl    %ebx            # restore %ebx
            ret
```

Stack Management

- How do you pass information to functions
  - In registers
  - In memory -- statically or in the heap (not on the SAPC since no malloc)
  - On the stack
- The stack is used to hold:
  - Return addresses
  - Parameters
  - Automatic data
- Since functions call other functions, there is a nesting of stack information called *stack frames*

Assembly Language I/O

- Have the instruction: **inb**

  which is used as follows: **inb %dx, %al**
- Also have: **outb**

  which is used as follows: **outb %al, %dx**
- When called from C, such as:

```
outpt(port, byte);
```

  how do we get the arguments?
- We use the stack:

```
movw 4(%esp),%dx    # port into dx
movb 8(%esp),%al    # byte into al
```

portio.s

```
.globl _outpt, _inpt
.text
# output byte to port
# call from C: outpt(port, byte)
_outpt:    movw 4(%esp),%dx          # port into dx
           movb 8(%esp),%al          # byte into al
           outb %al, %dx             # OUT instruction
           ret

# input byte from port
# call from C: byte = inpt(port)
_inpt:     xorl %eax,%eax            # clear eax
           movw 4(%esp),%dx          # port into dx
           inb %dx,%al               # IN instruction: byte into al
           ret                       # return with byte in al
```

Condition Codes or Flags

- Find in text on page 88
- ZF or zero flag is set when an arithmetic or logical operation produces a result of zero
  - Examples would be adding -1 and +1, or decrementing a variable
  - Note that moves, I/O, and push/pop instructions do not set this flag (unlike other machines)
  - Example:

```
cmpb   $'0',%al        # test if al holds a '0'
```

- SF or sign flag is set if an arithmetic or logical operation generates a result where the MSB is a 1
  - Examples would be decrementing a variable from 0 to -1, or adding a large negative number to a smaller positive number

Carry Flag - Addition

- CF or carry flag is set if an arithmetic operation produces a result that exceeds the capacity of a register
  - Set by a carry out of or a borrow into the high order bit
  - Examples (unsigned):

0xB0	$1011\ 0000_2$	176_{10}
0x60	$0110\ 0000_2$	96_{10}
1 0x10	$1\ 0001\ 0000_2$	272_{10}

Carry bit

- The carry flag is set if there is a carry out of the MSB of the arithmetic result (what is overflow?)
- Question, what would this mean if the numbers were signed?

More on the Carry Flag

- **Consider the following**

```
movb    $0xB0,%al
addb    $0x60,%al
```

  - **What result is produced in `%eax`?**
    - » **First instruction produces:** **`%eax` = 000000b0**
    - » **Second instruction produces:** **`%eax` = 00000010**

      plus the carry flag is set

- **This is out of bounds for unsigned numbers**
  - **What happens for signed numbers**
  - **Now we have to handle overflow**

Overflow -- Addition

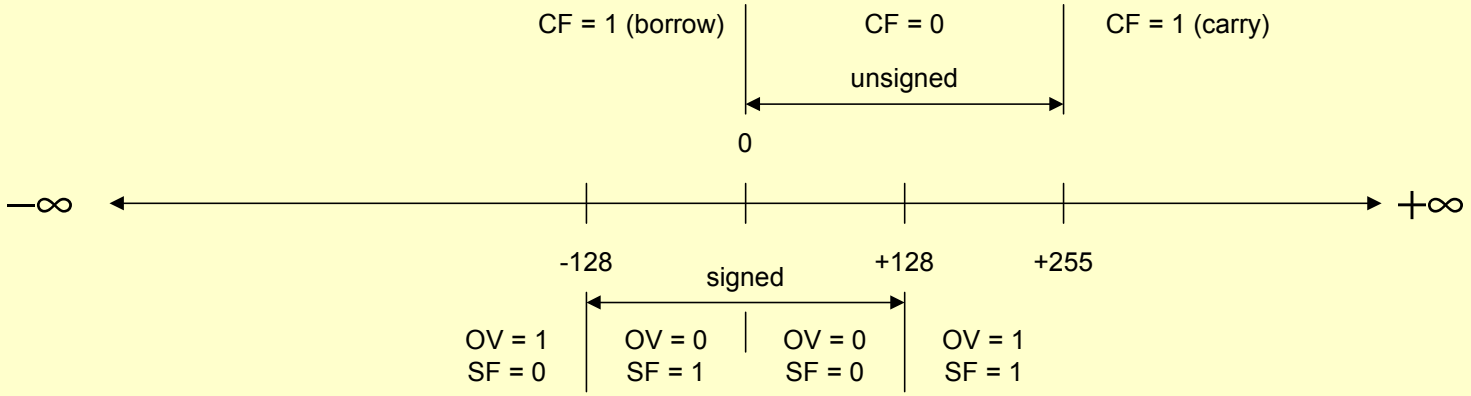
- Suppose we want to add 0x60 to itself?

0110 0000	0x60
0110 0000	0x60
1100 0000	0xC0

- If the arithmetic is unsigned, this is okay; but if it is signed, then the result is incorrect due to the carry into the MSB
- Overflow occurs whenever both operands are of the same sign and the result generated is of the opposite sign
  - In the example above for signed numbers, the overflow bit would be set and the carry bit would not be set

Compare Sets Flag Register

- **Normally, unsigned compare is defined as:**
  - **compare == if ZF =1**
    (subtract) < if ZF = 0 && CF =1
    > if ZF = 0 && CF = 0
- **Graphically, we can represent signed and unsigned numbers as:**



Compare Sets Flag Register (Cont'd)

- **So we can now say for a signed compare:**
  - **compare == if ZF =1**
    (subtract) < if ZF = 0 && OV !=SF
    > if SF = OV
- **An important point to remember:**
  - **Unsigned comparisons use ABOVE and BELOW**
  - **Signed comparison use GREATER and LESS**

Using Conditional Jumps

- Use the flags register but what actually happens depends on whether the operands are unsigned or signed
  - Examples:

```
cmpl    op2,op1          # op1 - op2 > 0
jg      label            # test if greater than
                         # tests ZF + (SF + OF) = 0
```

  - How about if Temp > 100?

```
cmpl    $100,temp        # note restriction on operands
jg      hitemp
```

More Conditional Jumps

- Want to bracket a value
  - if 10 ≤ X ≤ 20 then do_it
  - Let's assume that X is in %eax; the code would look like:

```
            cmpl   $10,%eax      # check lower bounds
            jl     skip          # skip if < 10
            cmpl   $20,%eax      # check upper bounds
            jg     skip          # skip if > 20
do_it:                           # do it
            …
skip:                            # do not do it
```

- Examine Table 3-6 in S&S

Addressing Modes

- **Displacement**
  - While Intel specifies it as: `mov 4[ebx],al`
  - We use gnu and it becomes: `movb %al,4(%ebx)`
  - Very useful for indexing through a range of values and is commonly found on RISC and CISC machines
- **Scaled**
  - Again, Intel specifies as: `mov array[4*edx],eax`
  - We use gnu and it becomes: `movl %eax, array(,%edx,4)`
- **Summarizing, we have:**
  - Register
  - Immediate
  - Direct
  - Register indirect
  - Register indirect with displacement
  - Scaled index

Using C Structs

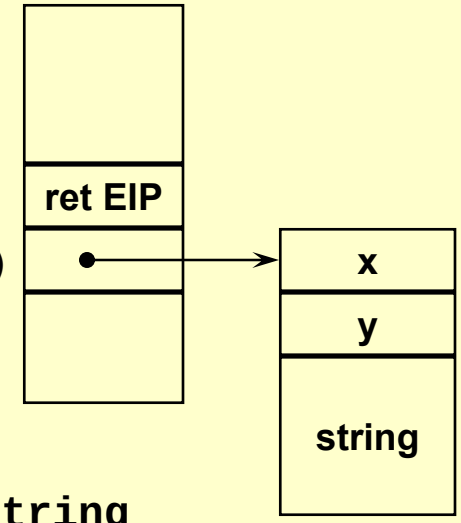
- How do we access a C structure such as:

```
#define NAMELEN 20
struct teststruct {
        int x, y;
        char name[NAMELEN];
};

struct teststruct t;
t.x = 2; t.y = 5;
strcpy(t.name, “eckhouse”);
trystruct(&t);       /* param passed to asm
  via pointer */
```

Using C Structs (Cont'd)

- Assembly code would look like:

```
movl      4(%esp),%edx    # ptr to t
movl      (%edx),%eax     # x itself
  . . .
movl      4(%edx),%eax    # y itself
```



- If we want the string pointer we need to:

```
movl      8(%edx),%edx    # ptr to named string
movb      (%edx),%al      # first char in string
```

- Other fancier modes but we can live without them

Serial Communications (RS232)

- Allows us access to more of the “real” hardware
  - Able to detect and measure what is happening
  - Regardless of 9- or 25-pin connectors (DB9 or DB25) have 8 pins carrying information
    - » Two data pins for sending and receiving
    - » One ground pin
    - » Five control pins
- See page 647 of S&S for the 8 data, status, and control ports
  - Notice that some ports provide a dual function depending on a read or write operation
  - See serial.h for the C definitions of the ports and offsets

COM1 Port

- The first port is addressed at 3F8, but we can use `COM1_BASE+UART_RX (or ...UART_TX)`
- Must be able to determine if a character is available
- Use `UART_LSR` to access status bits:
  - `UART_LSR_DR for data ready bit`
  - `UART_LSR_THRE for transmit-hold-register-ready bit`
- See serial.h for all the other bits

Polling the COM Port

- **Can be done in C**

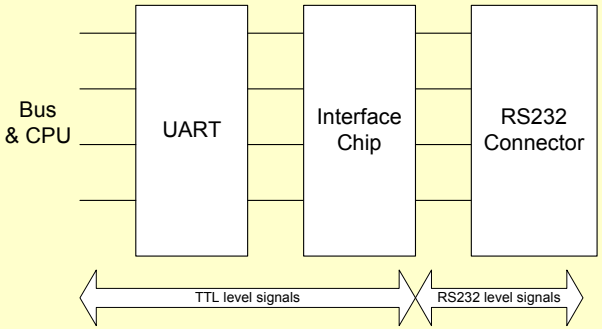
```
#include <serial.h>

void pollputc(unsigned char ch) {
        while ((inpt(COM1_BASE + UART_LSR) & UART_LSR_THRE) == 0)
                ; /* polling loop, waiting for THRE bit to go on */
        outpt(COM1_BASE + UART_TX, ch);
}
```

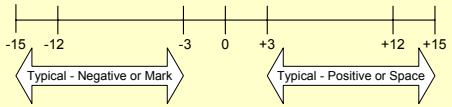
and assembly language as well

- **But in Gas we use inb and outb**

Serial Communication Signal

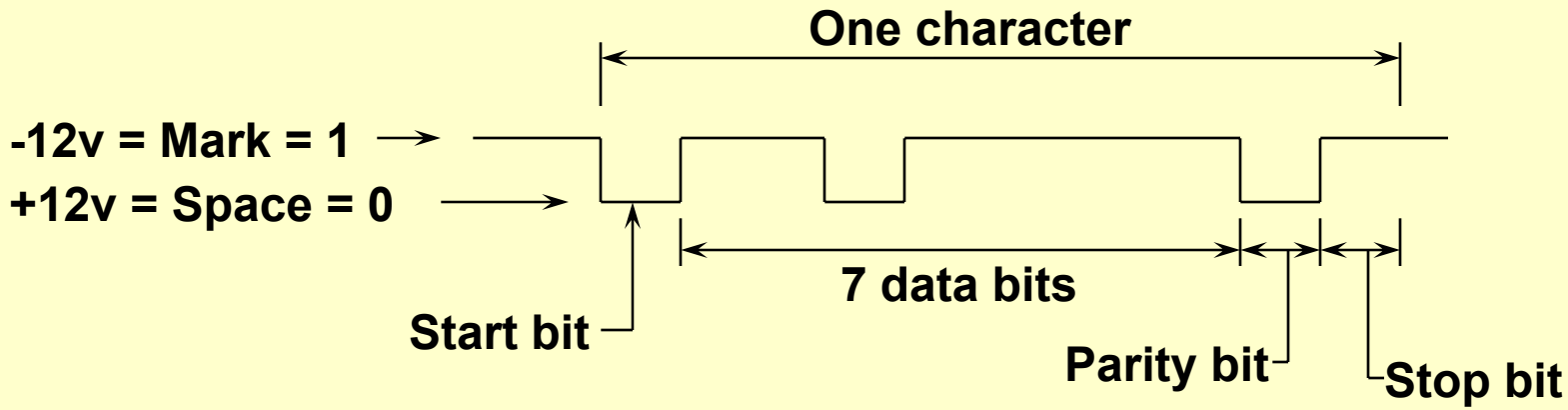
- Nominally TTL is 5 volt logic; this is true inside the PC and includes the UART
- RS232 is <u>not</u> 5 volt logic; it nominally ranges from -15 volts to +15 volts





- Besides the data lines there are the modem control lines called RTS, CTS, DSR, DTE, and CD

Defining the Serial Interface

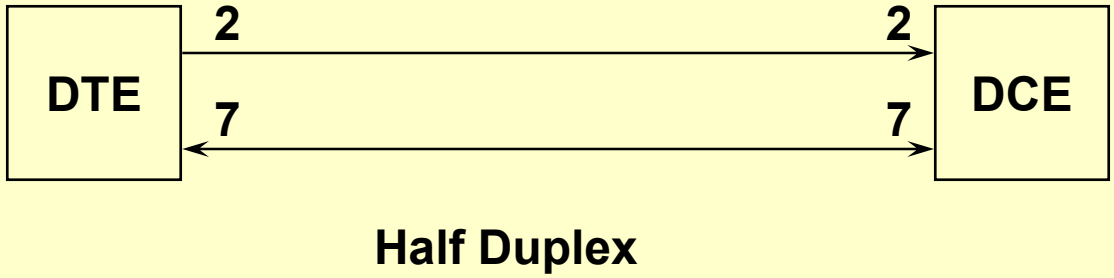


- Need to also describe the RS232 physical layer protocol
  - Data communications equipment (DCE)
  - Data terminal equipment (DTE)

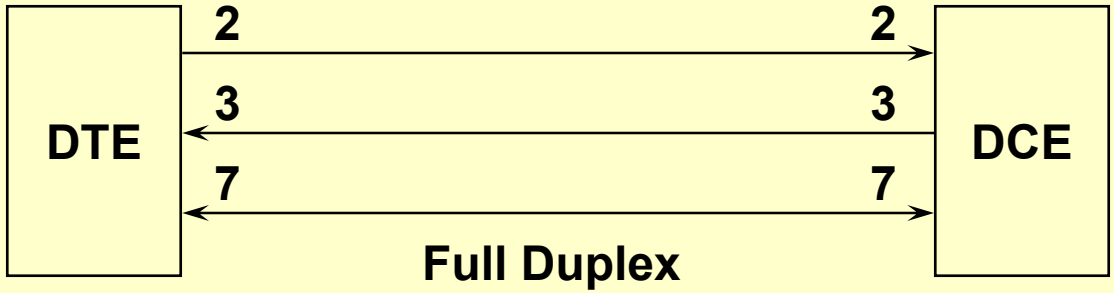
The RS232C Physical Layer

- Originally created as a standard between computer equipment and modems
- Specifies the plug/socket connections, the transmission path, and the signals (control and data)
- Modem is the DCE (Data Communications Equipment)
- Digital equipment connected to modem is DTE (Data Terminal Equipment)

RS232C Control Lines



Half Duplex



Full Duplex

Connecting RS232C Devices

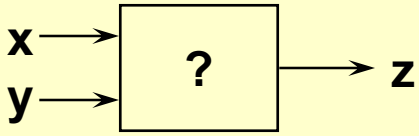
- Can you connect two DTEs together?
- Can you connect two DCEs together?
- Control and data signals for the 25-pin DTE:
  - Protective ground (pin 1)
  - Transmitted data (pin 2 - outgoing)
  - Received data (pin 3 - incoming)
  - Request to send (pin 4 - outgoing)
  - Clear to send (pin 5 - incoming)
  - Data set ready (pin 6 - incoming)
  - Signal ground (pin 7)
  - Carrier detect (pin 8 -incoming)
  - Data terminal ready (pin 20 - outgoing)

Flow Control

- **Both hardware and software flow control**
  - **SW uses X-ON and X-OFF**
  - **HW uses DTR/DSR and RTS/CTS but that only works between electrically connected systems; DCD needed over PSTNs**
- **How do you decide who is the transmitter?**
  - **Check for a voltage more negative than -3v on pins 2 and 3 since transmitter sends a marking signal when idle**
- **How do you decide who is the originating modem and who is the answering modem?**
  - **Example from old Bell 103A**
    - » **Originator uses: Space at 1070 cps, Mark at 1270 cps**
    - » **Receiver uses: Space at 2025 cps, Mark at 2225 cps**

Basic Logic Gates

- For a two input, one output gate, how many possibilities are there?

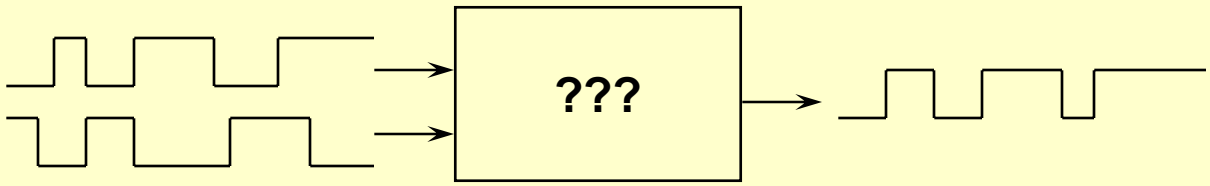


x	y	z									
0	0	0	0	0	0	0	0	0	0	...	1
0	1	0	0	0	0	1	1	1	1	...	1
1	0	0	0	1	1	0	0	1	1	...	1
1	1	0	1	0	1	0	1	0	1	...	1

- Assume a basic understanding of AND, OR, NOT, XOR and the negated forms: NAND and NOR

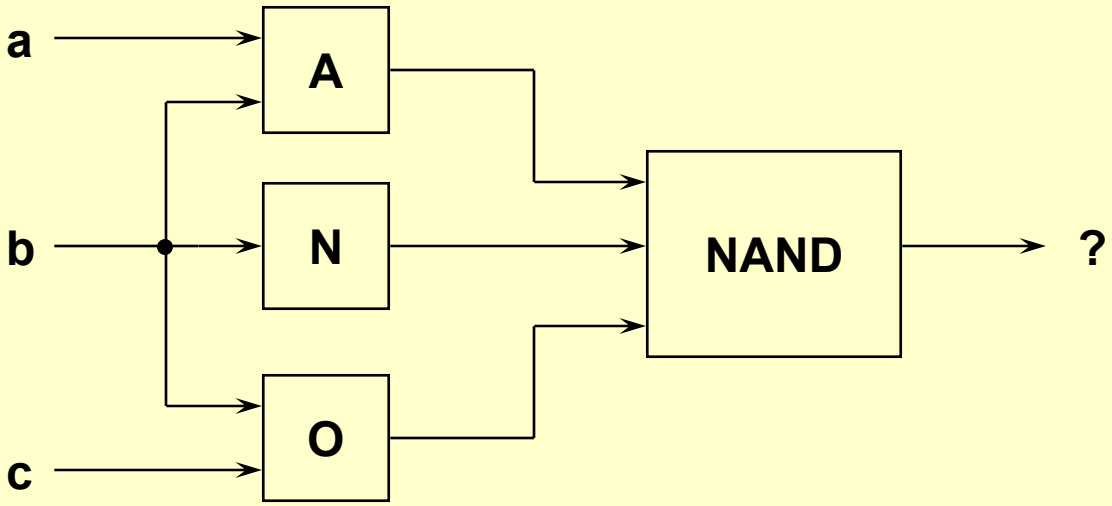
Basic Logic Gates

- Are there any limits to the number of inputs?
- What about limits on outputs?
- Can you input pulse trains rather than constant inputs?



What About Combinations?

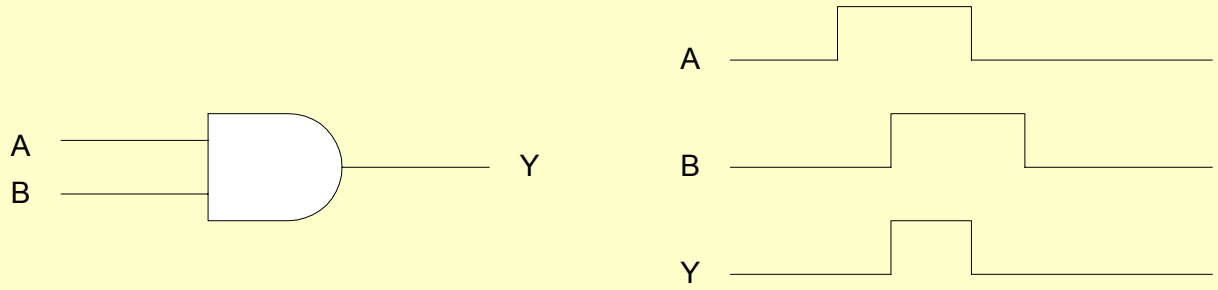
- What is the output of this circuit?



- Can you make the truth table for this circuit?

Timing Considerations

- **Things don't happen instantaneously**
  - **Signals arrive at some time**
  - **The logic takes some time to react**
  - **The output appears some time after the inputs**
- **Example, an AND gate**

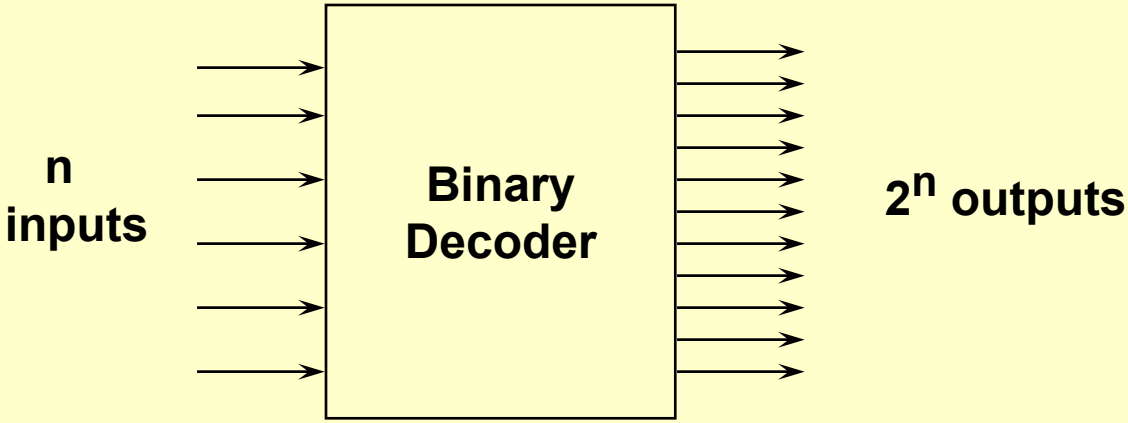


Combining the Basic Logic Gates

- Decoders
- Encoders
- Selectors - Multiplexers
- ALUs
- Control Units
- Buses
- Simple computers

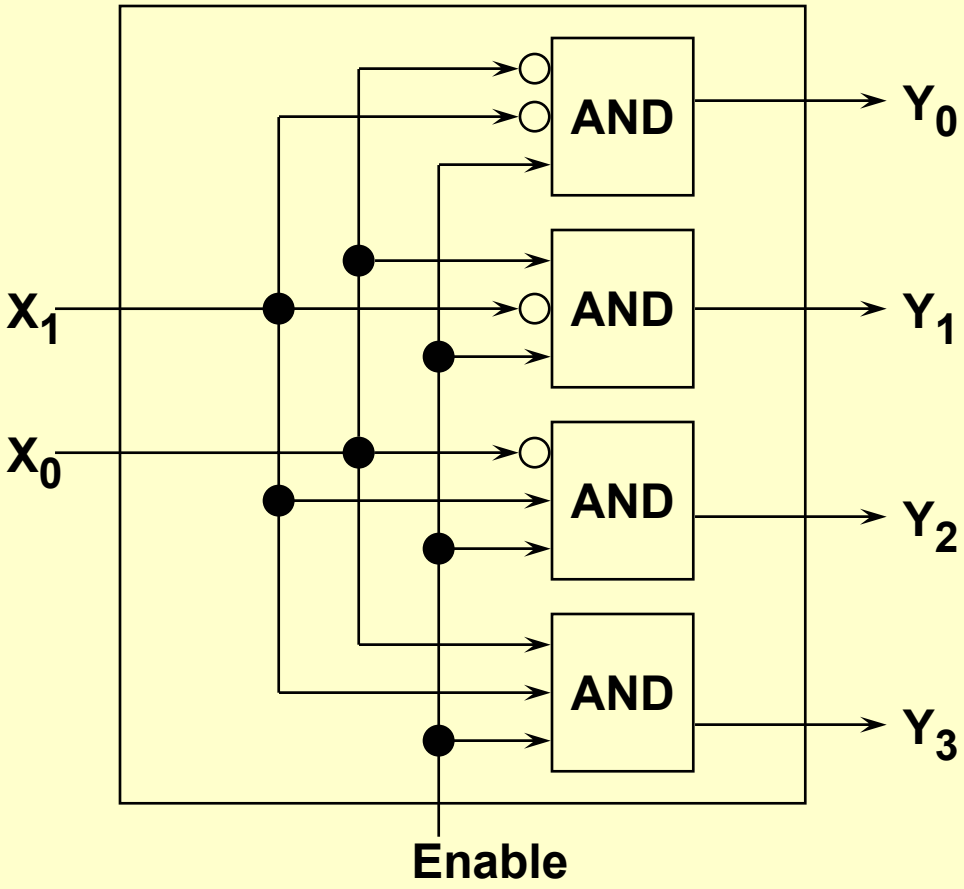
Binary Decoder

- Black box with n input lines and 2^n output lines
- Only one output is a 1 for any given input

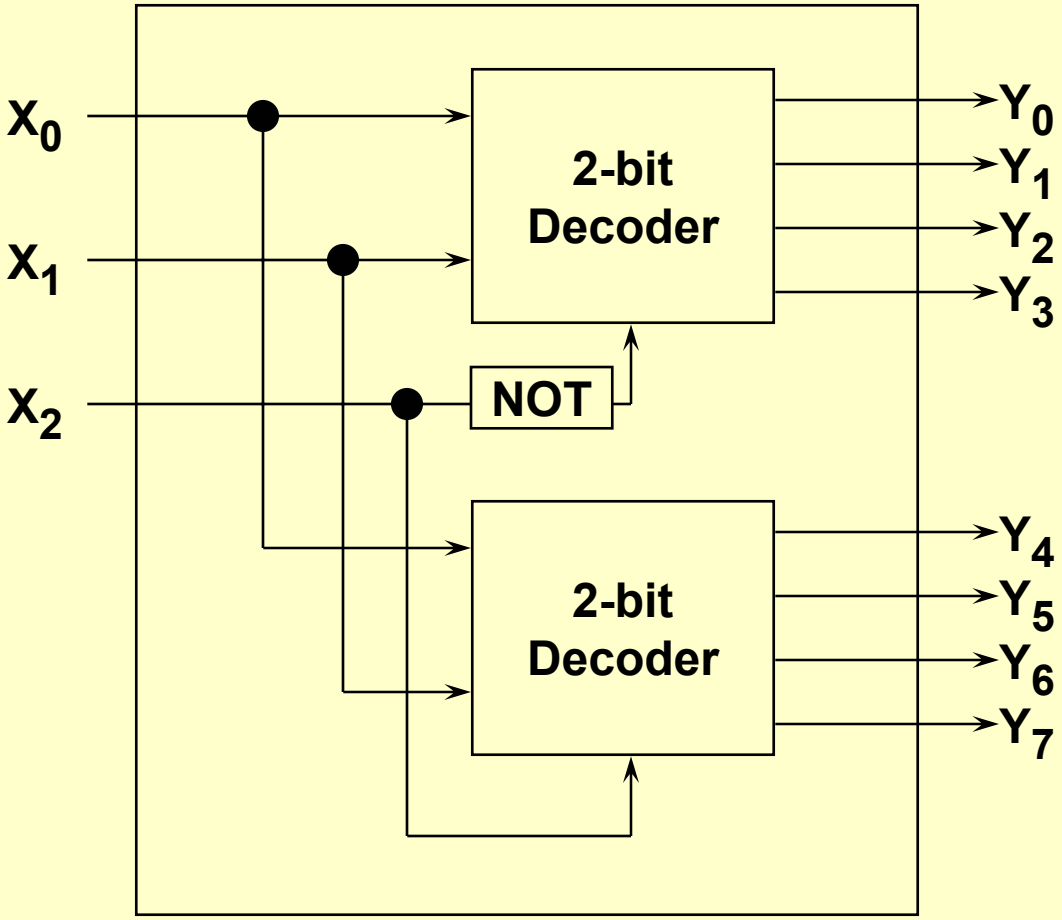


Building a Binary Decoder

- Start with a 2-bit decoder:

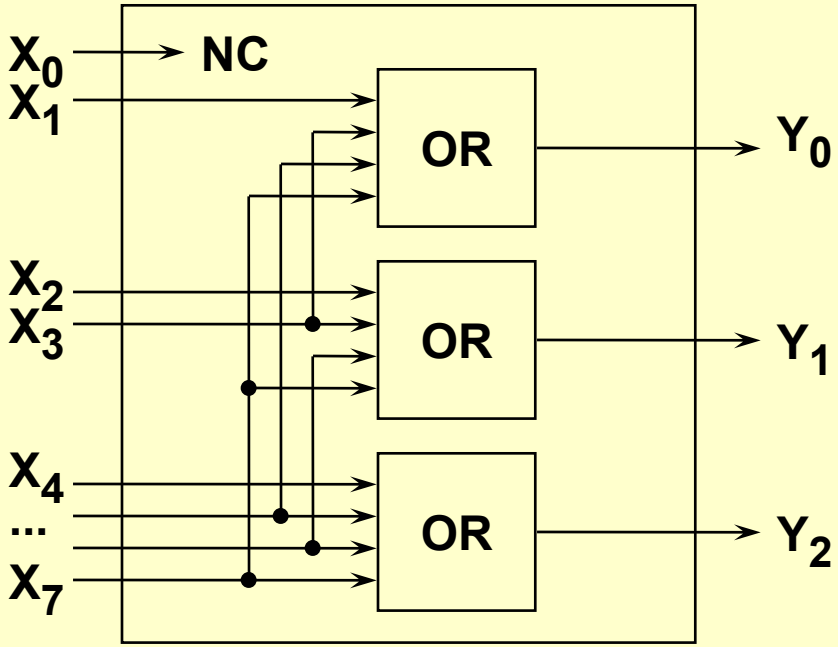


Then Add Two to Make Three...



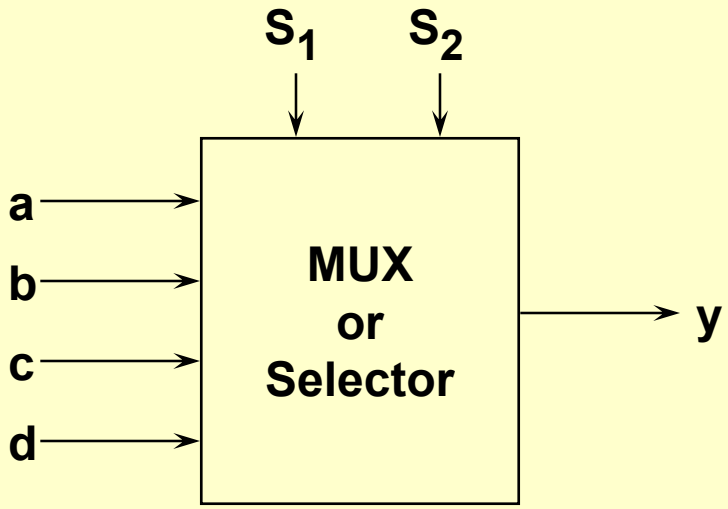
Developing an Encoder

- If we can decode, then we need to encode
- Want to encode from 1 out of n into a binary weighted form
- A keyboard encoder does this



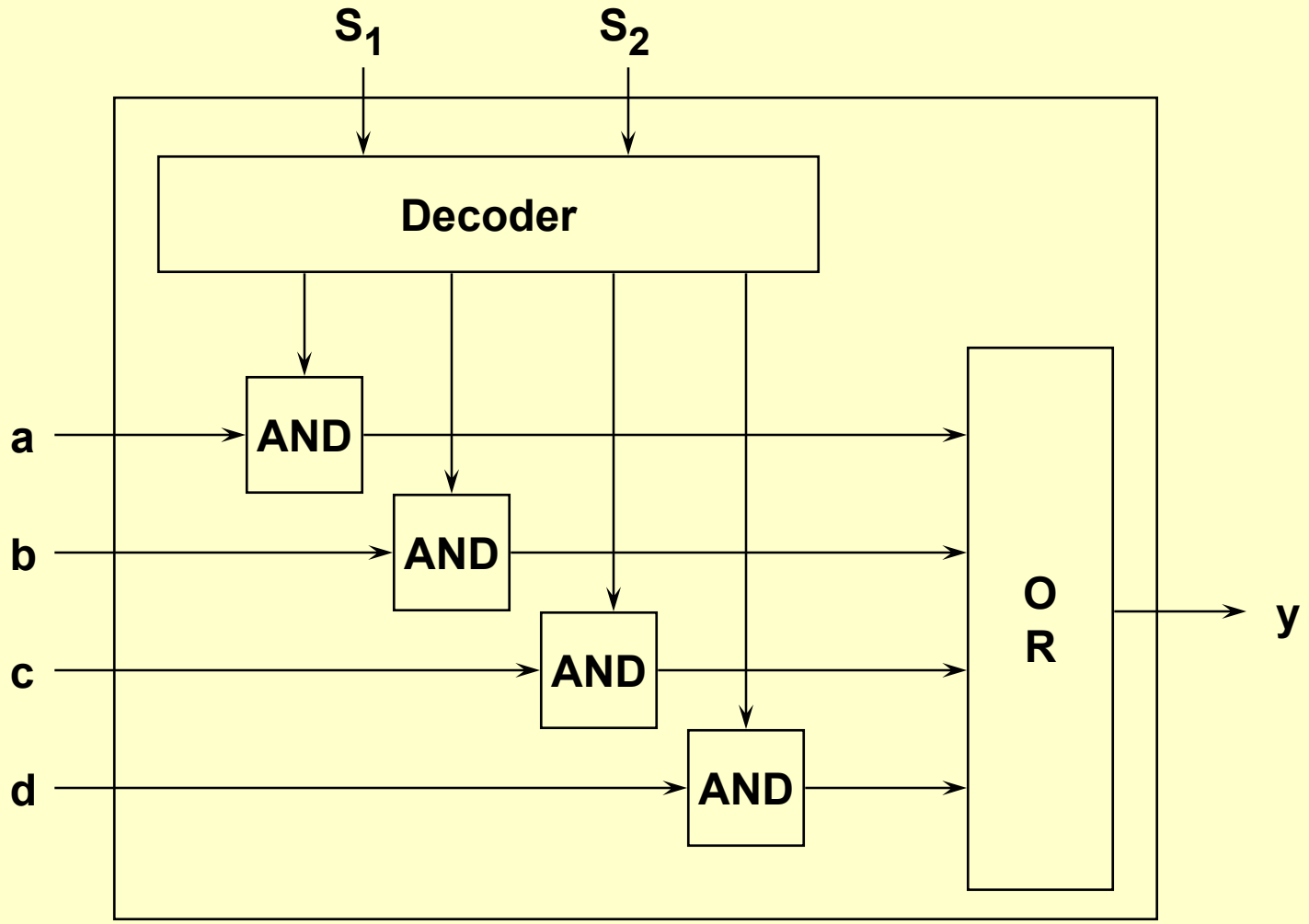
Next Comes a Selector

- Like a switch; also called a multiplexer or MUX

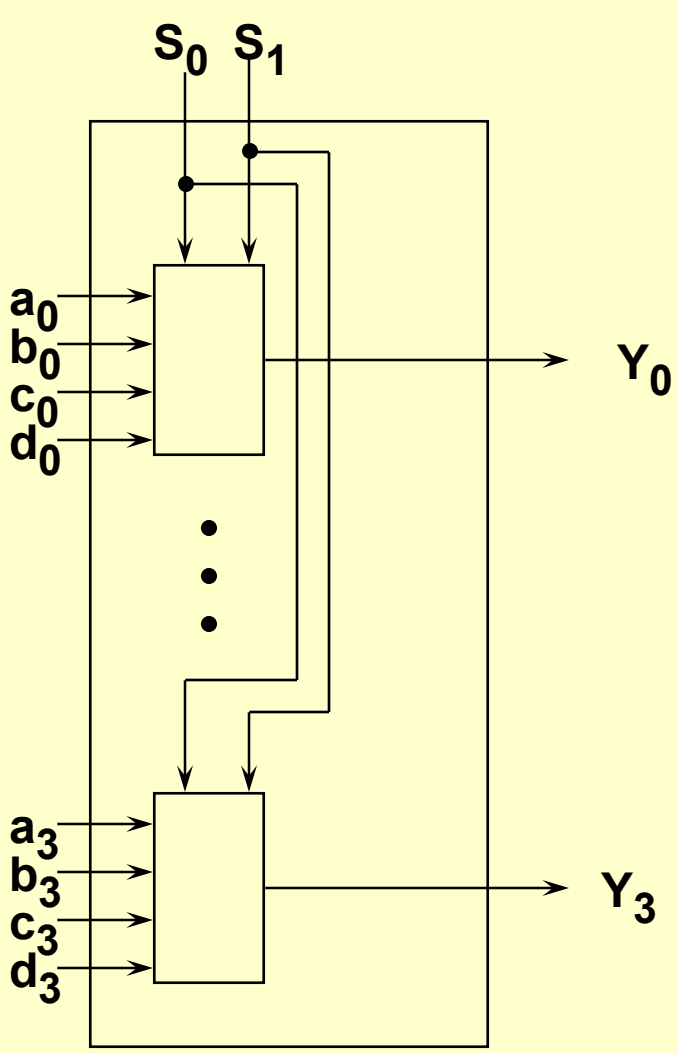


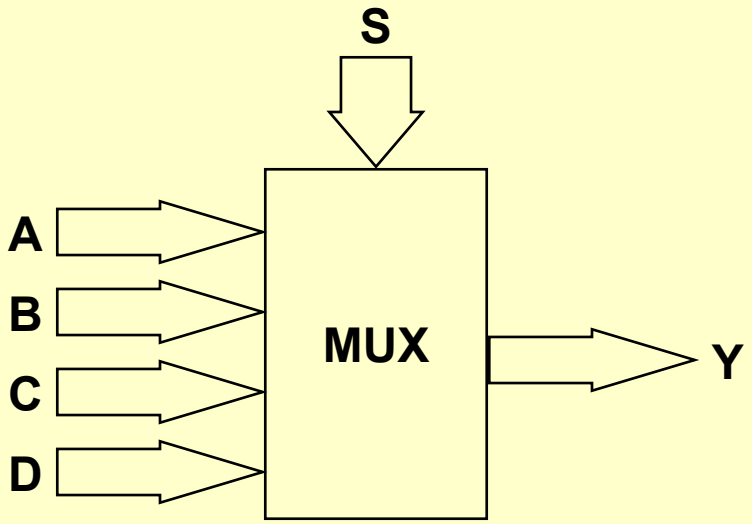
- Again, built it up from a simple base element

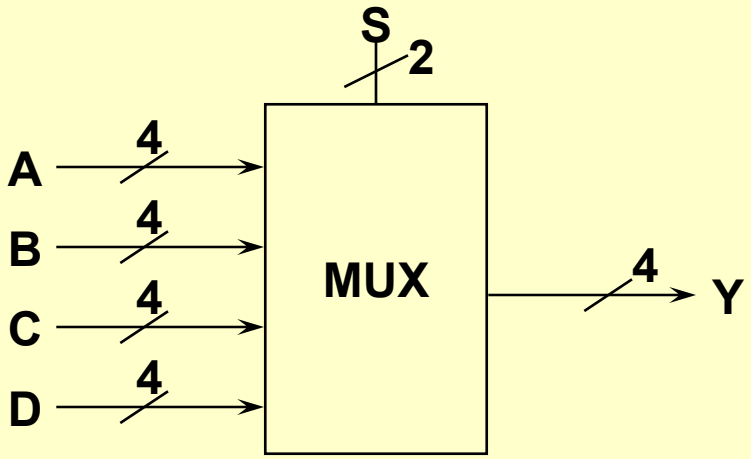
A 1-bit Selector



A 4-bit Selector

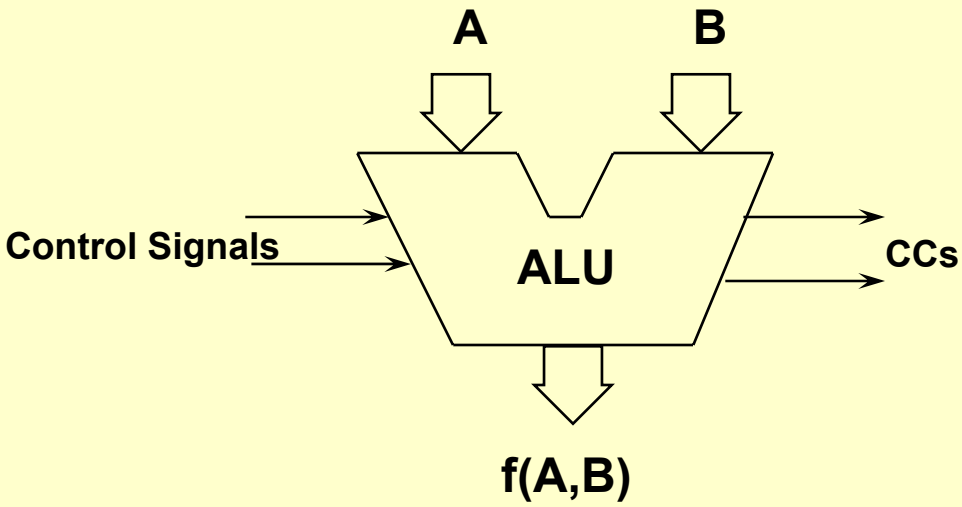






The ALU Is Next

- Logical and arithmetic operations
- Variations in
  - Base
    - » Binary
    - » Decimal
    - » BCD
  - Implementation
    - » Serial
    - » Parallel
    - » Pipelined



Other ALU Design Issues

- Operand interpretation
  - Integer
  - Multiple precision
  - Floating-point
- Operand representation
  - Sign-magnitude
  - Complement
  - Bias exponent
- Mathematical consistency
  - Under/over-flow
  - Exceptional values
  - Anomalies

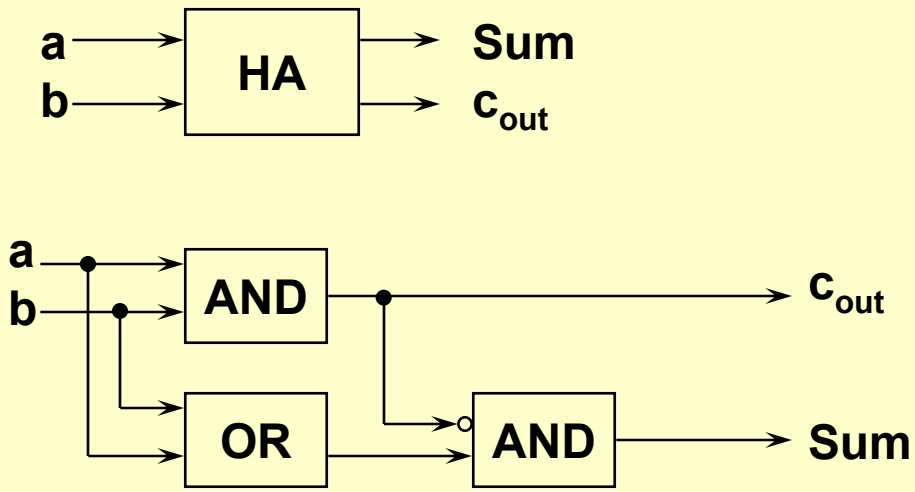
Simple Example - Serial by Bit Adder

- Develop a half-adder (HA)
- Use HA to build a full-adder (FA)
- Of course, this is not all there is to it

The Half-Adder

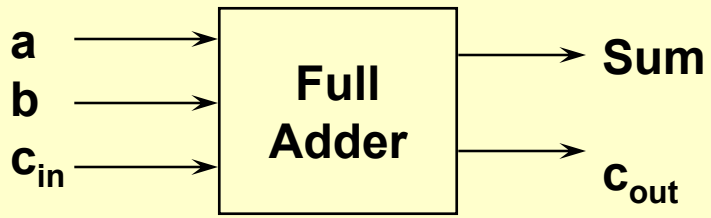
$$\text{Sum} = (\overline{a} \bullet b) + (a \bullet \overline{b}) = (\overline{a} + \overline{b}) \bullet (a + b)$$

$$\text{Carry} = a \bullet b$$

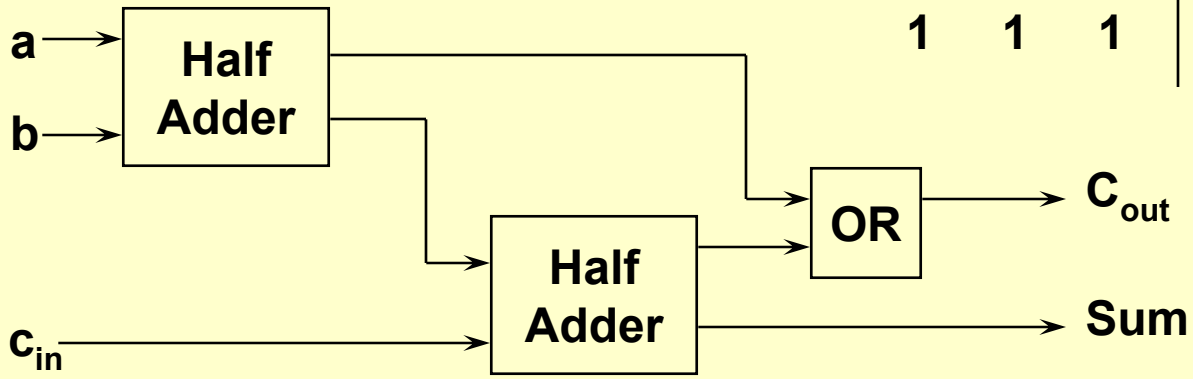


a	b	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

From HA to FA

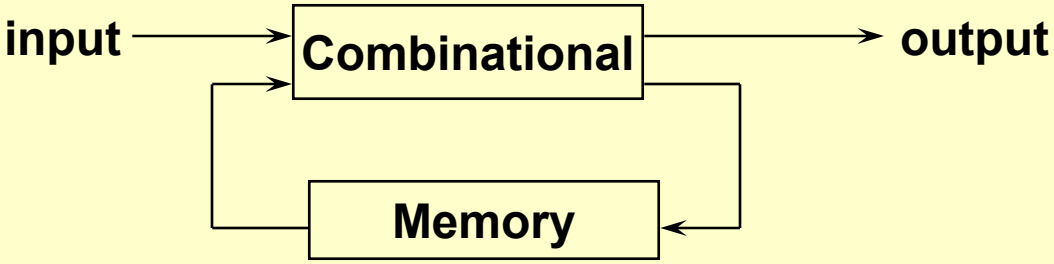


a	b	c_{in}	Sum	C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1



Practical Logic Gates

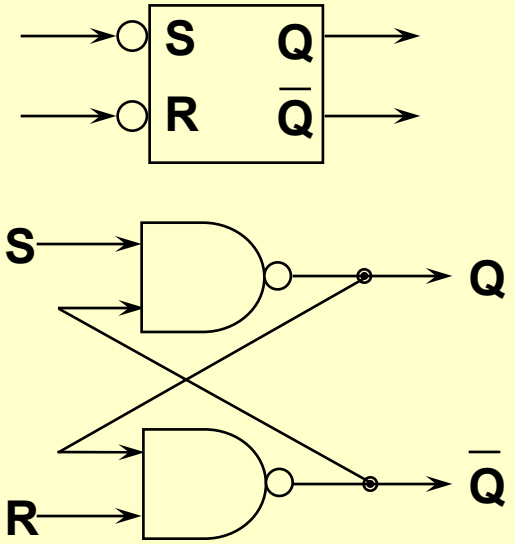
- **Transistor-Transistor-Logic (TTL) is generally in 14- (or 16-pin) package with V_{cc} on pin 14 (or pin 16) and GND on pin 7 (or 8) *BUT NOT ALWAYS***
- **Can build *combinational* or *sequential* circuits, with sequential being a combinational circuit with memory**



Simple Memories (Flip-Flops)

- Many different circuits; simplest is R-S
- Note the complemented inputs
- Can buy a standard TTL R-S flip-flop (279)

S	R	Q	$\overline{Q}$
0	1	1	0
1	0	0	1
1	1	no change	
0	0	prohibited	

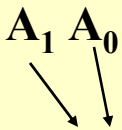


Programmable Interval Timer (PIT)

- Basic idea is to count down from some preset value (see Sec. 7-5 in S&S)
- Sixteen bit count at a rate of 1.193 MHz. (approximately every microsecond)
- Full downcount = 64K/1.193 x 10^6 = 1/18.2 sec or 55 milliseconds
- Can use this downcount to time to 55 millisecond accuracy by reading downcount value
- Examine Intel 8254 datasheet: find three channels on the chip so three timer/counter circuits (http://www.intel.com/design/periphrl/datashts/231244.htm)

PIT Characteristics

- **PIT chip has four I/O ports assigned to it:**



  - **Timer 0 assigned port 40 = 0100 0000**
  - **Timer 1 assigned port 41 = 0100 0001**
  - **Timer 2 assigned port 42 = 0100 0010**
  - **Control assigned port 43 = 0100 0011**
  - **Chip selected by "$\overline{\text{chip select}}$" and A_1-A_0**
  - **Other signals include $\overline{\text{read}}$, $\overline{\text{write}}$, and data**

Control Word Format

- Actually only a byte:

SC1	SC0	RW1	RW0	M2	M1	M0	BCD

- SC1-SC0 select which counter to write/read
- RW1-RW0 to latch value or select which byte of count value
- M2-M0 determines which operating mode
- BCD specifies whether binary or BCD count
- Command formats found in datasheet

Using the PIT in C

- Refer to timer.h

```
#define TIMER0_COUNT_PORT 0X40
#define TIMER_CNTRL_PORT 0X43
/* bits 6-7: */
  #define TIMER0 (O<<6)
  #define TIMER1 (1<<6)
/* Bits 4-5 */
  #define TIMER_LATCH (0<<4)
  #define TIMER_SET_ALL (3<<4)
/* Bits 1-3 */
  #define TIMER_MODE_RATEGEN (2<<1)
/* Bit 0 */
  #define TIMER_BINARY_COUNTER 0
```

Programming the PIT

- **Bits to initialize**

```
TIMER0 | TIMER_SET_ALL | TIMER_RATEGEN
  |TIMER_BINARY_COUNTER
```

- **Output to the timer I/O port**

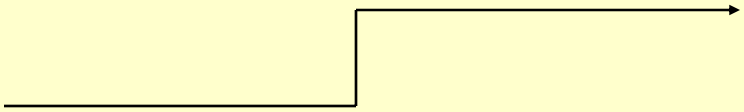
```
outpt(TIMER_CNTRL_PORT, …);
```

- **Then load the downcount**

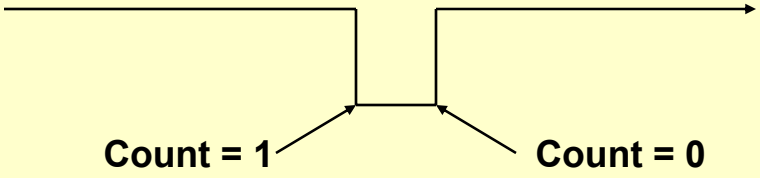
```
outpt(TIMER0_COUNT_PORT, count & 0xFF);     //
  LSByte
outpt(TIMER0_COUNT_PORT, count >> 8);       //
  MSByte
```

What Are the PIT Modes?

- **Mode 0: Count value loaded and countdown occurs on every clock signal; Out from counter remains low until count reaches 0 when it goes high**

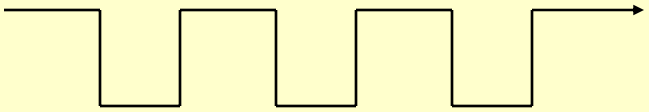


- **Mode 2: Counts down from loaded value; when count has decremented to 1, OUT goes low for one clock pulse and then goes high again; count is reloaded and process repeats**



What Are the PIT Modes? (Cont'd)

- **Mode 3: Functions as a divide by n square wave generator, where n is the count value; OUT starts high and alternates between low and high.**



Reading the Count Values

- Want to read the count value without disturbing the count in progress
- Have to consider that the counter is changing while we are attempting to read it
- Best way to read the count is to use the counter latch command to temporarily latch the count

```
outpt(TIMER_CNTRL_PORT, TIMER0 | TIMER_LATCH);
count = inpt(TIMER0_COUNT_PORT);
count |= (inpt(TIMER0_COUNT_PORT) << 8);
```

- Note that reading the count lets the latch again follow the count

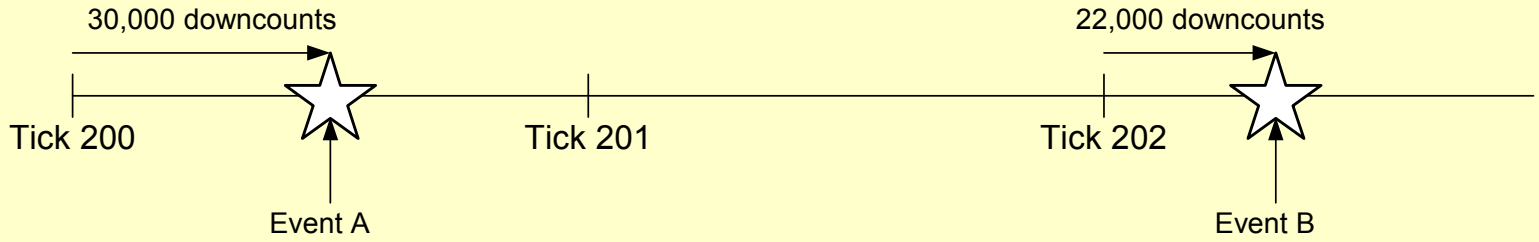
Converting Counts to Real Time

- Can count from 1 to 64K, that is from approximately one microsecond to 55 milliseconds
- What if we want something longer than that?
  - Have to perform repeated counts
    - » Example: 200 milliseconds = 3 * 55 +35
    - » Would need three full counts (called *ticks*) plus a partial count (called *downcounts*)
  - But how do you know when a tick has occurred?
    - » Could poll the device
    - » Better to use an interrupt
  - If interrupt occurs on every tick, which is counted, then the elapsed time in microseconds is approximately:
    - » [#ticks * 65536 + (startcount - stopcount)]/1.193

Measuring the Time Interval

- **Given:**
  - **Event A happened 30000 downcounts after tick 200**
  - **Event B happened 22000 downcounts after tick 202**

  how much time has elapsed?
- **Graphically**



  - **Elapsed time = 202 ticks + 22000 downcounts - (200 ticks + 30000 downcounts)**
  - **And tick = 64*1024 downcounts**

Interrupts for PCs in Protected Mode

- What is an interrupt?
- What does an interrupt do to the “flow of control”
- Priority levels
- Vectoring of interrupts
  - None
  - Vectored
- Interrupts used to overlap computation & I/O
  - Examples would be console I/O, printer output, and disk accesses
- Normally handled by the OS so under UNIX and NT, rarely coded by ordinary programmers
  - In embedded systems and real-time systems, part of the normal programming work

Interrupts (Cont'd)

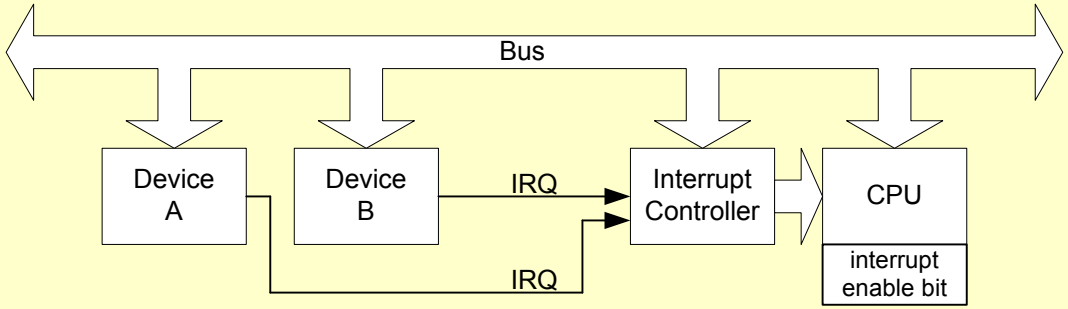
- **Why interrupts over polling? Because polling**
  - **Limits the CPU to one activity**
  - **Uses cycles that could be used more effectively**
  - **Code can't be any faster than the tightest polling loop**
- **Bottom line: an interrupt is an asynchronous subroutine call (triggered by a hardware event) that saves both the return address and the system status**

When an Interrupt Occurs

- Finish the current instruction
- Save minimal state information on stack
- Transfer to the interrupt handler, also known as the interrupt service routine (ISR)
- But there is more to it than this...
  - How do we know which device interrupted?
- And what happens if two (or more) devices request an interrupt at the same time?

Interrupts

- Complex hardware setup
- Needed for multitasking/multiprogramming OS



- Devices use IRQs to signal interrupt controller

Interrupt Controller

- On the PC known as the PIC which stands for "Programmable Interrupt Controller"
- Programmable means it has multiple possible behaviors selectable by software (via its own I/O ports)
- Devices send IRQ signals to interrupt controller
- Interrupt controller prioritizes signals, sending highest to CPU

CPU Interrupt Handling

- Enabling/disabling interrupts in the CPU
  - `sti` and `cli` instructions set and clear IF in EFLAGS
- CPU checks for interrupts between instructions if interrupts enabled (IF = 1)
  - Must save CPU state
  - Get ID ("nn") of interrupting device from interrupt controller
  - Uses nn to look up address of interrupt handler (ISR)
  - CPU enters kernel mode with IF=0
- ISR services the interrupt, including resetting the interrupt controller; ends with a special instruction "iret" on x86 to restore previously saved state and resume from point of interrupt

Interrupt Controller Details

- **Each device has an IRQ number based on its wiring to the PIC**
  - **Ex. COM2 uses IRQ3, timer 0 uses IRQ0**
- **PIC: the 8259A chip**
  - **Supports eight interrupt request (IRQ) lines**
  - **Two chips used in PC, called "master" and "slave"**
  - **Priority: highest to lowest order is IRQ0-1, IRQ8-15, IRQ3-7**
  - **Asserts INTR to CPU, responds to resulting INTA# with an 8-bit interrupt type code ("nn") on the data bus**

Interrupt Controller Programming

- PIC is accessible at port addresses 0x20 and 0x21 (for master), using “initialization command words” (ICWs) and “operational command words” (OCWs)
- ICWs used to set such things as:
  - How much to add to the IRQ to produce nn (8 used for DOS, 0x20 for Linux, 0x50 for Windows)
  - We trust the (Linux) bootup code to handle this setup
- OCWs used for:
  - EOI command: Reset interrupt in PIC after accepted by ISR (outb of 0x20 to port 0x20, for master)
  - Get/Set Interrupt Mask Register (port 0x21 for master)
    - » Ex: 0111 1110 = 0x7e enables IRQs 0 and 7, disables 2-6

Interrupt Process

- Requesting device generates a signal on IRQn
- PIC checks its interrupt mask before putting out a logic high on INTR line
- Between instructions, and if IF=1, the CPU sees INTR and initiates its *interrupt cycle*
- The interrupt handler (ISR) executes
- Requesting device is usually accessed in the ISR and is thus notified of the completion of the event
  - Ex: UART receiver detects inb for received char

CPU's Interrupt Cycle

- CPU detects INTR between instructions with IF=1
- CPU sends back a low on INTA#
- PIC responds by setting INTR low and puts out 8-bit interrupt code, *nn*, on data lines
- CPU reads nn and executes `int nn` instruction:
  - Machine state saved on stack (cs:eip and eflags)
  - IF set to zero
  - Accesses IDT[nn] to obtain ISR address
  - ISR address loaded in eip
    - » Change in eip causes the interrupt handler to execute next

Interrupt Handler Details

- **ISR must:**
  - **Save all registers used**
  - **Issue EOI command (end-of-interrupt) to PIC**
  - **Service the device, i.e., do whatever processing is needed for the event the device was signaling**
    - » **Ex. Read the received character, for UART receiver int's**
  - **Restore registers**
  - **Finish with iret instruction**

PIT Device (Timer 0)

- Simplest device: always is interrupting, every time it down counts to zero
- Can't disable interrupts in this device! Can mask them off in the PIC
- We can control how often it interrupts
- Timer doesn't keep track of interrupts in progress—just keeps sending them in
- So we don't need to interact with it in the ISR (but we do need to send an EOI to the PIC)

Timer Interrupt Software

- **`Initialization`**
  - **Disallow interrupts in CPU (`cli`)**
    - » **Unmask IRQ0 in the PIC by ensuring bit 0 is 0 in the Interrupt Mask Register accessible via port 0x21**
    - » **Set up interrupt gate descriptor in IDT, using irq0inthand**
    - » **Set up timer downcount to determine tick interval**
  - **Allow interrupts (`sti`)**
- **`Shutdown`**
  - **Disallow interrupts (`cli`)**
    - » **Disallow timer interrupts by masking IRQ0 in the PIC by making bit 0 be 1 in the Mask Register (port 0x21)**
  - **Allow interrupts (`sti`)**

Timer Interrupts: Two Parts to the Interrupt Handler

- **`irq0inthand` – the outer assembly language interrupt handler**
  - **Save registers**
  - **Calls C function irq0inthandc**
  - **Restore registers**
  - **Iret**
- **`irq0inthandc` - the C interrupt handler**
  - **Issues EOI**
  - **Increase the tick count, or whatever is wanted**

UART Interrupts

- The UART is a real I/O device, more typical of interrupt sources than timer 0
- The UART has four ways to interrupt; we'll study just receiver interrupts
- No interrupts are enabled until we command the UART to enable them, via register 1, the IER (i.e., port 0x3f8 + 1 or port 0x2f8 + 1)

UART Receiver Interrupts

- The receiver interrupts each time it receives a char, and remembers the interrupt-in-progress
- COM1 is connected to pin IR4 on the PIC, so its IRQ is 4 Similarly COM2's is 3
- The nn code generated by the PIC for COM1 is 0x24, so its interrupt gate descriptor is IDT[0x24]
- The ISR must read in the received char to satisfy the UART, even if no one wants the char. It also must send an EOI command to the PIC
- The receiver detects the inb for the char, and this completes the interrupt-in-progress

UART Interrupts (COM1)

- **`Initialization`**
  - **Disallow interrupts in CPU (`cli`)**
    - » **Enable interrupts in the UART (outb to port 0x3f9, IER)**
    - » **Unmask IRQ4 in the PIC by ensuring bit 4 is 0 in the Interrupt Mask Register accessible via port 0x21**
    - » **Set up interrupt gate descriptor in IDT, using irq4inthand**
  - **Allow interrupts (`sti`)**
- **`Shutdown`**
  - **Disallow interrupts (`cli`)**
    - » **Disable interrupts in the UART**
    - » **Disallow COM1 interrupts by masking IRQ4 in the PIC by making bit 4 be 1 in the Mask Register (port 0x21)**
  - **Allow interrupts (`sti`)**

UART (COM1) Interrupts: Two Parts of the Interrupt Handler

- **`irq4inthand` – the outer assembly language interrupt handler**
  - **Save registers**
  - **Call C function irq4inthandc**
  - **Restore registers**
  - **Iret**
- **`irq4inthandc` - the C interrupt handler**
  - **Issue the EOI command to the PIC**
  - **Input the char, and whatever else is wanted**

Something Called Exceptions

- Deviation from the normal condition
- While interrupts are asynchronous events initiated by the hardware ...

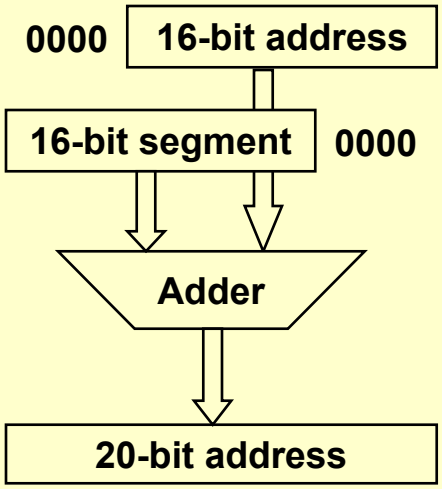
  exceptions are synchronous and initiated by both hardware and software
- If initiated by software they are also known as software interrupts
- Either way, both are detected by the computer hardware

What Causes An Exception?

- Certain types of instructions cause exceptions:
  - As the result of an abnormal condition
    - » Overflow, underflow, page fault, ...
  - As a result of executing a specific exception causing instruction (e.g., trace, trap, and emulator)
    - » Single stepping through a program is possible by setting the trap flag bit in the flags register
    - » Emulator trap for unimplemented instructions or auxiliary processing (e.g., Vector or FP ops)
    - » Traps for system calls that result in a change of the machine state

Addressing on the PC

- Without memory management in place, the x86 uses 16-bit addresses (referencing all of 64K!)
- To access the full 20-bit address, a segment register is used (notation is *segment:offset*)



Example:

address = 0x1234
seg reg = 0x5678

```
  0x01234
+0x56780
---------
```

20-bit result = 0x579B4

DMA Controller

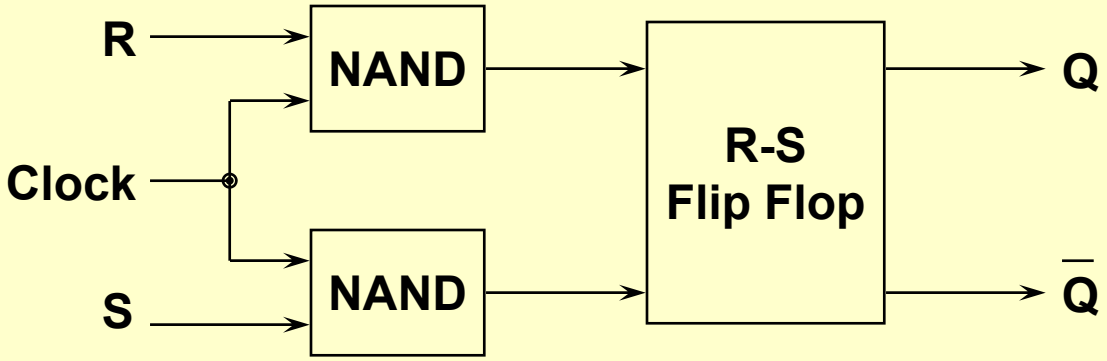
- Need based on moving data from one place to another (e.g., I/O and memory) without tying up CPU
- Specialized microcontroller that takes over CPU's bus (control, address, and data lines)
  - Without DMA
    - » As used in:
      - `in %dx,%al` and `movl %al, mem`
  - With DMA
    - » DMA supplies from/to address and data
  - So DMA cuts down the number of bus cycles needed to transfer data between devices
- PC has seven DMA channels

Using DMA

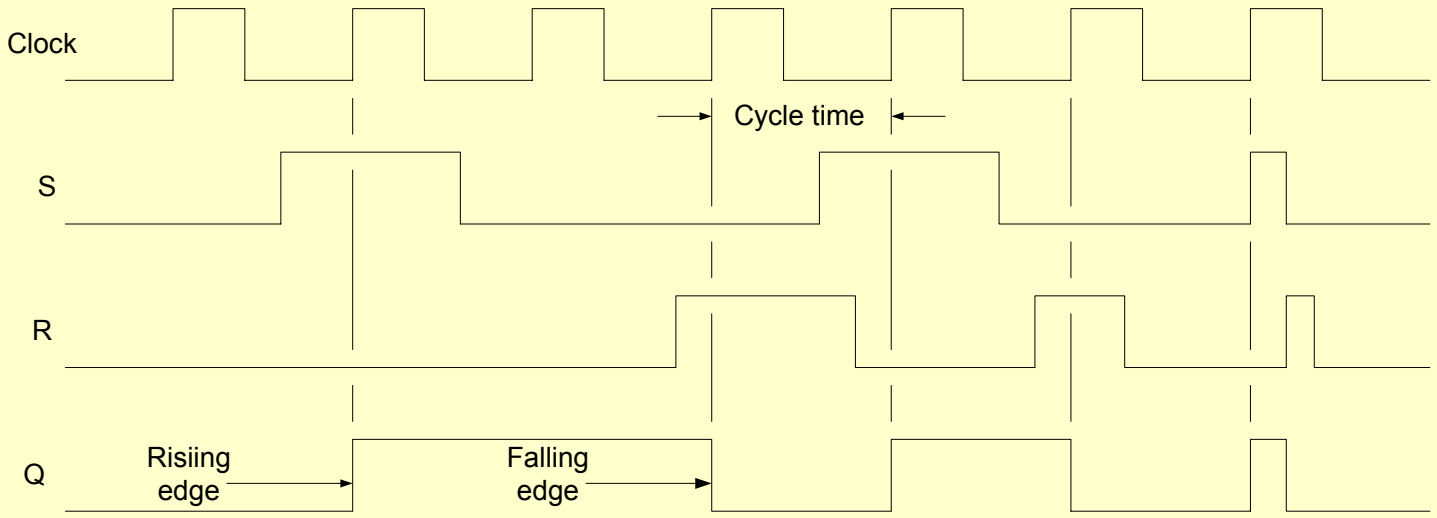
- Programming described on page 409 in S&S
- One limitation is the size of the DMA address space (16-bit value)
- Must use DMA page registers to access memory up to 16MB
- Performance:
  - Limited by bus speed to less than 4MB/sec.!
  - Can increase using block mode data transfers
    - » But this interferes with DRAM refresh
  - Faster CPUs (486 & Pentium) use string move instructions

Some Register Terminology

- Registers and latches
- Level-triggered versus edge triggered
- Use of a clock to make the circuit synchronous



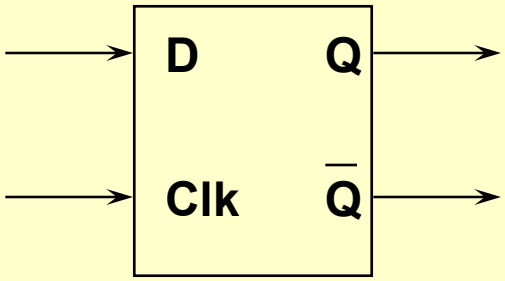
Waveform or Timing Diagram



Waveform diagram for clocked RS Flip-flop

The D-Type Flip-Flop

- Single data input and a clock
- Also called a “delay” flip-flop
- Internally uses master/slave form and becomes an “edge-triggered” flip-flop

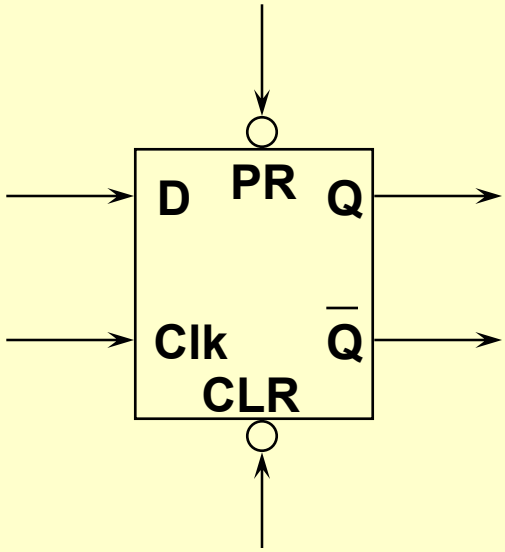


TRUTH TABLE

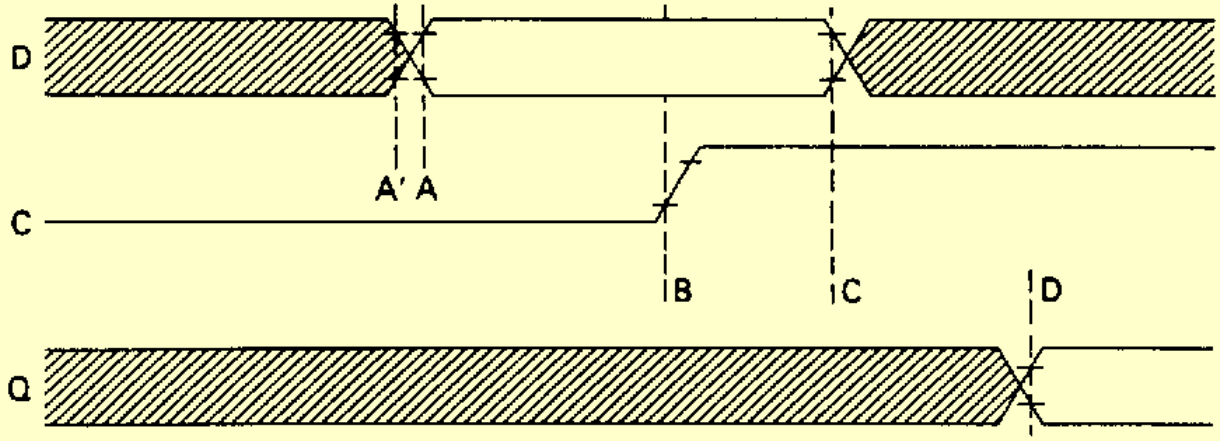
D	Clock	Q
0	⌐	0
1	⌐	1

Actual D-Type Flip-Flop

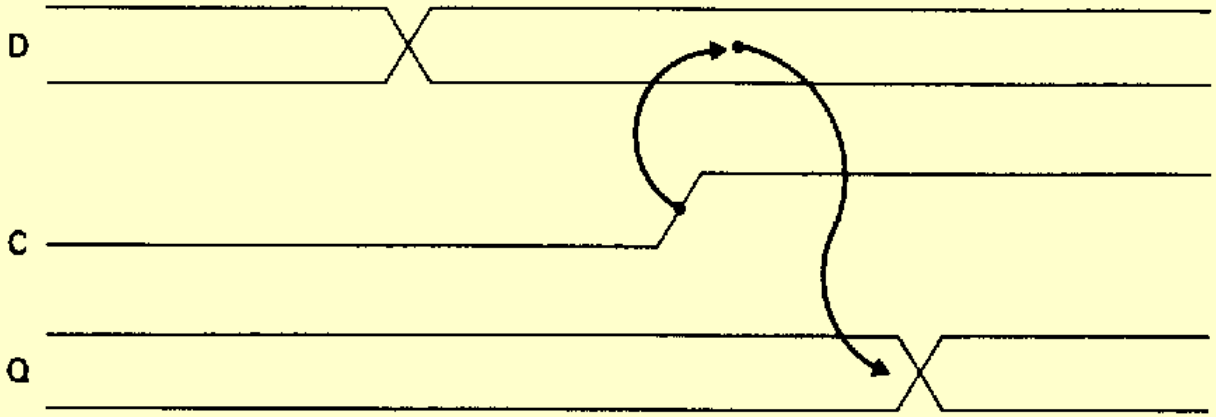
- Has preset (PR) and clear (CLR) inputs which can be set asynchronously (but not both at one time)
- Nomenclature use > for an edge-triggered input



Timing Diagrams for D Flip-Flop



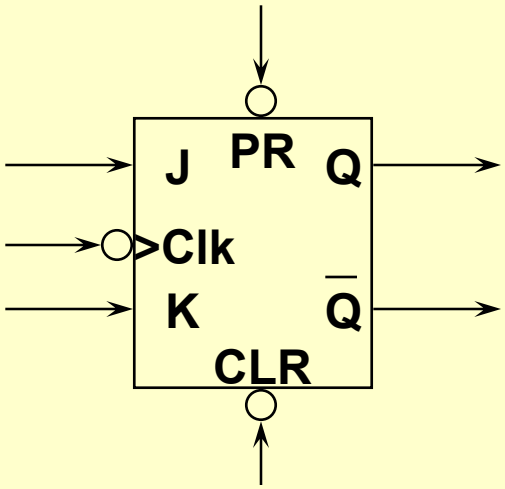
(c) The general form of the timing diagram



(d) An alternative form of the timing diagram

The J-K or Universal Flip-Flop

- Can build other FFs from it
- Three synchronous inputs (plus preset and clear)
- Can be edge- or level-triggered

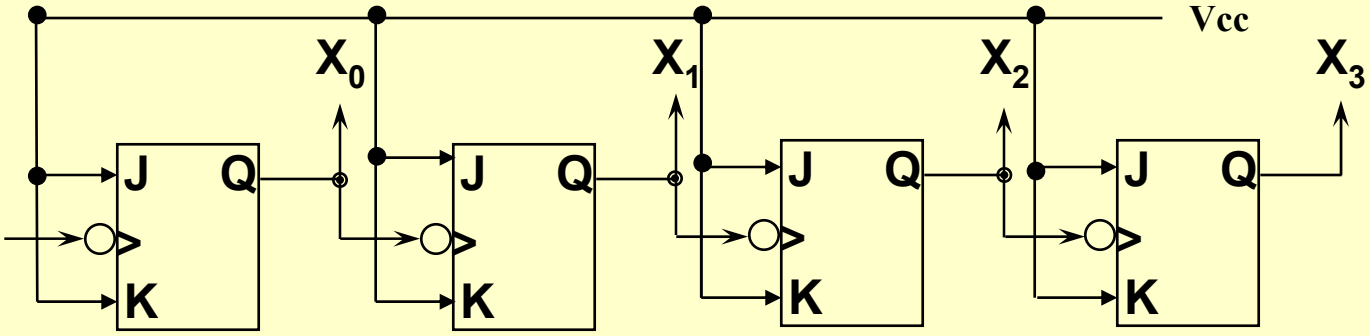


TRUTH TABLE

J	K	Clock	Q
0	0	⌐_	Stays same
0	1	⌐_	0
1	0	⌐_	1
1	1	⌐_	Toggles

Using Flip-Flops

- Primary use is for storage and counting
- Example: Mod-16 counter also known as a ripple counter



$X_3\ X_2\ X_1\ X_0$ counts 0 ... 15 (Decimal) sequentially

Timing diagram for Mod-16 Counter

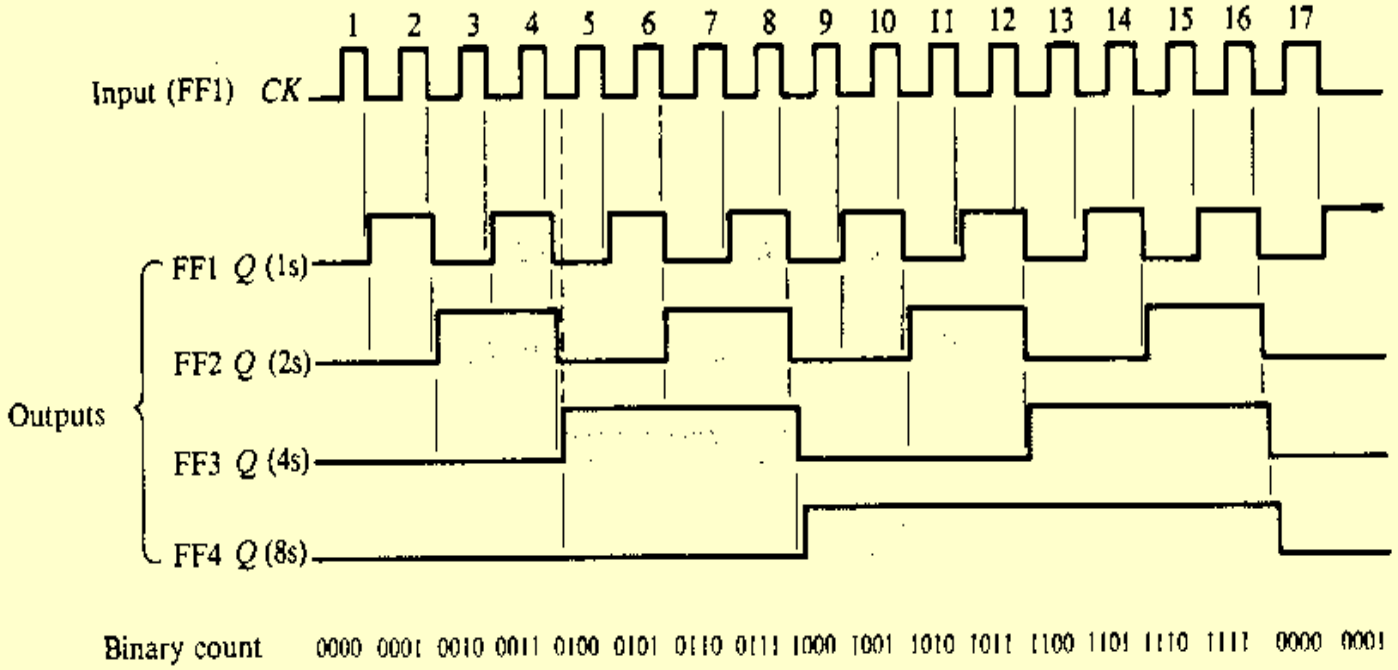
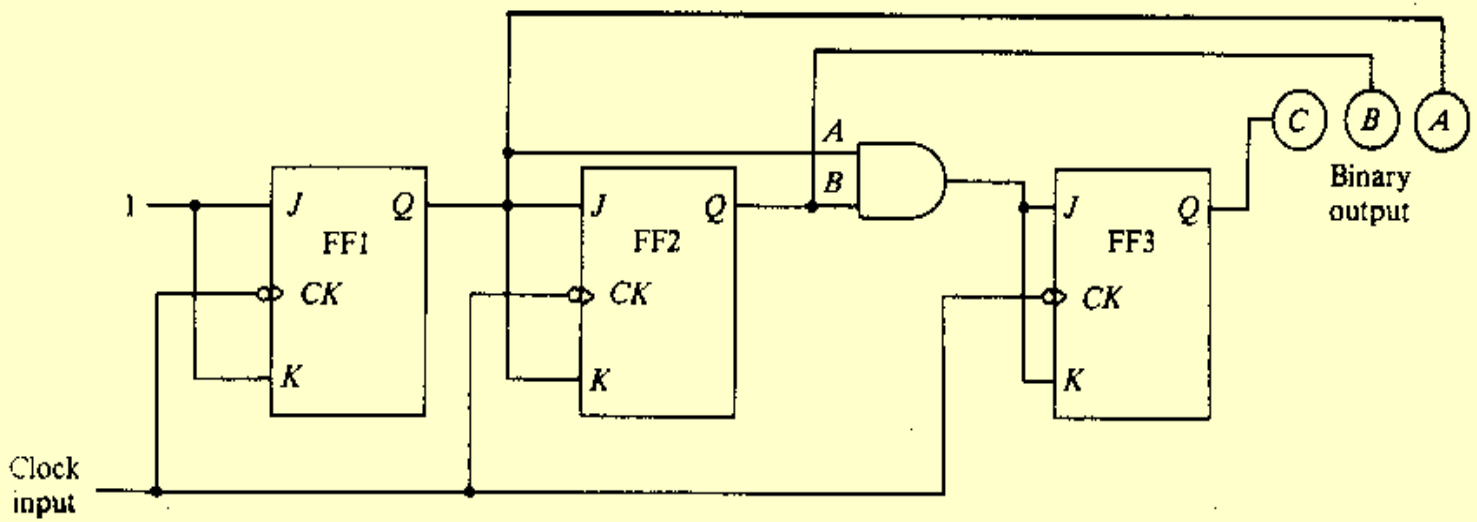


Fig. 8-3 Timing diagram for a mod-16 ripple counter

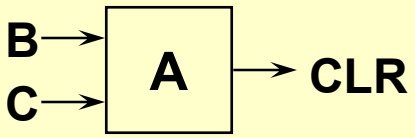
Note that the counter actually serves to divide down!

A Synchronous or Parallel Counter



(a) Logic diagram

1) What does this count to?
2) Can we we make it count to something different?
   Ans. Yes, using an AND gate:



What Kind of Chips Are Available?

- J-K flip-flop is a 7473
- Synchronous BCD up/down counter is a 74192

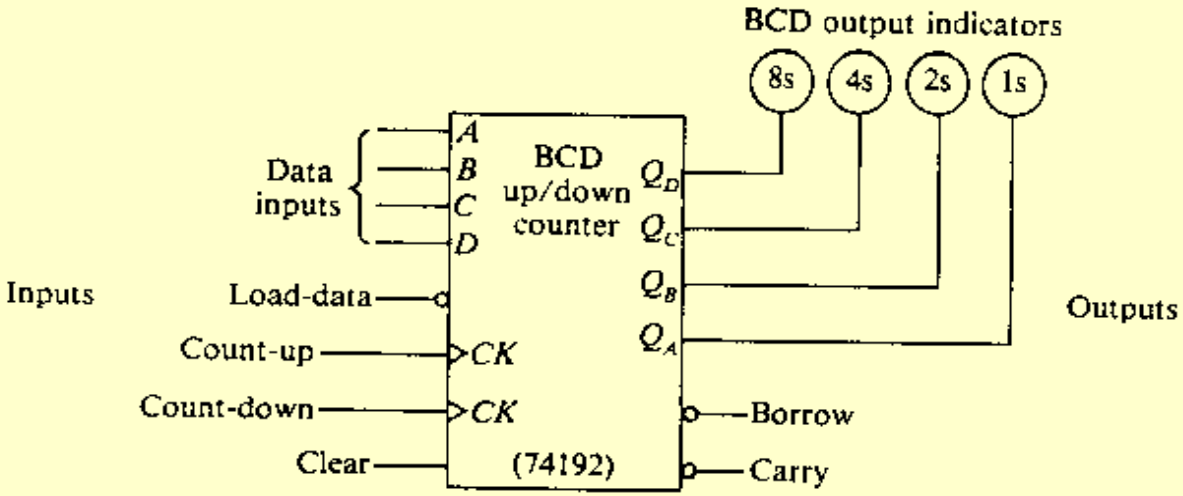


Fig. 8-12 The 74192 synchronous BCD up/down counter IC

Shift Registers

- **Many Combinations:**
  - **Serial in, serial out**
  - **Serial in, parallel out**
  - **Parallel in, serial out**
  - **Parallel in, parallel out**

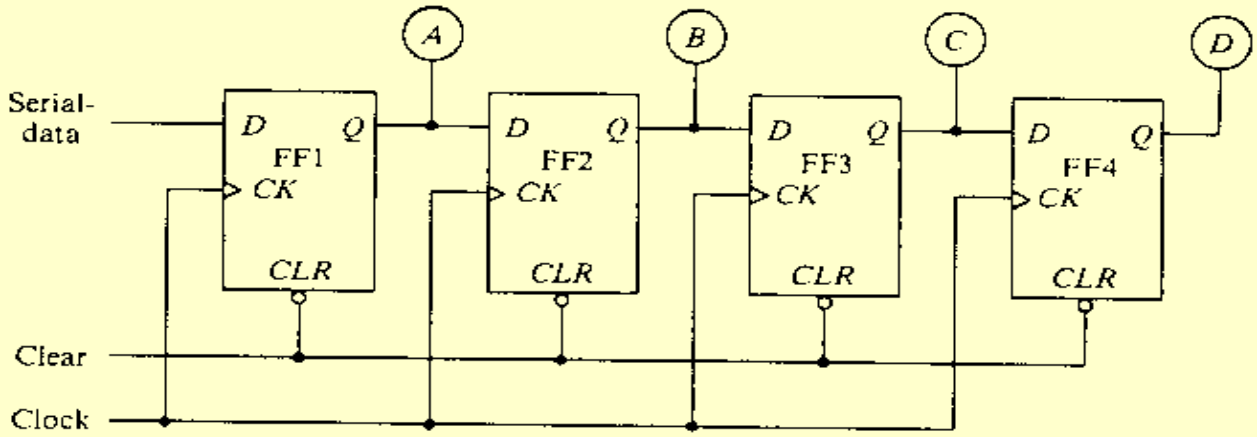


Fig. 9-2 Logic diagram of a 4-bit serial-load shift-right register

Parallel Load, Recirculating

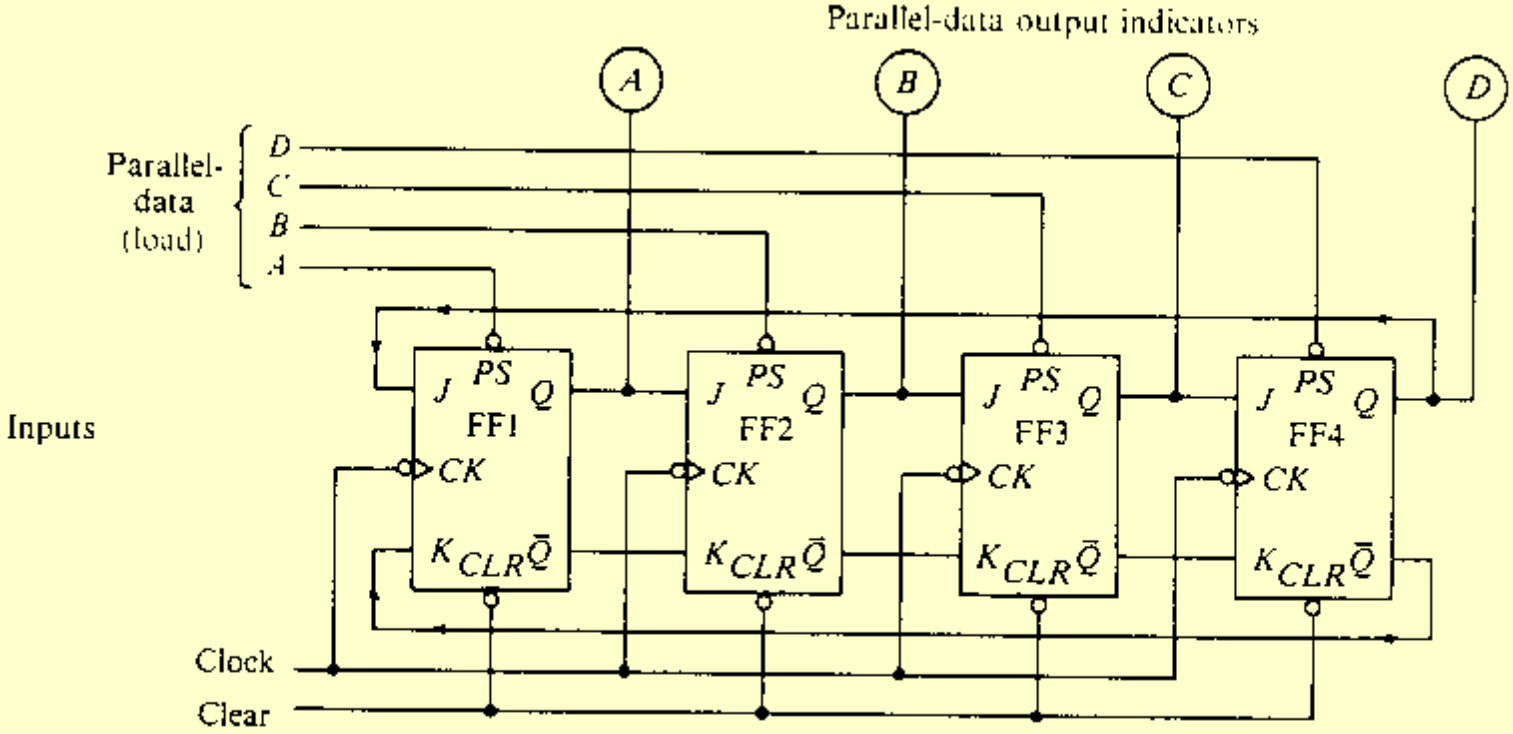


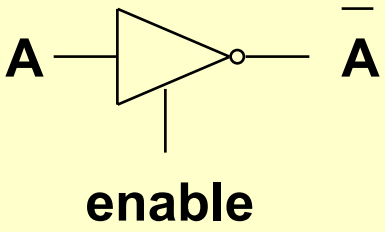
Fig. 9-5 Logic diagram of a 4-bit parallel-load recirculating shift-right register

Look up the universal shift register -- 74194 (Fig. 9-8)

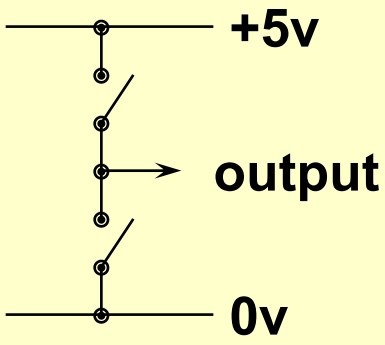
Tri-State Logic

- The problem with connecting element together is that each has to be in one logic state (0) or the other (1)
- This represents a conflict and we resolve it with ***tri-state logic***

Logically



Electrically

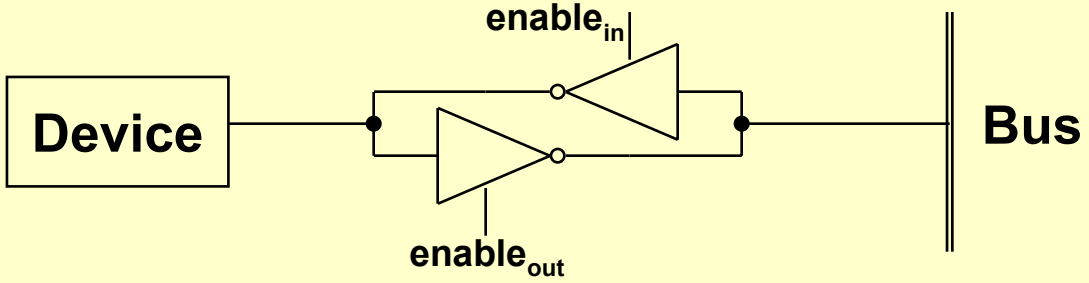


Truth Table

enable	A	Output
0	0	X
0	1	X
1	0	1
1	1	0

Tri-State Logic and Buses

- Often the logical element has an output enable pin to go from a floating output to the actual output of the circuit
- Inverters and buffers are used as bus drivers or buffers
  - Two such drivers or buffers are used to make the connection bi-directional
  - The gates also provide more "drive" onto the bus so that the bus signals are stronger and the bus can be longer

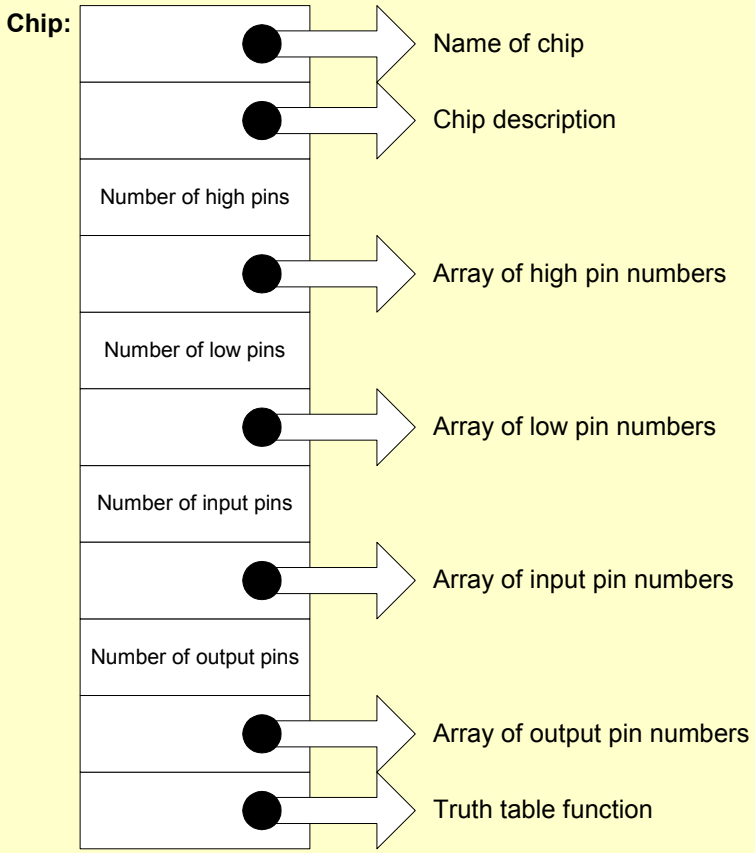


Some Notes on Machine Problem 4

- Two parts:
  - Chip tester
  - Digital oscilloscope
- Both use the parallel port as a means of performing digital I/O
  - Digital oscilloscope uses the serial port as the source of bits
- Most of the code is written so the number of new lines of code is small
  - But effort to understand other's code is substantial

Chip Tester

- **Chip data structures central to the testing code**



Chip Tester Configuration

- SAPC boards 5 & 6 have the parallel port wired to test an LS00 (quad NAND gate chip)
- SAPC boards 7 & 8 have the parallel port wired to test an LS138 (3-to-8 decoder chip)
- The given code makes a correspondence between:
  - Data register bits for the parallel port (0:7)
  - The DB25 connector (pins 2-9 and 10-13, 15)
  - The chip pins (1-14 or 1-16)
  - The direction of the chip pins (input or output)
- Note that the LS00 is symmetric in that the pins on the left side of the chip are functionally equivalent to the pins on the right side

ls00_softchip

- The truth table holds the testing information:

```
/* truth table:   in pin  5 4 2 1  out pin  6 3 */
static int TT[] = {
            0x03,     /* 0 0 0 0          1 1 */
            0x03,     /* 0 0 0 1          1 1 */
            0x03,     /* 0 0 1 0          1 1 */
            0x02,     /* 0 0 1 1          1 0 */
            0x03,     /* 0 1 0 0          1 1 */
            0x03,     /* 0 1 0 1          1 1 */
            0x03,     /* 0 1 1 0          1 1 */
            0x02,     /* 0 1 1 1          1 0 */
            0x03,     /* 1 0 0 0          1 1 */
            0x03,     /* 1 0 0 1          1 1 */
            0x03,     /* 1 0 1 0          1 1 */
            0x02,     /* 1 0 1 1          1 0 */
            0x01,     /* 1 1 0 0          0 1 */
            0x01,     /* 1 1 0 1          0 1 */
            0x01,     /* 1 1 1 0          0 1 */
            0x00,     /* 1 1 1 1          0 0 */
};
```

ls138_softchip

- Have three bits of input data (A_0, A_1, and A_2)
- Have three bits of enable ($*E_1$, $*E_2$, and E_3)
  - Where * means enabled low
- Can support only five bits of output data ($*O_0$, $*O_1$, …$*O_7$)
- Since softchip function has two arguments, the input data and the output results one solution would be a 32 entry table – but there are other ways of doing this

Changes to chiptest.c

1. **Add new entry for LS138 into chip table**
2. **Print out the description of the chip pins**
3. **Generate the possible output bit patterns**
4. **Compare output bit patterns to softchip results to see if chip is functioning correctly**

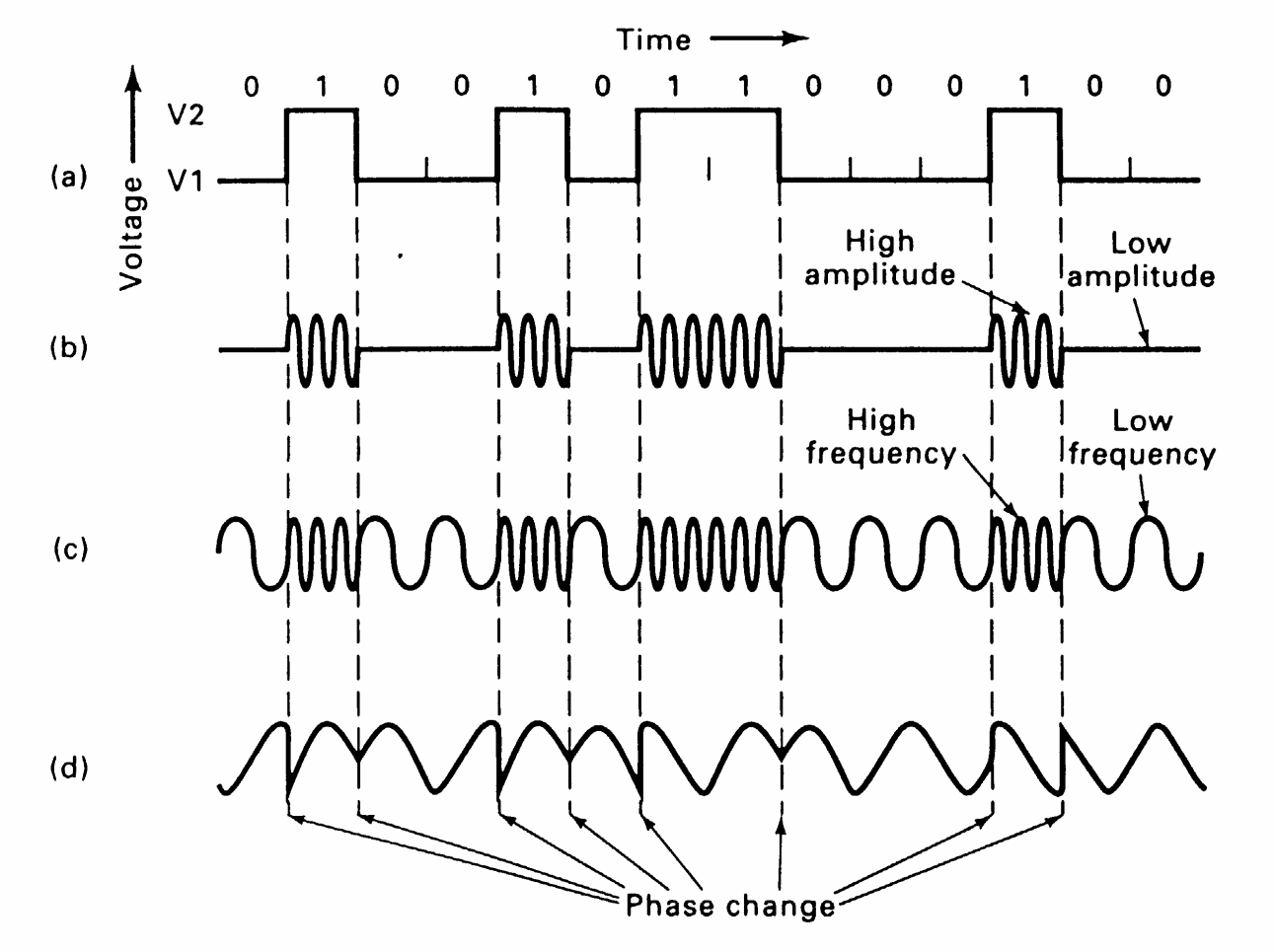
Digital Oscilloscope

- **Hook back the COM1 port to LPT1 but use a level converter (called a *line receiver*) to make sure voltage conversion is done**
- **Project description explains that sampling rate is about 100 times faster than bit times (at 9600 baud)**
- **Using | for high and _ for low, plus ||nn|| or __nn__ for a sequence of highs and lows, the displayed results might be: ||88||__67__|__||___||82||**
  - **or "88 highs", "67 lows", "1 high", "2 lows", "2 highs", "3 lows", and "82 highs"**

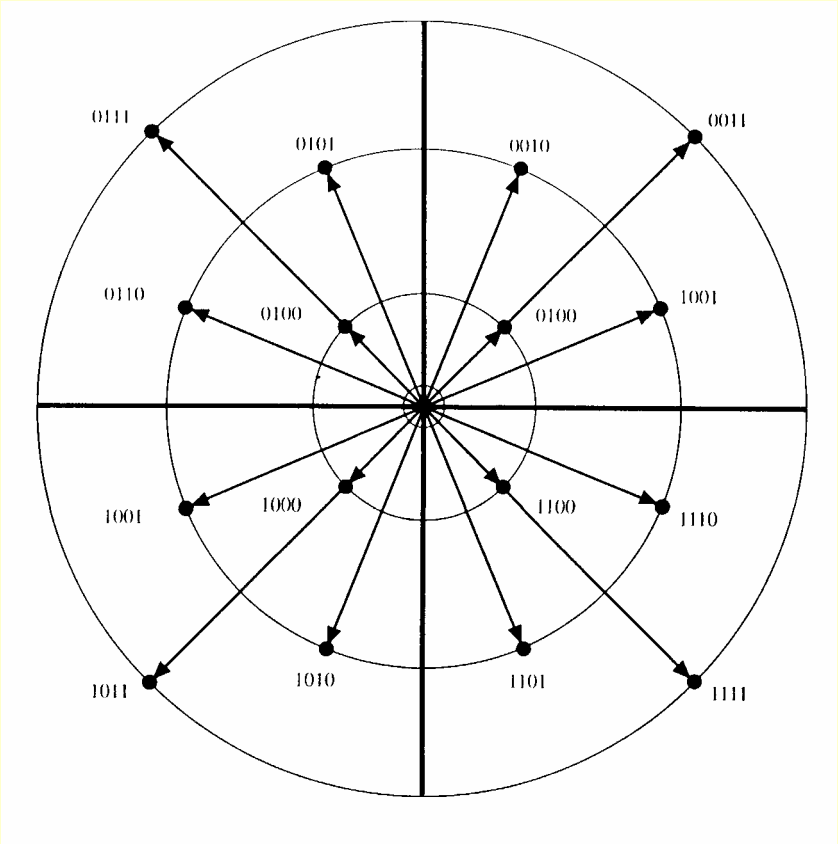
From bps to Baud

- **What's the difference between baud and bits per second?**
  - ***Baud*** **- the number of discrete conditions or signal events per second**
  - ***bps*** **- the number of bits transmitted per second**
  - **Baud is the same as bps only if each signal event represents exactly one bit; so in general: Baud ≤ bps**
- **Distinction a result of using modems**
  - **POTS can't transmit computer signals**
  - **Instead a modulated sine wave is transmitted in the range of 1-2 KHz; the sine wave is the *carrier***
  - **Modulation takes many forms: *amplitude*, *frequency*, and *phase* modulation**

Transmission of 01000101100010 0



Signal Constellation for 16-pt QAM



- Telephone bandwidth 300-3300 Hz (or 3000 baud)
- QAM offers 4 bits per baud using 12 possible phase shifts and 3 possible amplitudes
- V.32 uses baud rate of 2400 and QAM to yield 9600 bps
- Adding in additional encoding and compression yields today's V.90 modems

Baudrate Generator

- **Characters are sent/received asynchronously**
  - **Clocks of receiver and transmitter are independent and only nominally the same**
  - **Furthermore, the relative phases of the clocks are completely arbitrary**
- **Receiver strategy:**
  - ***Synch* on initial rising edge then sample bits**
  - **Sample 16 times the baud rate, starting with the eighth clock period after leading edge of start bit**
- **On SAPC, clock comes from 1.8432 MHz crystal**
  - **1.8432/16 = max baud rate = 115,200 bps**
  - **Rate set on SAPC by value in *divisor latch***

Baudrate Generator (Cont'd)

- **Set DLAB bit in "Line Control" UART register to use UART registers 0 and 1 for 16-bit divisor value**

```
outpt(baseport + UART_LCR, (inpt(baseport + UART_LCR)) | UART_LCR_DLAB);
```

- **Then put out two bytes containing divisor**
  - **9600 bps => divisor of 12**
  - **19,000 bps => divisor of 6**

Connecting Serial Devices

- Standard was to connect a DTE to a DCE
- But we don't always have the luxury, so …
  - Transmitted data (pin 2 - outgoing) connected to received data (pin 3 - incoming)
  - Request to send (pin 4 - outgoing) connected to clear to send (pin 5 - incoming)
  - Data terminal ready (pin 20 - outgoing) connected to data set ready (pin 6 - incoming)
  - Signal grounds (pin 7) connected to each other
  - Carrier detect (pin 8 -incoming) depends on whether or not modem control is required

Setting the Baud Rate

- UART over samples incoming bit
  - Minimum is 16X
  - UART clocked at 1. 843200 MHz
  - Must set up UART divisor so that it is:

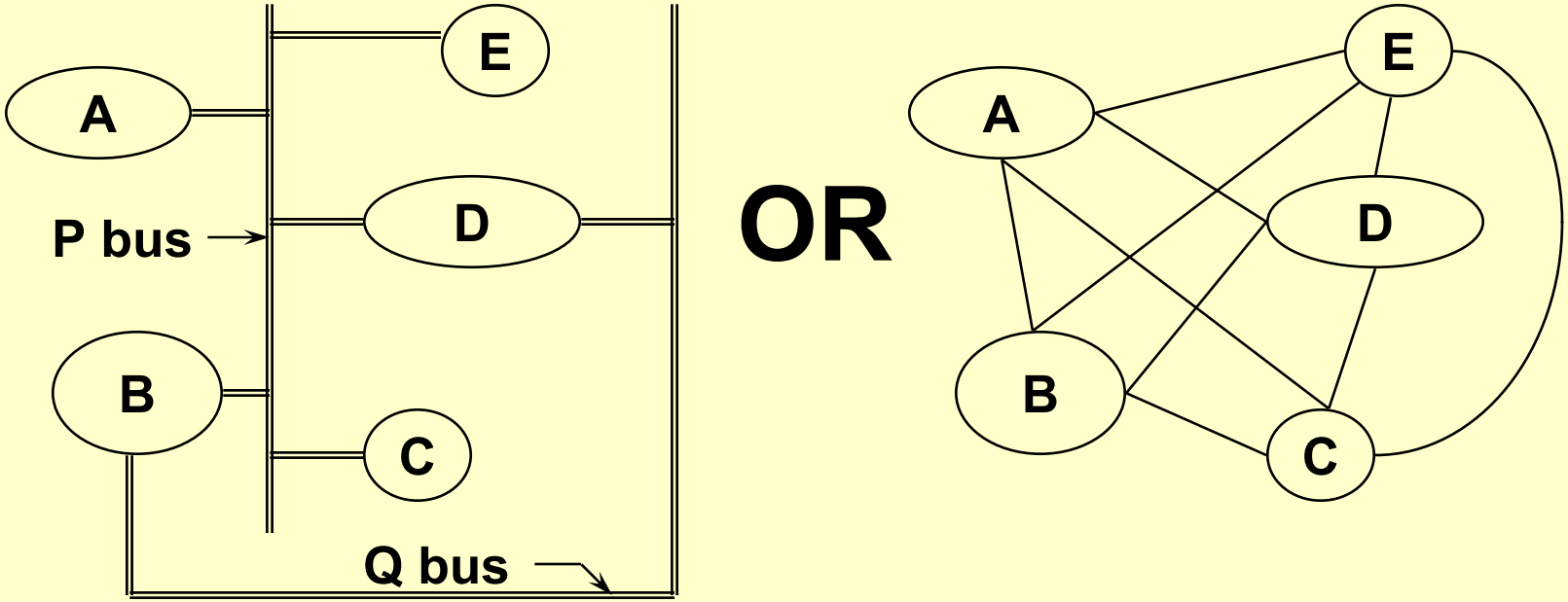
    ```
    UART_BAUD_CLOCKHZ/(baudrate * 16)
    ```
  - Look at `serial.h` where UART baud rate is set through the Line Control Register (LCR)
    - » Must turn on Divisor latch access (UART_LCR_DLAB)
    - » Must set word length to 8 bits (UART_LCR_WLEN8)
    - » Then load LSB followed by MSB
    - » Finally, turn off DLAB

Remaining Steps in `scope.c`

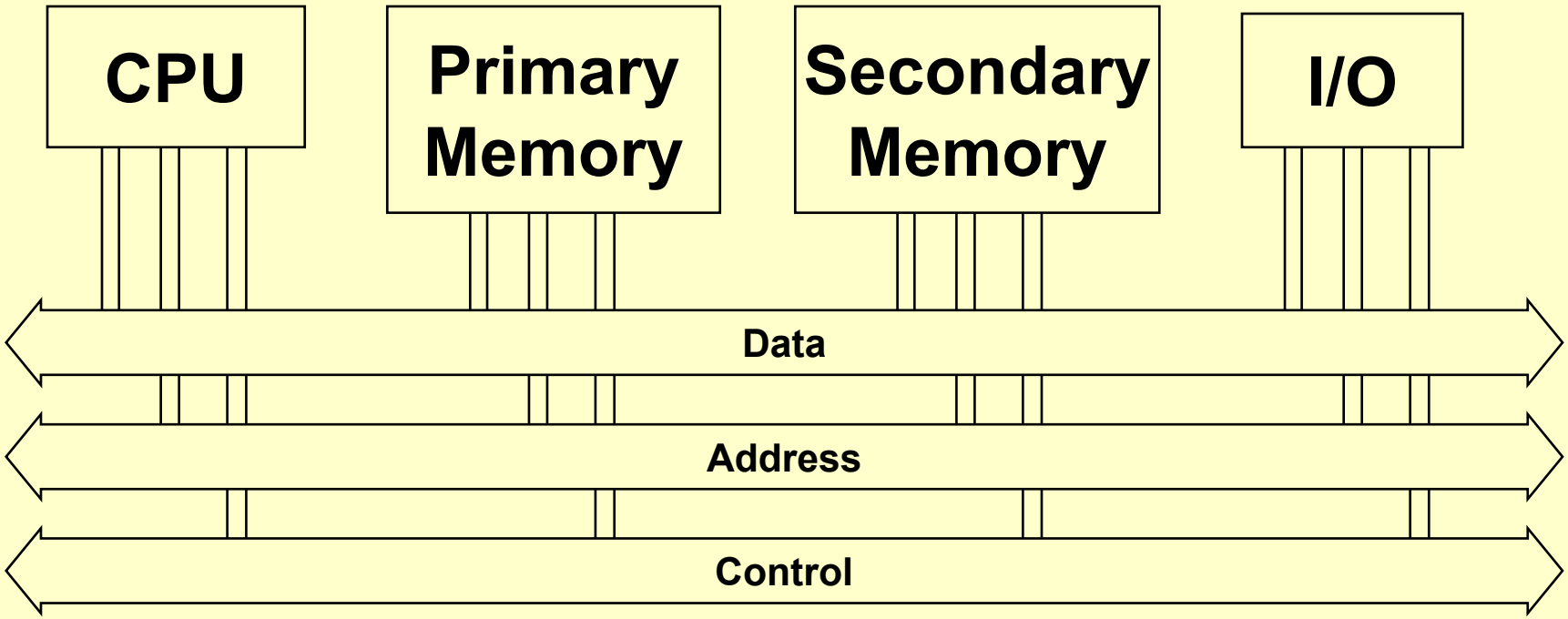
- **After setting up baud rate**
  - **Output a character to the serial port (COM1 but could be COM2)**
  - **Collect the data**
    - » **`collect.c` inputs from the `LP_STATUS` bit and saves into an array of data points up to a `MAXDATA` times**
    - » **You are asked to re-implement `collect.c` as an assembly language routine `ascollect.s`**
      - ❍ **Code is similar to `collect.c` in that it loops `MAXDATA` times moving the `LP_STATUS` bit into the data array**
  - **Display the results**
    - » **This code is written for you**

Buses

- Concept is to link together multiple functional units over a common data highway at a lower cost than connecting them directly



Bus - Essential Part of Any Computer



Bus Arbitration

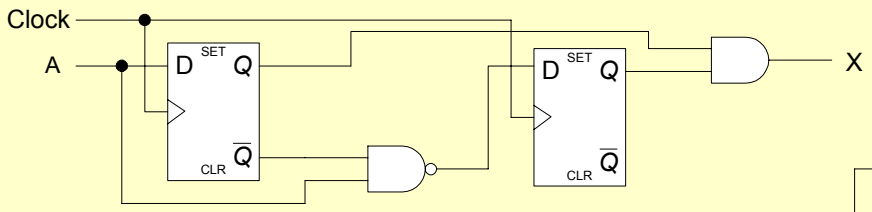
- Needed for intelligent peripherals (e.g., DMA), multiple CPU, and dynamic memory refresh
- Protocol implements some form of a bus request, bus grant, and bus acknowledge

Logic Summary

- **Combinational circuits:**
  - Made from gates without feedback
  - Outputs depend only on current inputs
  - Fully defined by truth table
  - Have no internal states
  - Does not use a clock; states constantly changing with inputs
- **Sequential circuits:**
  - Have feedback among the gates
  - Can have internal states
  - Outputs depend on inputs and past inputs (via internal states)
  - Often uses a clocked input
  - Not completely described by pure truth table on inputs

Describing Sequential Circuits

- **In general,**
  - **Next state = *f(inputs, last state)***
  - **Outputs = *f(inputs, last state)***
- **Example:**

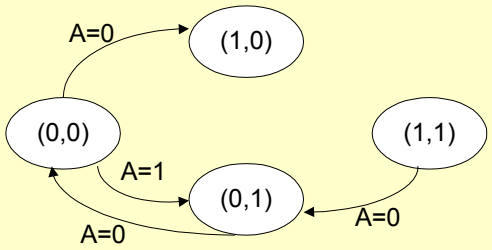


State = (Q1, Q2) [2 bits]
4 states: (0,0), (0,1), (1,0), (1,1)

Truth Table

A	Last State Q1	Last State Q2	Next State Q1	Next State Q2	X
0	0	0	0	1	0
1	0	0	1	0	0
0	0	1	0	0	0
.	.		.		.
.	.		.		.
.	.		.		.
0	1	1	0	1	1
1	1	1	1	1	1

- **State diagram:**



Instruction Execution Cycle

- **Instruction fetch**
  - Read instruction from memory
- **Instruction decode**
  - Inside CU; no memory access
- **Address generation**
  - Inside CU; may or may not access memory
- **Instruction execution**
  - Needed data may require memory access, then ALU operation performed
- **Write**
  - Processor state modified and results may be written to memory

See S&S, page 347

Enhancing Performance

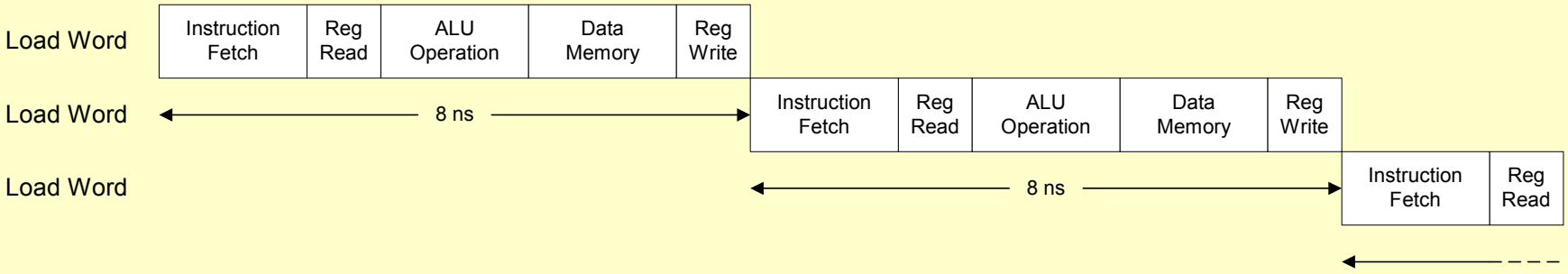
- **"Pipelining is an implementation technique in which multiple instructions are overlapped in execution",** (Patterson and Hennessey, "Computer Organization and Design", p. 436)
  - **It improves instruction throughput rather than individual instruction execution time**
  - **It exploits parallelism among the instruction in a sequential instruction stream**
  - **Under ideal conditions the speedup from pipelining equals the number of pipe stages**
  - **But there is some overhead associated with pipelining so SU is not ideal**
  - **No stage may be faster than the slowest stage of the pipe, or to put it another way, the slowest instruction determines the total time for the pipe**

Pipeline Example

- **From H&P, have seven single-cycle instructions with various timings in a five-stage pipe**

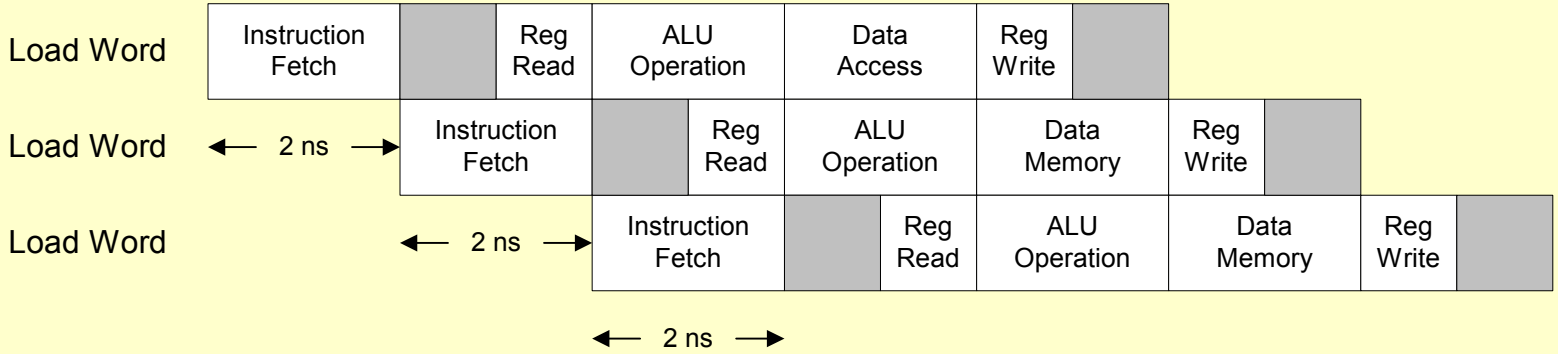
Instruction type	Instruction Memory	Register Read	ALU operation	Data Memory	Register Write	Total time
Load word	2 ns	1 ns	2 ns	2 ns	1 ns	8 ns
Store word	2 ns	1 ns	2 ns	2 ns		7 ns
ADD, SUB, AND, Or. SLT	2 ns	1 ns	2 ns		1 ns	6 ns
Branch	2 ns	1ns	2 ns			5 ns

- **Sequential execution:**



Pipeline Example (Cont'd)

- Using a 2 nsec clock cycle with pipelining



- Recall that pipelining improves performance by *increasing instruction throughput as opposed to decreasing the execution time of an individual instruction*
- Notice the idle time in the pipe at certain times

Pipeline Performance

- Ideal speedup is number of stages in the pipeline. Do we achieve this?
- Throughput ≡ # task completed / unit time
- Given *k* tasks and an n-stage pipeline where each stage takes the same unit of time to process and task arrive at the same unit time intervals:
  - It takes n time units to fill pipeline and process first task
  - Thereafter, pipeline processes 1 task every unit of time

$$T_p(k,n) = n + (k-1)$$

$$\text{Throughput} = \frac{k}{n + (k - 1)}$$

Pipeline Performance (Cont'd)

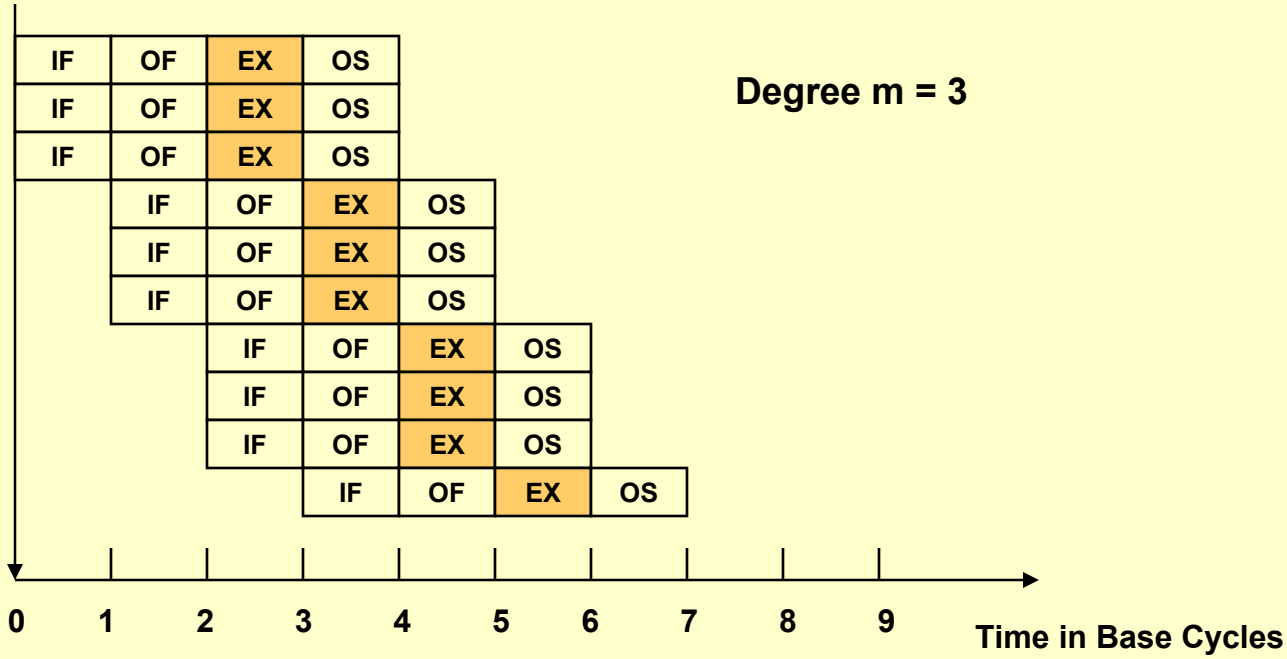
- For a non-pipelined machine

$$T_{np}(k,n) = nk$$

- So the speed-up (SU) is

$$SU = \frac{T_{np}(k,n)}{T_p(k,n)} = \frac{nk}{n + (k - 1)} \approx n \quad \text{for } k >> n$$

Superscalar Processors



Memory Access

- In timing an instruction we need to account for the number of bytes for memory I/O:
  - To hold the instruction
  - For operands fetched
  - To store results in
- The SAPC accesses memory as:
  - Caches instructions and saves unused parts
  - 64-bit units regardless of actual data size
- Instruction timing
  - Minimum of one cycle (2.5 nsec. for the 400 MHz 486)
  - Plus time to read/write memory, if not cached

Memory Access (Cont'd)

- **Cache**
  - Between CPU and main memory
  - Speeds up access to frequently used memory locations
  - May be write-back or write-through
- **Calculating read/write cycles**

`movl %eax, %edx`	reg/reg instruction	(2 bytes) .25R
`movl %eax, total`	reg/mem instruction*	(5 bytes) 0.625R+1W
`movl %edx, total`	reg/mem instruction	(6 bytes) 0.75R+1W

*accumulator is special

Makefiles

- **General form:**

```
target: dependencies
        <tab> commands
        <tab> commands
        <blank line>
```

"rule"

- **In the mp3 makefile we find:**

```
$(C)_dbg.opc: $(C).c $(PC_INC)/cpu.h …
    $(PC_CC) $(PC_CFLAGS_DBG) -c -o $(C)_dbg.opc $(C).c
```

- **We we invoke the makefile with C=itimes we get:**

```
PC_INC = /groups/ulab/pcdev/include
PC_CC = i386-gcc
```

Makefiles (Cont'd)

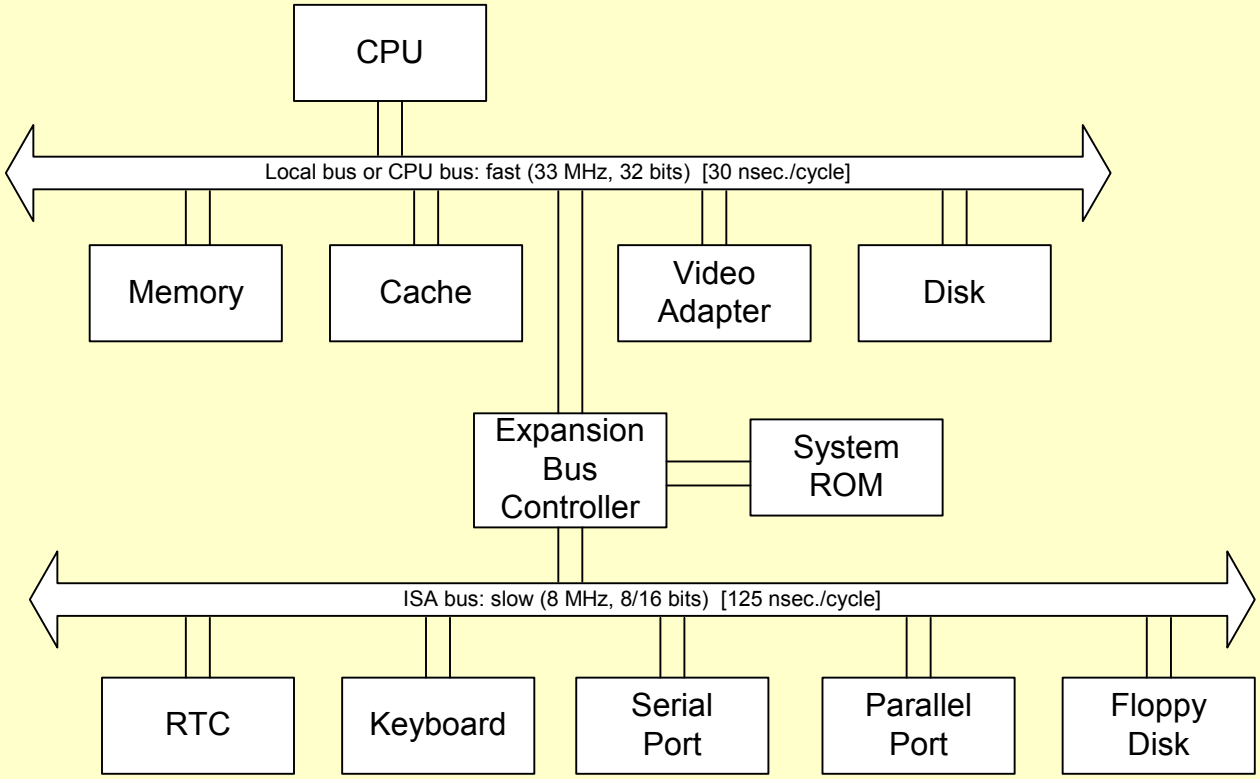
- **The character (macro) substitution then generates:**

```
itimes_dbg.opc: itimes.c /groups/ulab/pcdev/include/cpu.h …
    ~i386-gcc -Wall … -g -c -o itimes_dbg.opc itimes.c
```

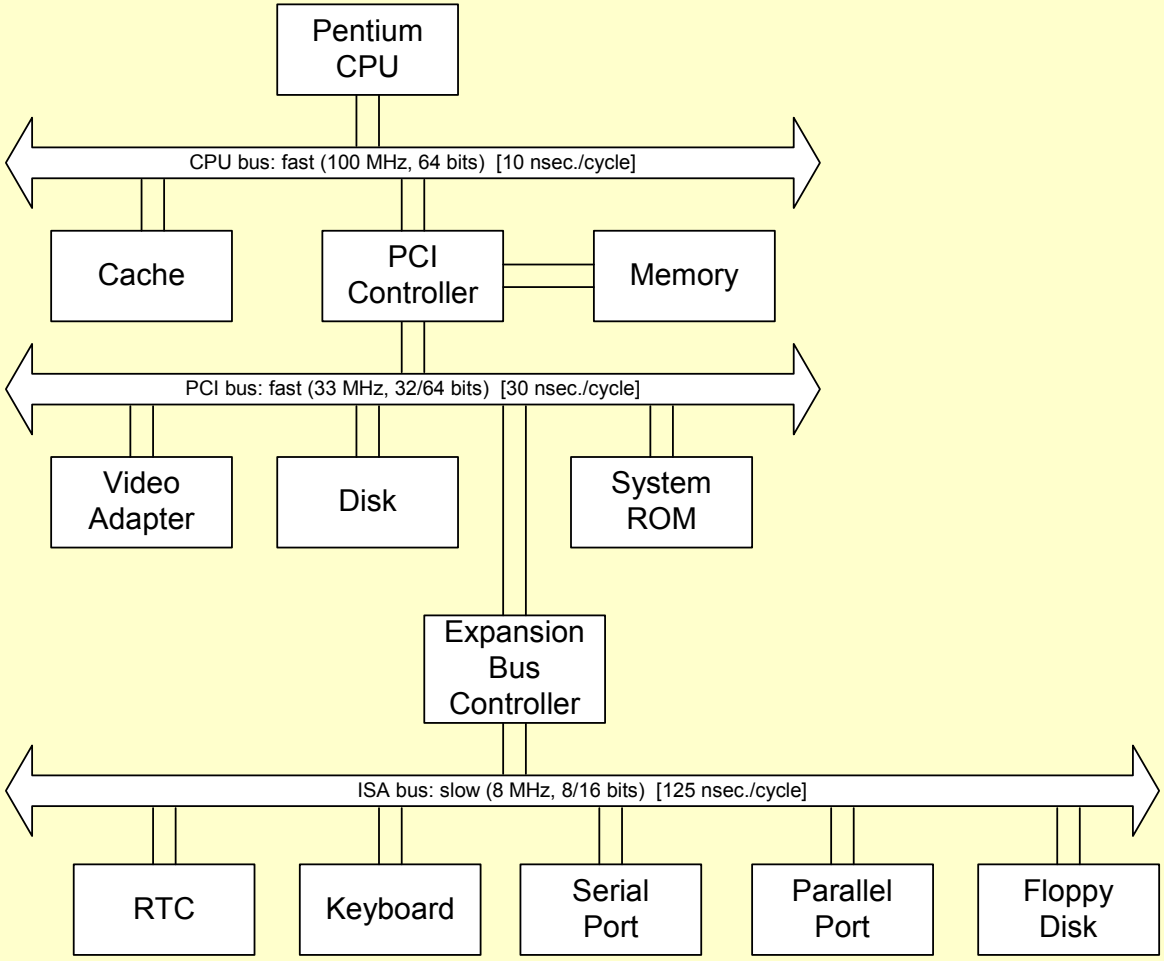
If anything is newer than .opc then the rule is invoked

- **We needed to change this to build optimized (-O2 flag) and put the new command form in "all"**

Big Picture (486)



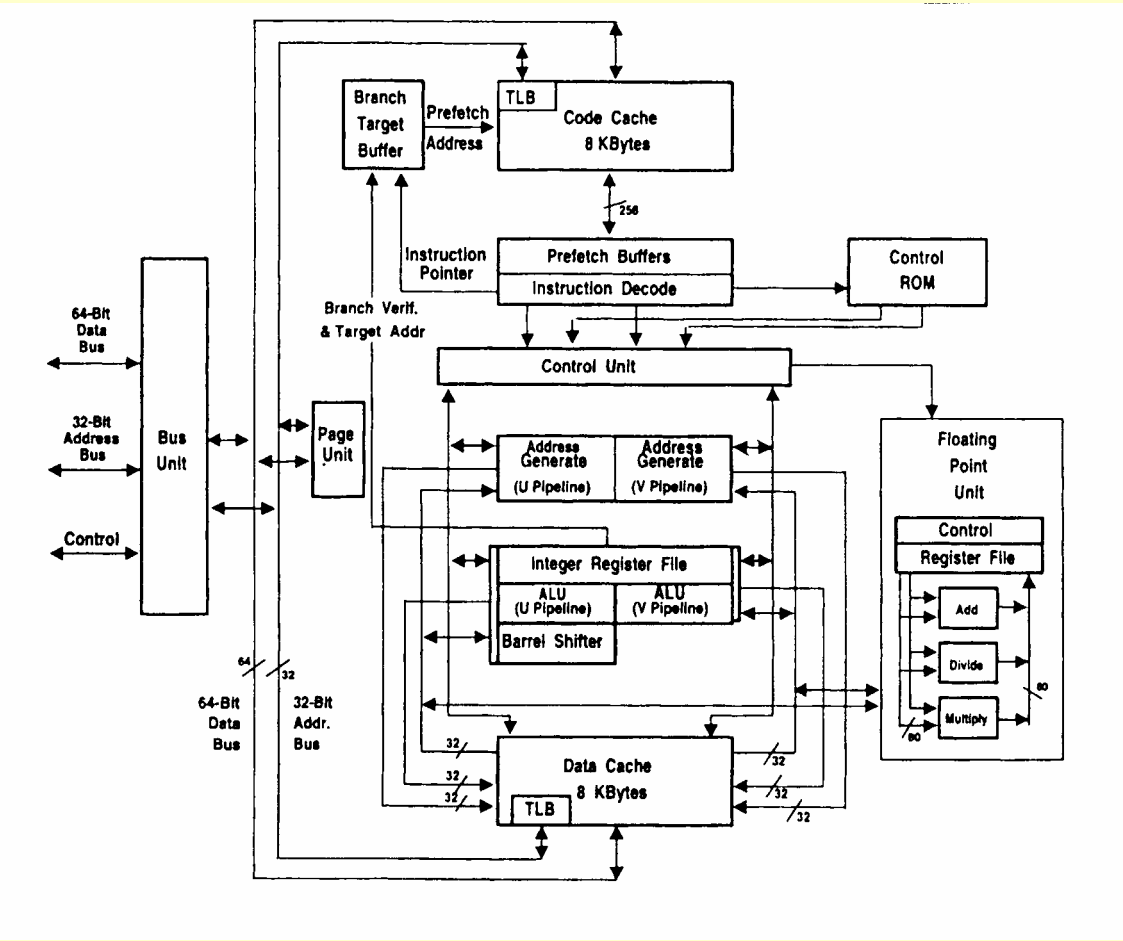
The Big Picture (Pentium)



CPU

- **Pentium processor (400 MHz AMD-K2)**
  - **Small and compact**
    - » **1/2 inch square**
    - » **3.3 million gates**
    - » **10 watts**
  - **Fast**
    - » **Pipelining and superscalar**
    - » **64-bit-wide data path**
    - » **Independent bus unit**
    - » **32-byte prefetch buffers**
  - **Complex**
    - » **Branch target buffer**
    - » **Separate FPU**

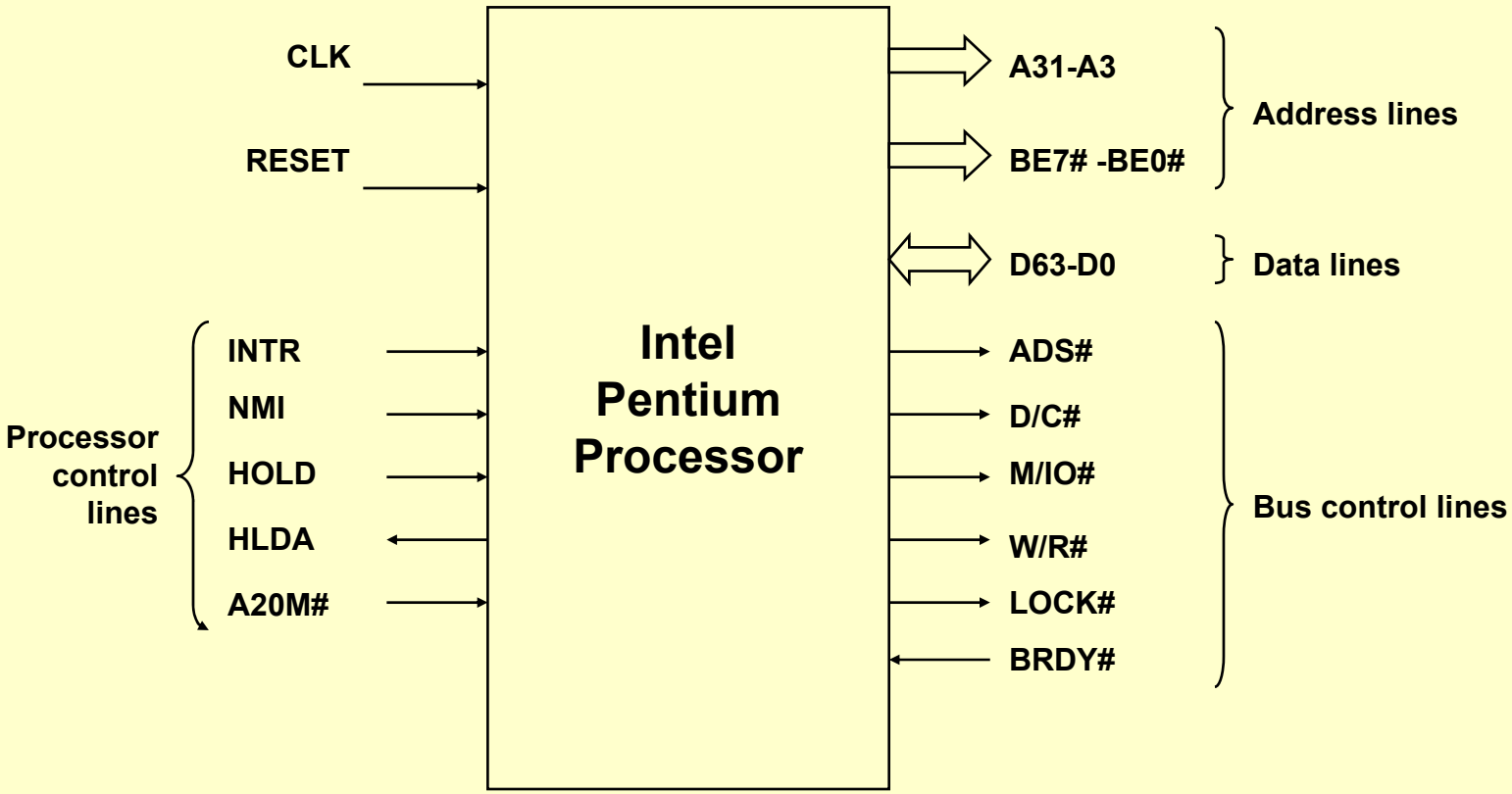
Pentium CPU Block Diagram



CPU (Cont'd)

- **CPU bus**
  - 168 signals divided into 10 groups
  - Address lines and each byte of the data lines have parity
  - System management signals used to implement power management
- **Address lines (total of 37)**
  - A31 - A3
  - BE7# - BE0# (called *byte-enable* lines)
- **Data lines (total of 64)**
  - D63-D0
  - 8 bytes accessed at a time
  - On the bus, memory accesses are aligned on a quadword

Pentium CPU Signals



Pentium CPU Signals (Cont'd)

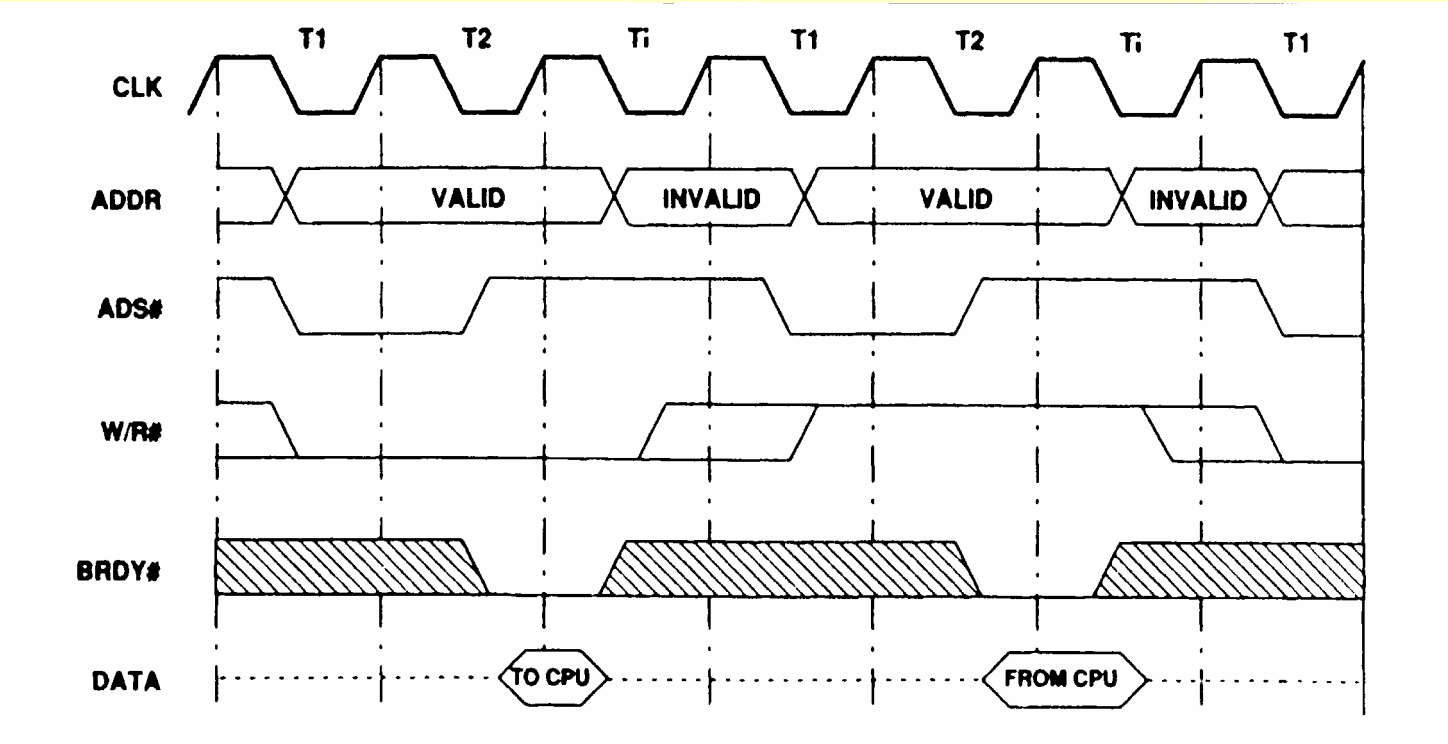
CLK	Clock
RESET	Reset
A31-A3	Address lines
BE7#-BE0#	Byte enable lines
D63-D0	Data lines
ADS#	Address status line (starts new bus cycle)
D/C#	Data or code
M/IO#	Memory or IO port address
W/R#	Write or read
LOCK#	Create read-modify-write sequence
BRDY#	Burst ready (from external device)
INTR	Interrupt
NMI	Non-maskable interrupt
HOLD	Device request CPU to relinquish bus
HLDA	CPU acknowledges new bus master
A20M#	Address line 20 mask

Pentium Reset & Control Lines

- **Reset**
  - **Starts off processor in "real mode"**
  - **Forces cs = 0f000h and ip = 0fff0h where ROM bootstrap resides**
  - **Soft reset possible**
- **Control lines**

M/IO#	D/C#	W/R#	Type of Bus Cycle
0	0	0	interrupt acknowledge
0	0	1	special cycle
0	1	0	I/O port read
0	1	1	I/O port write
1	0	0	memory code read
1	0	1	reserved
1	1	0	memory data read
1	1	1	memory data write

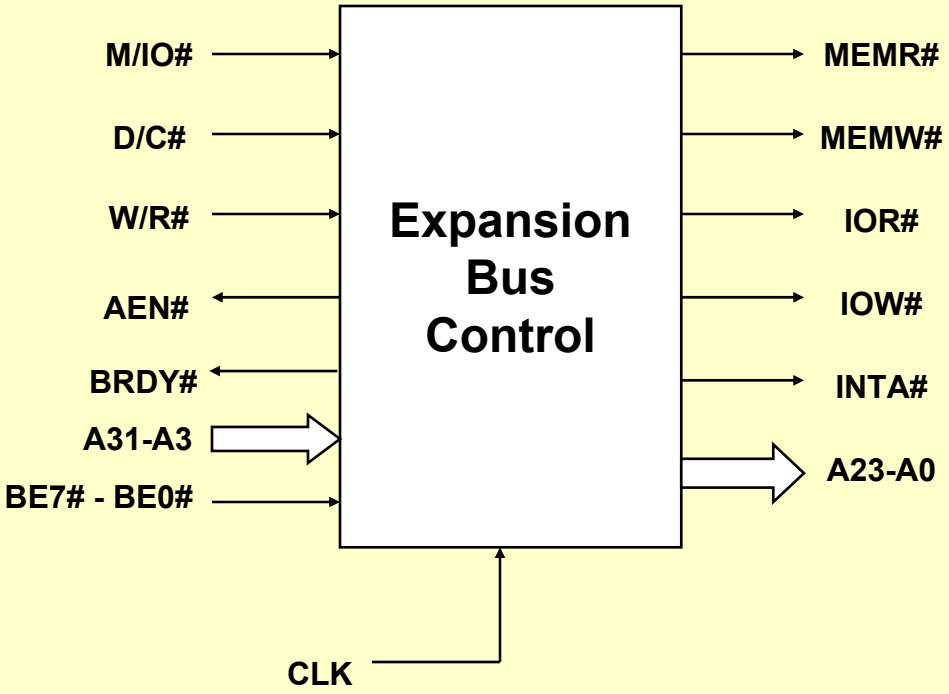
Timing Diagram



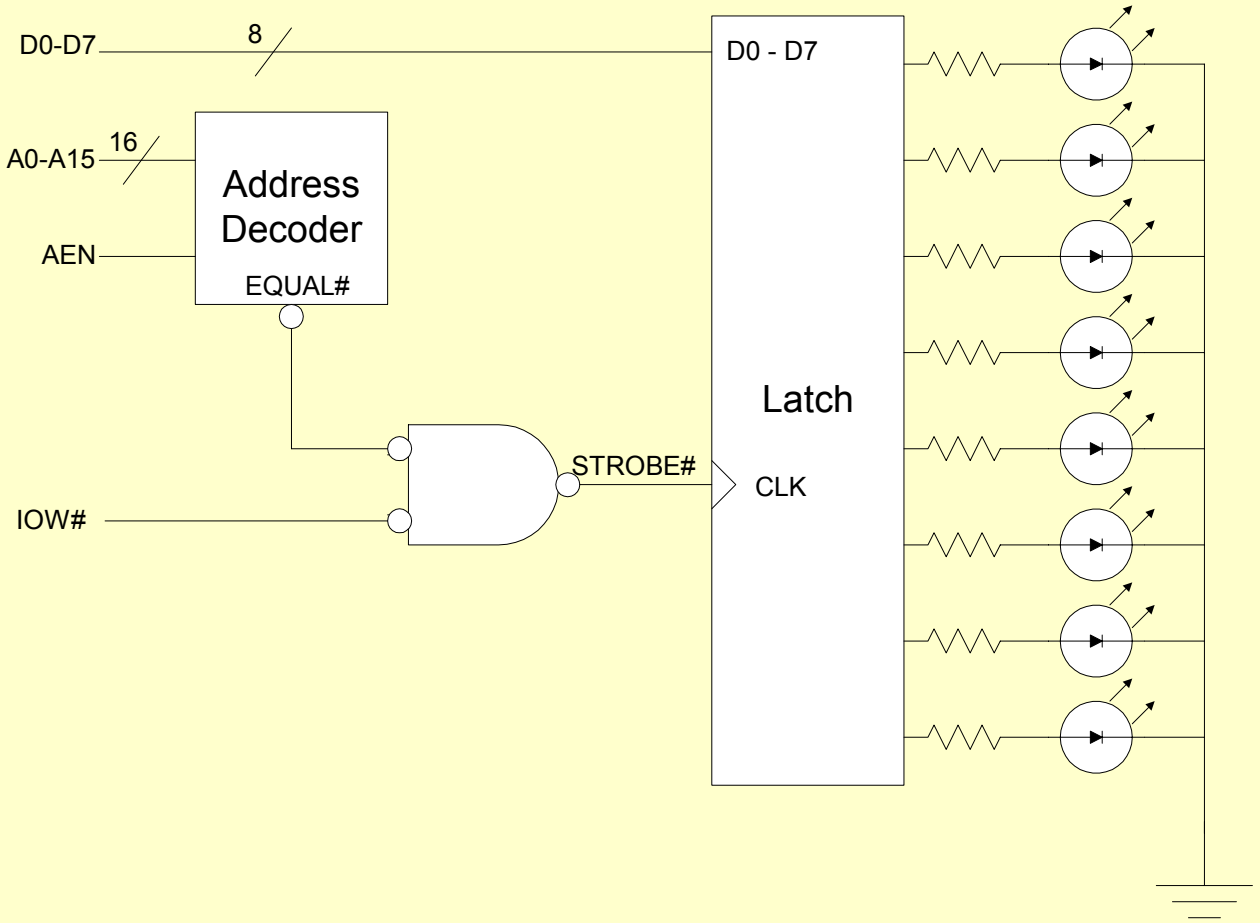
Memory Reads/Writes

- **CPU caches actually read/write in 32-byte units**
  - **Size of one cache line**
  - **Same as four 64-bit read/writes (i.e., four quadwords)**
  - **Uses *burst-read cycles***
    - » **Uses four consecutive T2 cycles**
- **Wait states**
  - **Synchronize data flow between the processor and various slower devices**
  - **Generally takes 3 clocks to transfer first quadword and 2 more for each remaining quadword**
  - **Referred to as 3-2-2-2 memory access**

Expansion Bus Controller



Simple 8-bit Output Port



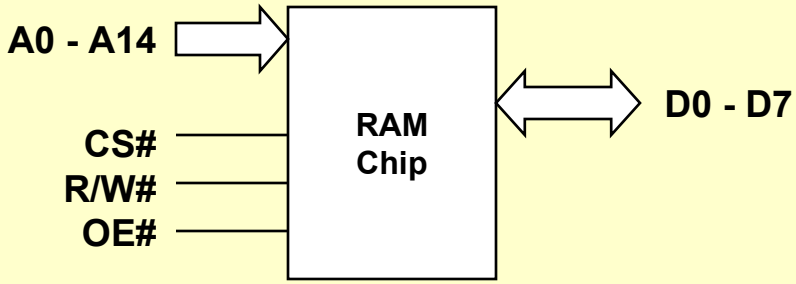
Raw and Cooked Keyboard I/O

- What do we mean by “raw” and “cooked”
- Example code (abortable puchar):

```
#include <serial.h>
#define CNTRL_C 3

int abortable_putchar(unsigned char ch) {
  while (1) {
    /* if COM2's THRE is on, output char */
    if (inpt(COM2_BASE+UART_LSR)&UART_LSR_THRE) {
      outpt(COM2_BASE+UART_TX, ch);
      return 0; }
    /* if a char is available on COM1 and it's a control-C, abort */
    if ((inpt(COM1_BASE+UART_LSR)&UART_LSR_DR) &&
        (inpt(COM1_BASE+UART_RX) == CNTRL_C))
      return -1;
  }
}
```

Static RAM (SRAM)

- Random access memory
- Simple interface and fast (10-20 nsec) but more costly (2-4X)



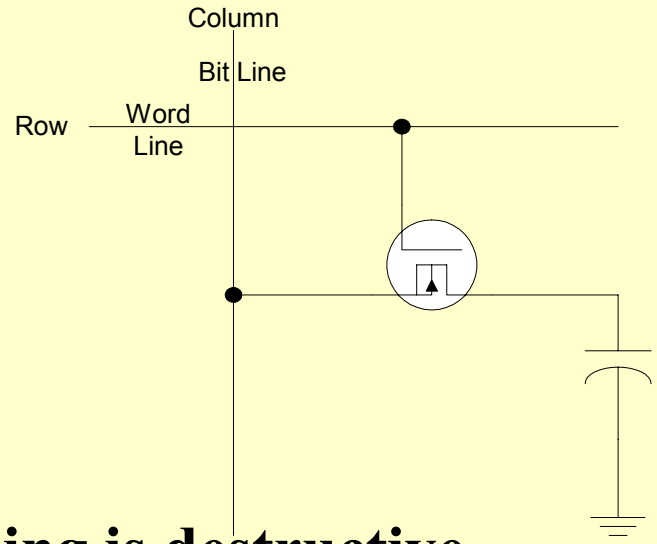
- How is the chip organized?

Read Only Memory (ROM)

- PC uses it to hold BIOS both for system and I/O adapters
- Slow (100-200 nanoseconds)
- BIOS often copied into DRAM (called *shadow* RAM)
- Various forms:
  - PROM
  - EPROM
  - EEPROM
  - Flash memory

Dynamic RAM (DRAM)

- Square array of bit cells



- Reading is destructive
- Charge leaks away in milliseconds

DRAM (Cont'd)

- For both reasons must perform a *memory refresh*
- Reading/writing on a row and column basis
- Advantages are:
  - Cells are simple
  - Uses less power
- Disadvantage:
  - Slower (20 - 30 nsec access time)
  - Total cycle time is 2X due to refresh
- Bottom line
  - 2-4X less chip area and 2-4X less power
  - Interleaving and access in column or page mode

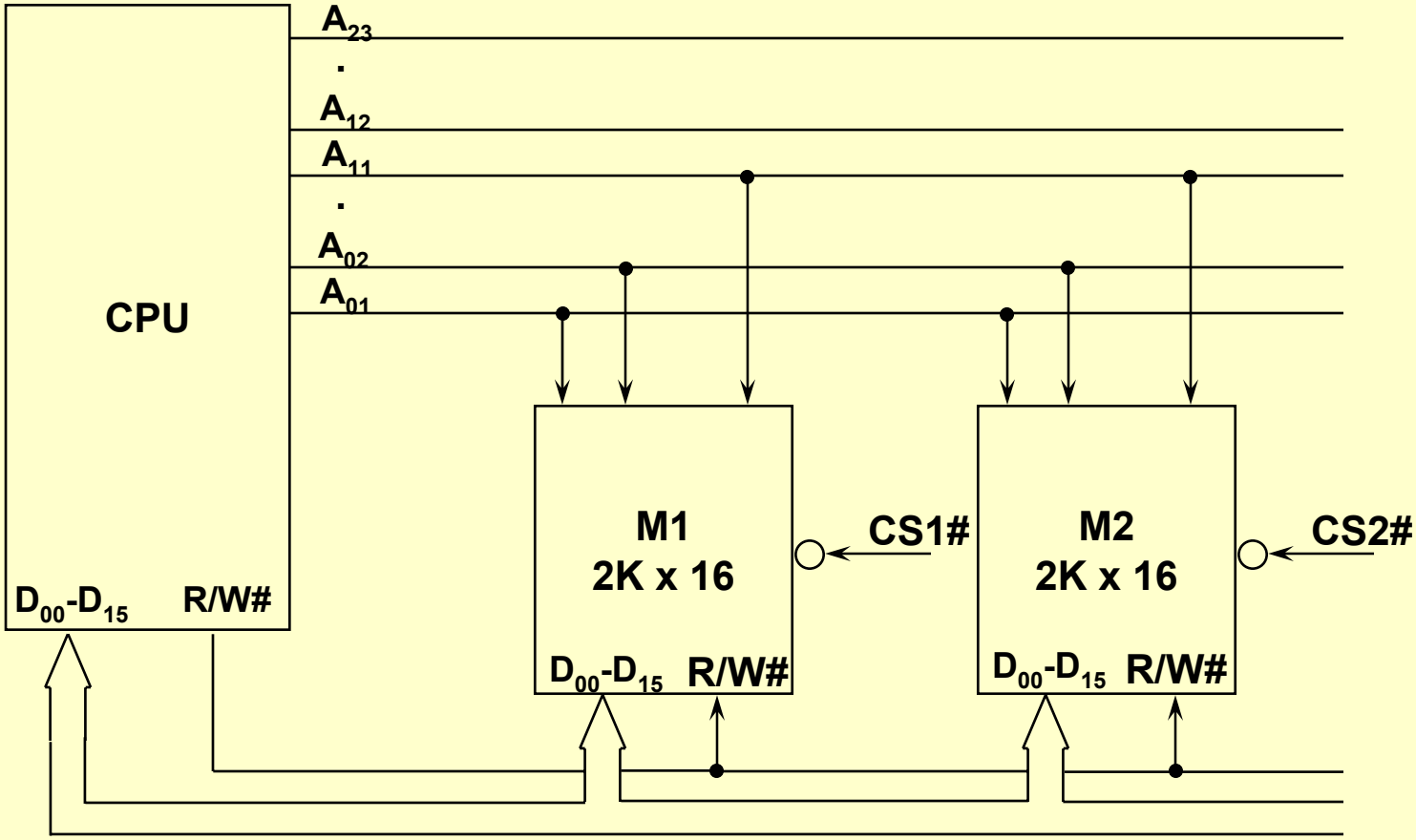
Using Commodity Parts

- You don't buy memory in 8M word chunks!
- What you do buy are identical parts that must be addressed correctly to fill the memory space
- What we use is the chip select (CS) to be a function of address and byte lines
- Sometimes other address decoding strategies are called for:
  - Full
  - Partial
  - Block

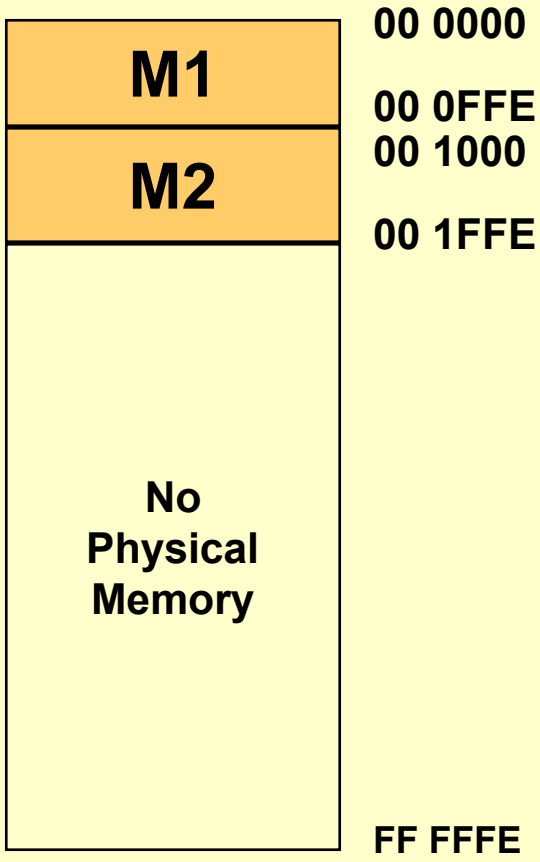
Full Address Decoding

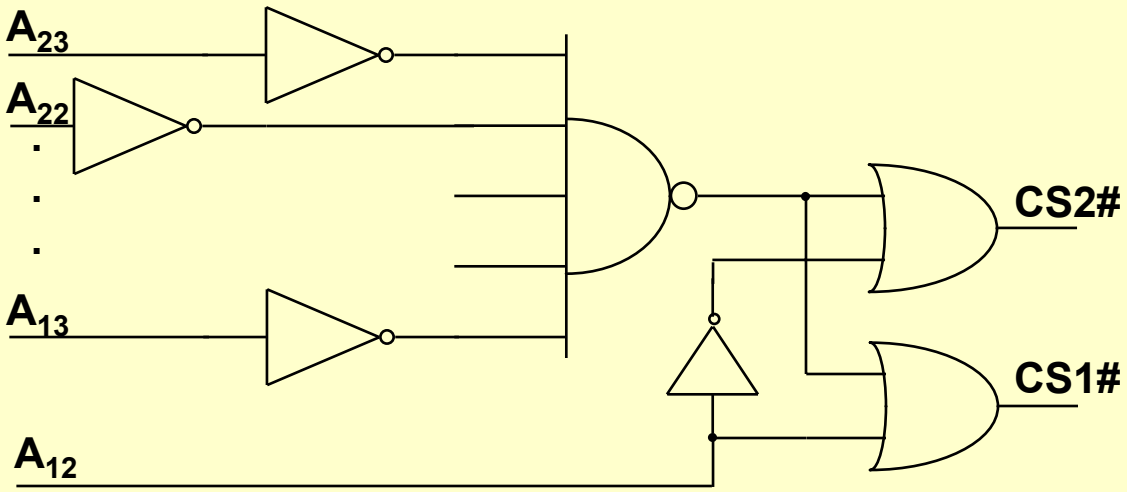
- Each addressable location within a memory component responds to only a single, unique address
- Need to use all the address lines
- May choose to not fill the address space with memory
- Can mix types of memory (RAM and ROM)

Addressing Memory Components



Memory Map


M1
M2
No
Physical
Memory
00 0000
00 0FFE
00 1000
00 1FFE
FF FFFE


A23
A22
A13
A12
CS2#
CS1#

Assigning Addresses

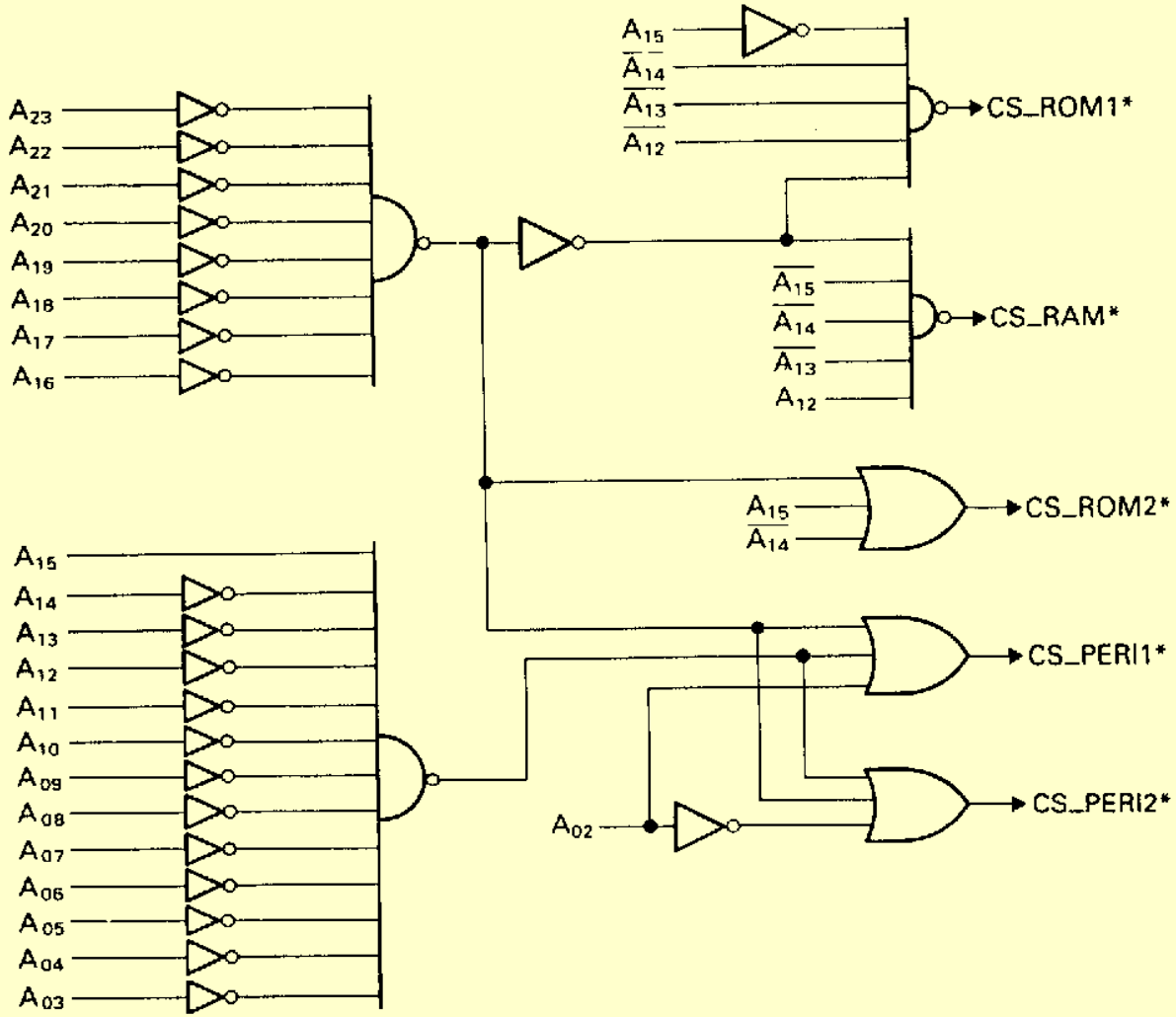
- **Suppose we are given:**
  - **10K words of ROM as one** } **2K block (called ROM1)**, **8K block (called ROM2)**
  - **2K words of RAM (called RAM)**
  - **2 words for peripheral 1 (called PERI1)**
  - **2 words for peripheral 2 (called PERI2)**
- **but arranged as:**
  - **00 0000 - 00 0FFF ROM1**
  - **00 1000 - 00 1FFF RAM**
  - **00 4000 - 00 7FFF ROM2**
  - **00 8000 - 00 8003 PERI1**
  - **00 8004 - 00 8007 PERI2**

Assigning Addresses (Cont'd)

- Note the hole from 00 2000 - 00 3FFF
- This is due to putting an 8K block of ROM on an 8K word boundary

	ADDRESS LINE																		
Device	23	22	21	20	...	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01
ROM1	0	0	0	0	...	0	0	0	0	x	x	x	x	x	x	x	x	x	x	x
RAM	0	0	0	0	...	0	0	0	1	x	x	x	x	x	x	x	x	x	x	x
ROM2	0	0	0	0	...	0	1	x	x	x	x	x	x	x	x	x	x	x	x	x
PERI1	0	0	0	0	...	1	0	0	0	0	0	0	0	0	0	0	0	0	0	x
PERI2	0	0	0	0	...	1	0	0	0	0	0	0	0	0	0	0	0	0	1	x

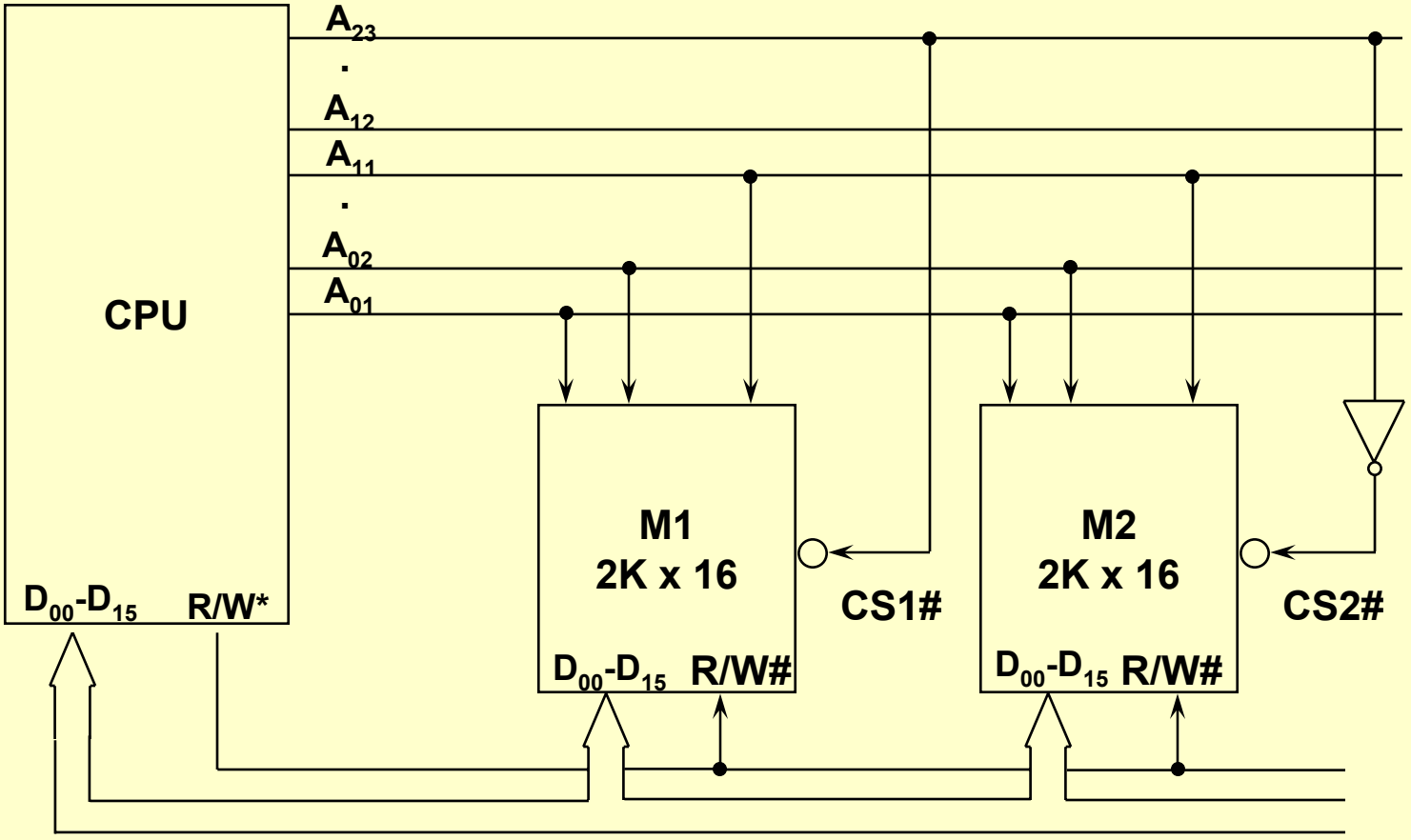
Full Address Decoding Network



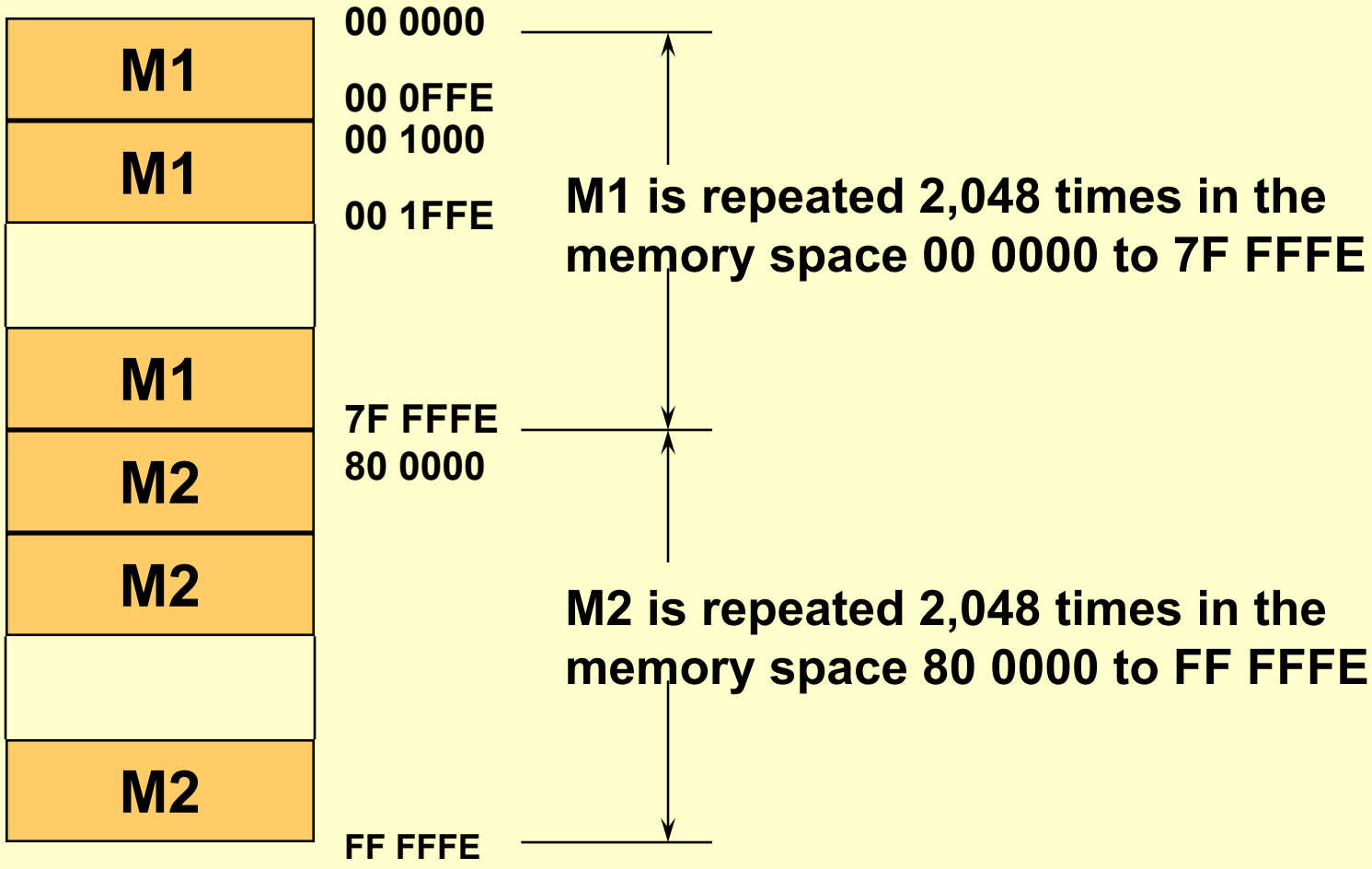
Partial Address Decoding

- Simple and inexpensive
- Not all the address lines take part in the decoding process
- The problem is that many sets of physical addresses map to the same physical memory
- Consider earlier example for full address decoding but this time using only the MSB of the address bus

Partial Address Decoding



Memory Map for Partial Decoding

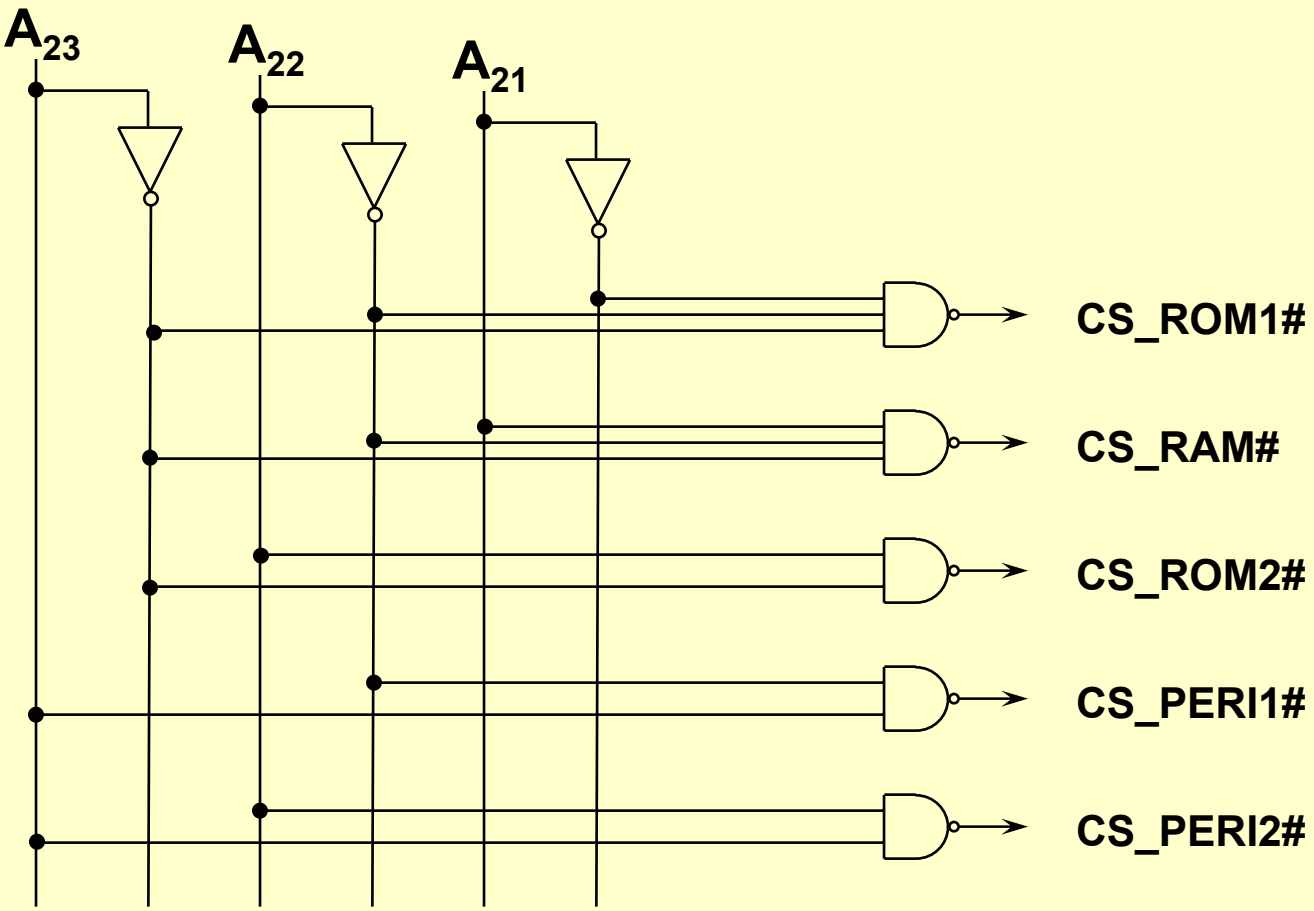


Address Table

	ADDRESS LINE																		
Device	23	22	21	20	...	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01
ROM1	0	0	0		...					x	x	x	x	x	x	x	x	x	x	x
RAM	0	0	1		...					x	x	x	x	x	x	x	x	x	x	x
ROM2	0	1			...			x	x	x	x	x	x	x	x	x	x	x	x	x
PERI1	1	0			...															x
PERI2	1	1			...															x

For this table, no further memory devices can be added the existing memory devices fill the entire 8M words of memory space. But suppose we assign A_{23} = 0 and use A_{22} to A_{20} to decode addresses; what then?

Implementing Partial Address Decoding



Block Address Decoding

- Compromise between partial and full address decoding
- Divide the memory space into a number of fully decoded blocks, generally of equal size
- Method then uses high-order bits of address to select the block and the low-order bits to select the sub-block

Discussion of mp5

- Adding new code to cmds.c but the concepts are ones already known
- Timing command are now in seconds, not microseconds, so have much longer intervals to time
- When timing interval elapsed, a “callback” routine will be initiated
- Serial interrupts added but no callback; rather want to loop COM1 characters to COM2
- Must create test routine for COM line

Overview of New Timing Commands

- **What is to be done?**
  - The timer interrupt every <interval> seconds
  - Display the number of such interrupts since timeon
  - Using a "callback" function, prints out, every now and then, based on the interval, the string (1) … (2) ...
- **New functions in cmds.c:**

```
timeon(CMD *cp, char *arguments)
```

  - Initializes a counter
  - Calls an initialization routine with callback routine

```
timeoff(CMD *cp, char *arguments)
```

  - Shuts down tick counter

What's in `tickpack.c`

- **`init_ticks` has to:**
  - Save (statically) the interval value
  - Turn off interrupts
  - Set the interrupt gate
  - Enable the PIC for the timer IRQ
  - Set up the timer down count
  - Restore interrupts
- **`irq0inthandc` has to:**
  - Acknowledge the PIC interrupt
  - Check for end of interval and if reached, execute the callback function

What's in `tickpack.c` (Cont'd)

- **`shutdown_ticks` has to:**
  - **Turn off interrupts**
  - **Disable the timer interrupt**
  - **Restore interrupts**

Most of this has already been done in mp3; look carefully at timepack_sapc.c

Overview of Serial Port Command

- `spi <dev> <on|off>`
  - `spi <dev> on` enables interrupts on input from `<dev>`
  - Echoes the character received from that serial port back to the other serial port when interrupt occurs
  - `spi <dev> off` disables those interrupt
- What code is needed?
  - Within `cmds.c` must keep track of whether serial port interrupts are on or off
    - » Might be helpful to print the mask register (0x21) and the eflags register using `get_eflags()` function
  - Have to write interrupt handlers for at least IRQ4 (COM1)

What's In Serial Port ISR?

- As an example, irq4inthandc
  - Acknowledge the PIC interrupt
  - Input the character
  - If you want an escape character to shut down the interrupts (say a '#'), then need to call `shutdown_comints` if received
  - Otherwise, need to output the character received on COM1 to the COM2 device

What's in COM Port Interrupt Package?

- **Initialize the COM port**
  - **Turn off interrupts**
  - **First, check for any characters already received so as to clear the UART buffer**
  - **Set the interrupt gate**
  - **Enable interrupts in the UART's interrupt enable register**
  - **Enable the PIC for the COM IRQ**
  - **Restore interrupts**
- **Shut down the COM port**
  - **Disable the PIC for the COM IRQ**
  - **Disable interrupts in the UART's IER**

COM Port Interrupt Package (Cont'd)

- **Last routine gets the COM status**
  - **Test the COM IER to see if it is set**
  - **Return the status**

Interrupts and Real-Time Processing

- Modern operating systems seek to improve performance through concurrent (but not simultaneous) processing
- Use interrupts to support multitasking and multiprocessing
- Kernel routines include:
  - Creation, suspension, termination, communication, and execution of processes
  - Scheduling
  - Allocation of main and secondary memory resources
  - Connection between peripheral devices and tasks/processes
  - Various system services

Requirements for Implementing a Multi-User Operating System

- **Hardware:**
  - **Two or more modes of operation (kernel, executive, system, and user)**
  - **Interrupt/exception handling**
  - **Memory management**
- **Software:**
  - **File management**
  - **IOCS**
  - **Utilities**

Hardware Protection

- Dual-Mode Operation
- I/O Protection
- Memory Protection
- CPU Protection

Dual Mode Operation

- **Sharing system resources requires operating system to ensure that an incorrect program cannot cause other programs to execute incorrectly**
- **Provide hardware support to differentiate between at least two modes of operation**
  - ***User mode* – execution done on behalf of a user**
  - ***Monitor mode* (also *supervisor mode* or *system mode*) – execution done on behalf of operating system**
- **Have to have a *mode bit***
- ***Privileged instructions* can only be issued only in monitor mode**

Memory Protection

- Must provide memory protection at least for the interrupt vector and the interrupt service routines
- But also need to make sure that memory outside the defined range for a particular user is protected
- While many methods possible, the most popular is *demand paging*
- This subject is covered in a course on operating systems

CPU Protection

- *Timer* – interrupts computer after specified period to ensure operating system maintains control
  - Timer is decremented every clock tick
  - When timer reaches the value 0, an interrupt occurs
- Timer commonly used to implement time sharing
- Timer also used to compute the current time
- Load-timer is a privileged instruction

General-System Architecture

- Given that I/O instructions are privileged, how does the user program perform I/O?
- System call – the method used by a process to request action by the operating system
  - Usually takes the form of a trap to a specific location in the interrupt vector.
  - Control passes through the interrupt vector to a service routine in the OS, and the mode bit is set to monitor mode
  - The monitor verifies that the parameters are correct and legal, executes the request, and returns control to the instruction following the system call

Protection System

- Protection refers to a mechanism for controlling access by programs, processes, or users to both system and user resources
- The protection mechanism must:
  - Distinguish between authorized and unauthorized usage
  - Specify the controls to be imposed
  - Provide a means of enforcement

Operating System Services

- Program execution – system capability to load a program into memory and to run it
- I/O operations –the operating system must provide some means to perform I/O
- File-system manipulation – program capability to read, write, create, and delete files
- Communications – exchange of information between processes executing either on the same computer or on different systems tied together by a network
- Error detection – detecting errors in the CPU and memory hardware, in I/O devices, or in user programs

System Calls

- Can consider this as the way a process invokes the services of an operating system
- We write programs and programs include system calls (read/write, get time, exit a program, ...)
- Sometimes we make special requests (spawn or fork a new process, get more main memory, wait on external event, ...)
- Can categorize system calls based on intent

System Programs

- System programs - convenient environment for program development and execution and include:
  - File manipulation
  - Status information
  - File modification
  - Programming-language support
  - Program loading and execution
  - Communications
  - Application programs
- Most users' view of the operation system is defined by system programs, not the actual system calls

Simple Approach

- **MS-DOS – written to provide the most functionality in the least space**
  - **Not divided into modules**
  - **Although MS-DOS has some structure, its interfaces and levels of functionality are not well separated**
- **UNIX – limited by hardware functionality and original OS had limited structuring consisting of two separable parts:**
  - **Systems programs**
  - **The kernel**
    - » **Consisted of everything below the system-call interface and above the physical hardware**
    - » **Provided all operating-system functions; a large number of functions for one level.**