The MIPS "RISC" Architecture

- Few, simple instructions
- Load/store architecture (move in/out of registers, rather than to/from memory)
- Simple addressing
- Why RISC?
  - Optimize speed of functions that get used most
  - Small processor design (rapid design cycle)
  - Rely on smart compilers to produce good code!
- MIPS Registers (all are 32-bit)
  - general purpose: r0 - r31 (but r0 always contains 0)
  - these may be given other names, depending on usage conventions. (e.g., r29 ↔ *sp*)

MIPS Registers, continued

- LO / HI for fixed-point multiplication/division
- Floating-point registers, f0 - f31
- PC = program counter
- t0 - t9 (temporaries. Use them at will, but subroutines may clobber them)
- s0 - s7 (saved registers. Well-behaved subroutines will not permanently change them.)
- a0 - a3 (argument registers. Up to 4 subroutine arguments go here.)
- v0 - v1 (Subroutines return values here.)
- sp (stack pointer)
- ra (return address of subroutine)
- +5 others

MIPS Registers: the Full Story

Register name	Register #	Usage
$zero	0	constant 0
$at	1	reserved for assembler
$v0	2	expr eval & function result
$v1	3	expr eval & function result
$a0	4	argument 1 to a function
$a1	5	argument 2 to a function
$a2	6	argument 3 to a function
$a3	7	argument 4 to a function
$t0	8	temporary (not preserved)

MIPS Registers, continued

Register name	Register #	Usage
$t1	9	temporary (not preserved)
$t2	10	temporary (not preserved)
$t3	11	temporary (not preserved)
$t4	12	temporary (not preserved)
$t5	13	temporary (not preserved)
$t6	14	temporary (not preserved)
$t7	15	temporary (not preserved)
$s0	16	saved temporary (preserved)
$s1	17	saved temporary (preserved)
$s2	18	saved temporary (preserved)

MIPS Registers, continued

Register name	Register #	Usage
$s3	19	saved temporary (preserved)
$s4	20	saved temporary (preserved)
$s5	21	saved temporary (preserved)
$s6	22	saved temporary (preserved)
$s7	23	saved temporary (preserved)
$t8	24	temporary (not preserved)
$t9	25	temporary (not preserved)
$k0	26	reserved for OS kernel
$k1	27	reserved for OS kernel
$gp	28	pointer to global area

MIPS Registers, continued

Register name	Register #	Usage
$sp	29	stack pointer
$fp	30	frame pointer
$ra	31	return address (to caller)

Note: MIPS also has 32 *floating-point* registers. Two registers are paired for double precision numbers. Odd numbered registers cannot be used for arithmetic or branching, just as part of a double precision register pair.

Two Instructions

- **ADD**
  - e.g., **add $t2, $t0, $t1** *# (t2) <— (t0) + (t1)*
  - First 2 operands must be registers, but third can be "immediate" data — stored in the instruction
  - e.g., **add $t2, $t0, 64** *# (t2) <—(t0) + 64*
  - there is an actual **ADDI** instruction, used by the assembler when the 3rd arg to **ADD** is a 16-bit constant.
- LI (load immediate)
  - e.g., **li $t0, 176** *# (t0) <— 176*
  - stores an integer into a register
  - this is an example of a "pseduo-instruction". The assembler translates this into real instructions.

Assembly Language Programming

- **Adding 1 + 2**

```
main:                    # SPIM starts execution at main.
        li    $t1, 1     # load 1 into $t1.
        li    $t2, 2     # load 2 into $t2.
        add   $t0, $t1, $t2   # $t0 <- $t1 + $t2.
        li    $v0, 10    # syscall code 10 is for exit
        syscall          # make the syscall
```

- **SYSCALLs**
  - are calls to support routines that do I/O, exit gracefully, etc. Code for function to be performed is passed in $v0, e.g.
    - **code 5 = read integer into $v0**

Service	System call code	Arguments	Result
print_int	1	$a0 = integer	
print_float	2	$f12 = float	
print_double	3	$f12 = double	
print_string	4	$a0 = string	
read_int	5		integer (in $v0)
read_float	6		float (in $f0)
read_double	7		double (in $f0)
read_string	8	$a0 = buffer, $a1 = length	
sbrk	9	$a0 = amount	address (in $v0)
exit	10		
print_char	11	$a0 = char	
read_char	12		char (in $a0)
open	13	$a0 = filename (string), $a1 = flags, $a2 = mode	file descriptor (in $a0
read	14	$a0 = file descriptor, $a1 = buffer, $a2 = length	num chars read (in $a0)
write	15	$a0 = file descriptor, $a1 = buffer, $a2 = length	num chars written (in $a0)
close	16	$a0 = file descriptor	
exit2	17	$a0 = result	

Add2.asm (sum of 2 integers)

```
main:   #       $t0 and $v0     - used to hold the 2 numbers.
        ## Get first number from user, put into $t0.
        li      $v0, 5          # load syscall read_int into $v0.
        syscall                 # make the syscall.
        move  $t0, $v0          # move the number read
        ## Get second number from user, leave it in $v0.
        li      $v0, 5          # load syscall read_int into $v0.
        syscall                 # make the syscall.
        add     $a0, $t0, $v0   # compute the sum.
        li      $v0, 1          # load syscall print_int into $v0.
        syscall                 # make the syscall.
        li      $v0, 10         # syscall code 10 is for exit.
        syscall                 # make the syscall.
```

More Instructions

- To move data between registers and memory:
  - **LW / LH / LB** — load word / halfword / byte
  - **SW / SH / SB** — store word / halfword/ byte
- LA — load address (e.g., LA $t0, labelOfData)
  - puts 32-bit address of label in $t0. In practice, labelOfData assembles to something like 4097($at), so actual address cannot be determined till runtime
- Arithmetic / Logical Instructions
  - ADD and SUB
  - MUL $rd, $rs1, $rs2 vs. MULT $r1, $r2
  - DIV $r1, $r2
  - AND, OR, XOR, NOT
  - SLL, SRL, SRA

Directives

- A C-style string, null-terminated, and produced by
  - `hello_msg:  .asciiz "Hello\n"`
- **.byte 0xFF** *#produces one byte = 1111 1111*
- **.ascii "ABC"** *# produces 3 bytes with ASCII codes*
- **.text** *# assemble what follows into "program space,"*
- **.data** *# vs. assemble what follows into "data space"*
- Assembler directives begin with "."
- Printing an asciiz string (see hello.asm)

```
        .text
        la    $a0, hello_msg
        li    $v0, 4
        syscall
```

Branch Instructions

- `b lab` *# unconditional branch to instruction lab*
- **`beq, bne, blt, bgt, ble, bge`**
  - e.g., bgez $r, lab #branch if $r ≥0
  - Note: lab is actually converted to a 16-bit offset by assembler — so branches are limited to "distances" of about ± 2^{15} words (added to PC)
- Unsigned Comparisons
  - **e.g., compare** `1111 … 1111` to `0000 … 0000`
  - **`bgeu, bgtu, bleu, bltu`**
- Miscellaneous Data Movement
  - **`move`** $rd, $rs *# like LW, but for registers*
  - **`mfhi`** $rd *# also MTHI, MFLO, MTLO*

Decision-Making

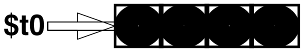
- **IF ($t0 < $t1) THEN *do-first* else *do-second***

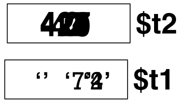
```
          …
          bge    $t0, $t1, to_ge
          # Here only if ($t0) < ($t1)
          # … code for do-first …
          b      end_if_code
to_ge:    # Here only if ($t0) ≥ ($t1)
          # … code for do-second …
          # 2 forks merge back together
end_if_code:
```

- Example: Use a "moving pointer" to find the length of an ASCIIZ string stored at **the_string**
- Example: Check if a string is a "palindrome"

Converting a String to a Number

- Assume $t0 points to a string of one or more decimal digits:





- Algorithm:

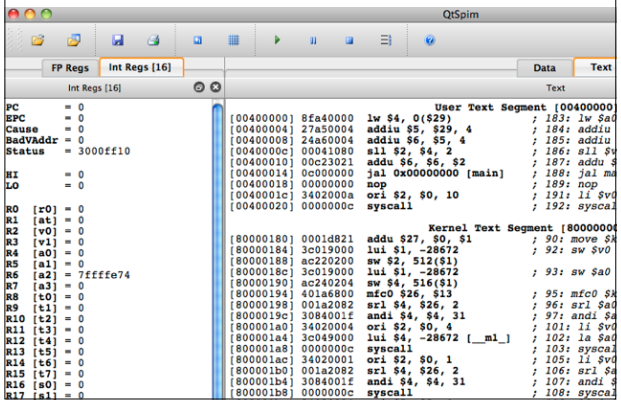
```
$t2 <— 0
$t1 <- ($t0);  $t0 ++
$t1 < '0'  OR  > '9'  —> Done!
$t2 <- $t2 * 10 + ($t1 - '0' )
```

- See program **atoi-1.asm**. This program does NOT handle negative numbers; produces 0 result if there are no digits; does not check overflow; does not check for illegal input characters!

Addressing Memory

- MIPS is a "load-store" architecture.
  - Bare machine allows only one addressing mode, **constant ($register)**
- However, SPIM (the virtual machine) allows all of the following, along with "pseudo-instructions," such as **abs**
  - **($register)**
  - **a constant**
  - **a symbolic label**
  - **a symbolic label ± a constant**
  - **a symbolic label ± a constant ($register)**

Using QtSpim



Detecting Overflow

- Multiplication of two 32-bit integers produces a 64-bit product in the special registers **HI** and **LO**
- The **MUL** "instruction" expands into more than one instruction that can produce the right result if it can be represented in 32 bits.
- Use the underlying **MULT** instruction to check for a possible overflow condition:

```
li    $t4, 10          #no immediate MULT
mult  $t2, $t4
mfhi  $t5
bnez  $t5, overflow
mflo  $t2
bltz  $t2, overflow    #product is in $t2
```

Subroutine Linkage

- The environment of a function includes
  - **values of the function (parameters/arguments)**
  - **values of local variables**
  - **a record of where to return to when this activation of the function completes**
- The environment is kept on the stack in a block called the *stack frame* (one per activation)
- Example:
  - `f1 ( … )`
    `{ … f2(…) … }`
  - `f2 ( … )`
    `{ … f1( … ) … }`

Euclid's Algorithm as Subroutine (see Euclid.asm)

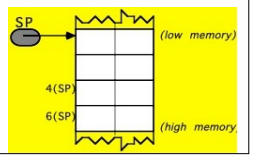
```
gcd:   div    $a0, $a1    # divide $a0 by $a1
       mfhi   $t0         # the remainder is in HI - get it to check
       beqz   $t0, done   # if the remainder is 0, you're done!  Go
       move   $a0, $a1    # if the remainder is not 0, put $a1 into
       move   $a1, $t0    # put the remainder into $a1
       b      gcd         # go back to step 1 (div)

done:  move   $v0, $a1    # we have the answer - put it in $v0 to r
       jr     $ra         # jump back to main
```


Pointers & Offset/Displacement Addressing for LOADs & STOREs

- lw $rt, 16-bit-offset($rb)
  - sign-extend offset to 32 bits; then add it to
  - the value in $rb to give address of word to be moved
  - Note: $rb is typically $sp (stack pointer), or $at (a global pointer, reserved for use by the assembler)
- The *Stack* grows from larger addresses to smaller ones

Global space is pre-allocated as a result of calculations made at assembly time.



Register Conventions

- $t0 - $t9 are CALLER-saved; if the caller wants them preserved, it must do the saving (in its stack frame) since the callee is under no obligation to preserve them.
- $s0 - $s7 are CALLEE-saved; every subroutine is obliged to preserve them. So a routine that wants to use them MUST save them on entry, and restore them on exit.
- Caller-Callee Protocol
  - CALLER preparation for calling
    - put parameters into $a0 - $a3
    - if caller needs any of $t0 - $t9 saved, save them in caller stack frame
    - execute a `jal CALLEE`, which puts the address after the instruction itself into `$ra`

Caller-Callee Protocol, continued

- CALLEE preamble
  - Create stack frame
    - $sp <— $sp - <frame size>
  - Save in the frame
    - $fp
    - any of $s0 - $s7 used by callee (known at compiling/coding time)
    - $ra, unless this routine does not call any other (leaf routine)
  - $fp <— $sp + <frame size> # $fp is typically old $sp
- BODY of CALLEE
  - local variables are referred to as displacements off $sp — i.e., positions within the "frame"

Caller-Callee Protocol, continued

- CALLEE return preparation
  - $v0 <— <return value>
  - restore callee-saved registers from frame ($fp, $s0-$s7, $ra)
  - restore stack pointer: $sp <— $sp + <frame size>
  - return using JR $ra
- CALLER cleanup
  - restore caller-saved registers ($t0-$t9), if any
  - return value is in $v0
- Understanding RECURSION: see fib-s.asm
- Note: A function that requires additional temporary storage of size that is not known till runtime (e.g., a sequence of input characters), can use the stack.

Recursive Fibonacci

```
        move $s0, $a0
        blt  $s0, 2, fib_base_case     # if n < 2 ...
        sub  $a0, $s0, 1       # compute fib (n - 1)
             jal    fib
r1:     move $s1, $v0          # s1 <- fib (n - 1).
        sub  $a0, $s0, 2       # compute fib (n - 2)
             jal    fib
r2:     move $s2, $v0          # $s2 <- fib (n - 2).
             add    $v0, $s1, $s2   # fib (n -1) + fib (n -2)
        b    fib_return
fib_base_case:     li     $v0, 1
fib_return:        ...              # deal with stack frame
                   jr  $ra
```


Arithmetic and Logical Instructions

Absolute value

`abs rdest, rsrc` *pseudoinstruction*

Put the absolute value of register `rsrc` in register `rdest`.

Addition (with overflow)

`add rd, rs, rt`

0	rs	rt	rd	0	0x20
6	5	5	5	5	6

Addition (without overflow)

`addu rd, rs, rt`

0	rs	rt	rd	0	0x21
6	5	5	5	5	6

Put the sum of registers `rs` and `rt` into register `rd`.

Addition immediate (with overflow)

`addi rt, rs, imm`

8	rs	rt	imm
6	5	5	16

`add $t2, $t2, $t1`

is assembled into

`add $10, $10, $9`

and is of the form

`add  rd, rs, rt`

0	rs	rt	rd	0	0x20
6	5	5	5	5	6

0000 0001 0100 1001 0101 0000 0010 0000

0 1 4 9 5 0 2 0

Other Instruction Formats

Branch on equal

```
beq rs, rt, label
```

4	rs	rt	Offset
6	5	5	16

Jump Instructions

Jump

```
j target
```

2	target
6	26

Unconditionally jump to the instruction at target.

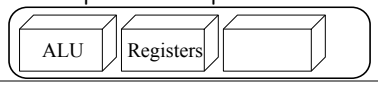
Jump and link

```
jal target
```

3	target
6	26

RISC vs CISC

- As VLSI improvements made it possible to squeeze more components onto a processor chip, more complex instruction sets became possible.
  - e.g., The VAX's POLY X, D, ADDR instruction (the description of this takes 4 pages in the manual)
- Such complex instructions are not implemented by custom hardware, but by programs in a special language called microcode that controls movement and operations on data in the internal processor registers.
  - Microcode ("firmware") is prepared "at the factory" and frozen into the processor chip.



From the VAX Machine Architecture Manual

Integer and Floating Point Instructions

POLY

POLYNOMINAL EVALUATION

Purpose: allows fast calculation of math functions

Format: opcode arg.rx, degree.rw, tbladdr.ab

Operation:

```
tmp1 ← degree;
if tmp1 GRTU 31 then RESERVED OPERAND EXCEPTION;
tmp2 ← tbladdr;
tmp3 ← [(tmp2);          !tmp3 accumulates the
                         partial result
                         !tmp3 is of type X
if POLYH then −(SP) ← arg;
tmp4 ← 0;                !underflow flag for
                         original 11/780
while tmp1 GRTU 0 do
   begin                 !computation loop
   tmp5 ← [arg * tmp3];  !tmp5 accumulates new partial
                         result.
                         !tmp3 has old partial result.
```

The VAX POLY Instruction, continued

Integer and Floating Point Instructions

```
    if FU EQL 1 then tmp4 ← 1;
                        !set underflow flag
                        (original 11/780)
    end;
  tmp1 ← tmp1 - 1;
  tmp2 ← tmp2 + [size of data type];
  tmp3 ← tmp5;
                        !update partial result in tmp3
  end;
if POLYF then
  begin
  R0 ← tmp3;
  R1 ← 0;
  R2 ← 0;
  R3 ← tmp2;
  end;
if POLYD or POLYG then
  mbegin
  R1'R0 ← tmp3;
  R2 ← 0;
  R3 ← tmp2;
```

Integer and

Description: The table addr
coefficients; the
polynomial is p
table is specifie
creasing addres
same as the data

Evaluation is ca
of R0 (R1'R0 for
are replaced by
result = C[0]
degree and x =

The unsigned v
numbered coeff
requires four lo
instruction is inte

Notes: 1. After execu
POLYF
R0 = result
R1 = 0
R2 = 0

Problems with CISC

- Complex design
- Complex instructions are very rarely used
  - in part, because compilers don't want them
- Full instruction set must be maintained in new CPUs
- The microcode describes use of multiple registers, arithmetic units, etc. But most of the time, these functional units are idle!
- Cycles vs. Instructions
  - Each physical processor has a "cycle time" = the time for the smallest indivisible register transfer, arithmetic (etc.) operation. 2.5 gHz = 2.5 billion cycles/sec.
  - An instruction typically takes several cycles to execute completely. CISC = many cycles per instr.

Making a Fast RISC Machine

- Fast clock rate (time/cycle)
- Low cycles per instruction (cycle/inst)
  - Streamlining the the hardware can improve the CPI by factors of 5 or more
- Relatively low instruction counts (inst/task), optimizing compiler technology
- "Pipelining" to improve effective cycles/instr.
  - Goal: To keep as much of the processor busy at the same time as possible
  - First: Keep instruction types simple, so there are only a few different "ways" instructions work
    - **e.g., except for MUL/DIV, all MIPS arithmetic/logical instructions work on 3 registers, or 2 registers plus immediate data**

Pipelining

- Second: Execute different "parts" of several instructions simultaneously
- Hypothetical, simple, non-pipelined CPU:
  - `IF ID EX MEM WB`
    `IF ID EX MEM WB`
  - Each instruction takes 5 "clock ticks"
  - IF = instruction fetch from cache
  - ID = instruction decode
  - EX = execute (arithmetic, etc.)
  - MEM = memory operation (load/store)
  - WB = write back (changes to register, if any)
- With RISC, you can get 5 (or more) instructions in progress at once (R4000).

Pipelining, continued

Single-Issue, 5-Stage RISC Pipeline

Instruction	Pipeline Stages							
1	IF	ID	EX	MEM	WB			
2		IF	ID	EX	MEM	WB		
3			IF	ID	EX	MEM	WB	
4				IF	ID	EX	MEM	WB
5					IF	ID	EX	MEM	WB

5 (or more) instructions in progress at once (R4000)

Typical of early & current RISCs; some used 4 stages

Attempt to issue (start) 1 instruction per cycle
Current & new systems: start 2 or more, sometimes!

Cycles/instruction = 1 (at best), 1.2-1.6 more typical

Instruction Scheduling

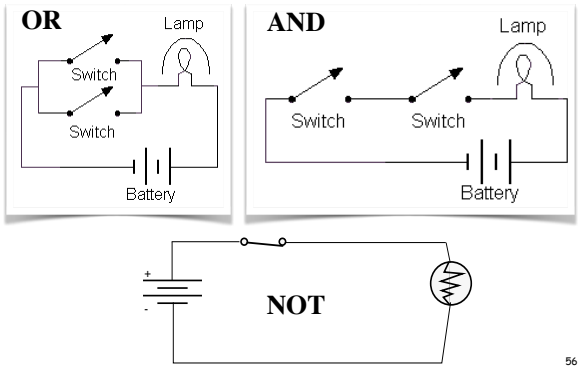
- What if the result from one instruction is used by the next instruction, but the first instruction isn't done?
  - Hardware stall / interlock
  - Insert "NO-OP"s in between to slow things down;
- Example: The instruction after a LOAD cannot use the result of that LOAD; similarly, the instruction after a branch is executed even if the branch is taken. For example,

```
      lw   $R7,  X
?????            #delay slot
add  $R7, $R7, 1
```

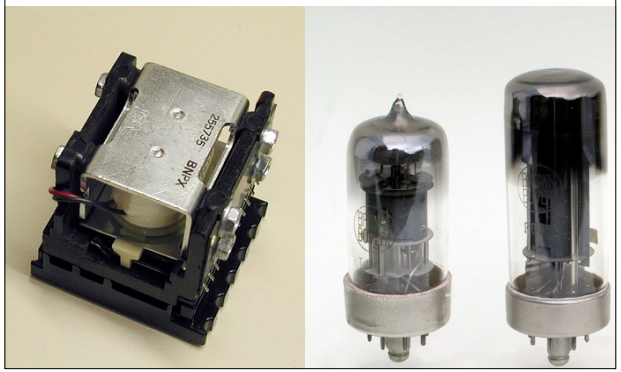
Reorder the Instructions!

- a = b+ 1; if (c == 0) d = 0;
  - lw $r2, b
  - nop
  - add $r2, $r2, 1
  - sw $r2, a # a = b + 1
  - lw $r3, c # r3 = c
  - nop
  - bne $r3, zero, lab # if (c != 0) skip
  - nop
  - sw $zero, d # if (c == 0), then d = 0
  - lab:

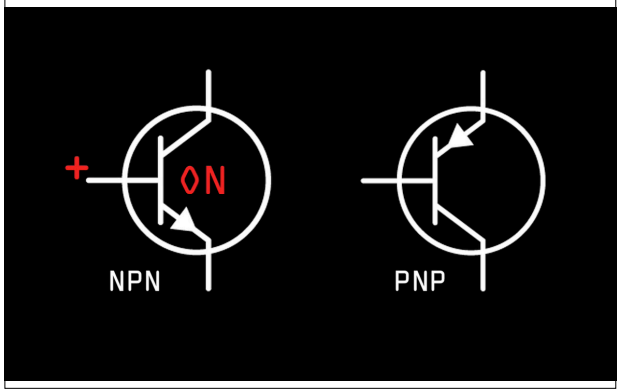
Switching Circuits



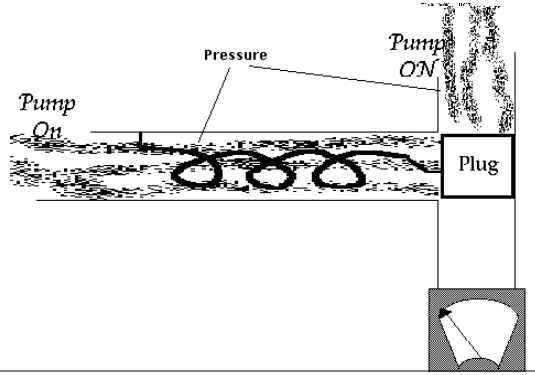
Electromechanical Relays and Vacuum Tubes



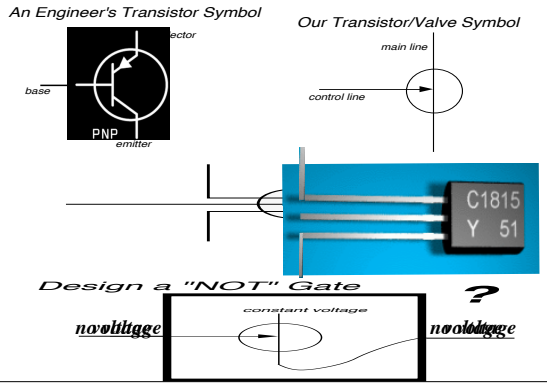
Transistors: Electronic Switches



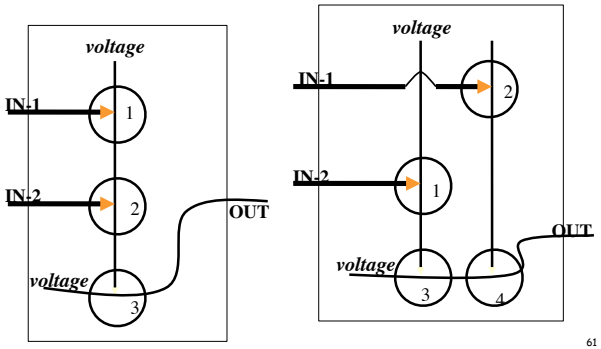
How a Transistor Works



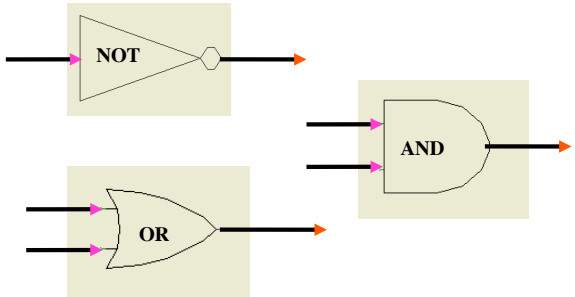
Transistor Logic Gate Circuits



"OR" and "AND" Gates

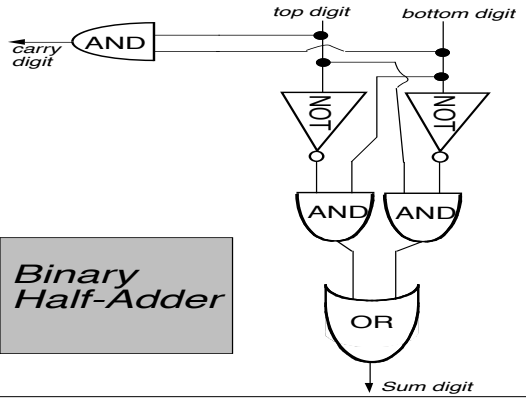


Representing the Gates



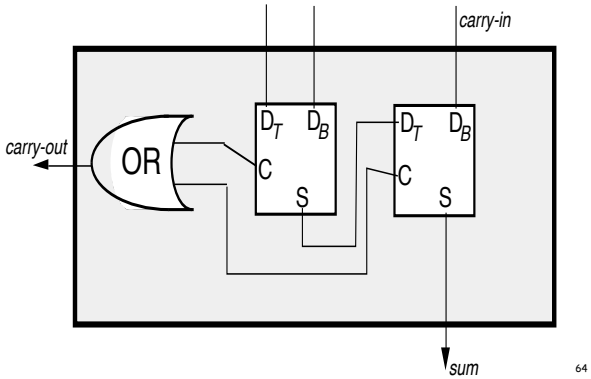
See **http://math.hws.edu/TMCM/java/labs/xLogicCircuitsLab1.html**

Binary Half-Adder



Binary Half-Adder

Full Adder



Circuit to Add 7-bit Numbers