HDL Example 7.11 2:1 MULTIPLEXER

Verilog

```
module mux2 #(parameter WIDTH = 8)
             (input  [WIDTH-1:0] d0, d1,
              input              s,
              output [WIDTH-1:0] y);

  assign y = s ? d1 : d0;
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity mux2 is -- two-input multiplexer
  generic(width: integer);
  port(d0, d1: in  STD_LOGIC_VECTOR(width-1 downto 0);
       s:      in  STD_LOGIC;
       y:      out STD_LOGIC_VECTOR(width-1 downto 0));
end;

architecture behave of mux2 is
begin
  y <= d0 when s = '0' else d1;
end;
```

7.6.3 Testbench

The MIPS testbench loads a program into the memories. The program in Figure 7.60 exercises all of the instructions by performing a computation that should produce the correct answer only if all of the instructions are functioning properly. Specifically, the program will write the value 7 to address 84 if it runs correctly, and is unlikely to do so if the hardware is buggy. This is an example of *ad hoc* testing.

```
# mipstest.asm
# David_Harris@hmc.edu 9 November 2005
#
# Test the MIPS processor.
# add, sub, and, or, slt, addi, lw, sw, beq, j
# If successful, it should write the value 7 to address 84

#           Assembly              Description              Address    Machine
main:       addi $2, $0, 5        # initialize $2 = 5      0          20020005
            addi $3, $0, 12       # initialize $3 = 12     4          2003000c
            addi $7, $3, -9       # initialize $7 = 3      8          2067fff7
            or   $4, $7, $2       # $4 <= 3 or 5 = 7       c          00e22025
            and  $5, $3, $4       # $5 <= 12 and 7 = 4     10         00642824
            add  $5, $5, $4       # $5 = 4 + 7 = 11        14         00a42820
            beq  $5, $7, end      # shouldn't be taken     18         10a7000a
            slt  $4, $3, $4       # $4 = 12 < 7 = 0        1c         0064202a
            beq  $4, $0, around   # should be taken        20         10800001
            addi $5, $0, 0        # shouldn't happen       24         20050000
around:     slt  $4, $7, $2       # $4 = 3 < 5 = 1         28         00e2202a
            add  $7, $4, $5       # $7 = 1 + 11 = 12       2c         00853820
            sub  $7, $7, $2       # $7 = 12 - 5 = 7        30         00e23822
            sw   $7, 68($3)       # [80] = 7               34         ac670044
            lw   $2, 80($0)       # $2 = [80] = 7          38         8c020050
            j    end              # should be taken        3c         08000011
            addi $2, $0, 1        # shouldn't happen       40         20020001
end:        sw   $2, 84($0)       # write adr 84 = 7       44         ac020054
```

Figure 7.60 Assembly and machine code for MIPS test program

```
20020005
2003000c
2067fff7
00e22025
00642824
00a42820
10a7000a
0064202a
10800001
20050000
00e2202a
00853820
00e23822
ac670044
8c020050
08000011
20020001
ac020054
```

Figure 7.61 Contents of `memfile.dat`

The machine code is stored in a hexadecimal file called `memfile.dat` (see Figure 7.61), which is loaded by the testbench during simulation. The file consists of the machine code for the instructions, one instruction per line.

The testbench, top-level MIPS module, and external memory HDL code are given in the following examples. The memories in this example hold 64 words each.

HDL Example 7.12 MIPS TESTBENCH

Verilog

```
module testbench();

  reg      clk;
  reg      reset;

  wire [31:0] writedata, dataadr;
  wire        memwrite;

  // instantiate device to be tested
  top dut(clk, reset, writedata, dataadr, memwrite);

  // initialize test
  initial
    begin
      reset <= 1; # 22; reset <= 0;
    end

  // generate clock to sequence tests
  always
    begin
      clk <= 1; # 5; clk <= 0; # 5;
    end

  // check results
  always @ (negedge clk)
    begin
      if (memwrite) begin
        if (dataadr === 84 & writedata === 7) begin
          $display ("Simulation succeeded");
          $stop;
        end else if (dataadr !== 80) begin
          $display ("Simulation failed");
          $stop;
        end
      end
    end
endmodule
```

VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.all; use IEEE.STD_LOGIC_UNSIGNED.all;
entity testbench is
end;

architecture test of testbench is
  component top
    port(clk, reset:         in  STD_LOGIC;
         writedata, dataadr: out STD_LOGIC_VECTOR(31 downto 0);
         memwrite:           out STD_LOGIC);
  end component;
  signal writedata, dataadr:   STD_LOGIC_VECTOR(31 downto 0);
  signal clk, reset, memwrite: STD_LOGIC;

begin
  -- instantiate device to be tested
  dut: top port map(clk, reset, writedata, dataadr, memwrite);

  -- Generate clock with 10 ns period
  process begin
    clk <= '1';
    wait for 5 ns;
    clk <= '0';
    wait for 5 ns;
  end process;

  -- Generate reset for first two clock cycles
  process begin
    reset <= '1';
    wait for 22 ns;
    reset <= '0';
    wait;
  end process;

  -- check that 7 gets written to address 84
  -- at end of program
  process (clk) begin
    if (clk'event and clk = '0' and memwrite = '1') then
      if (conv_integer(dataadr) = 84 and conv_integer
          (writedata) = 7) then
        report "Simulation succeeded";
      elsif (dataadr /= 80) then
        report "Simulation failed";
      end if;
    end if;
  end process;
end;
```

HDL Example 7.13 MIPS TOP-LEVEL MODULE

Verilog

```
module top(input         clk, reset,
           output [31:0] writedata, dataadr,
           output        memwrite);

  wire [31:0] pc, instr, readdata;

  // instantiate processor and memories
  mips mips(clk, reset, pc, instr, memwrite, dataadr,
            writedata, readdata);
  imem imem(pc[7:2], instr);
  dmem dmem(clk, memwrite, dataadr, writedata,
            readdata);
endmodule
```

VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.all; use IEEE.STD_LOGIC_UNSIGNED.all;
entity top is -- top-level design for testing
  port(clk, reset:          in     STD_LOGIC;
       writedata, dataadr:  buffer STD_LOGIC_VECTOR(31 downto
                                                    0);
       memwrite:            buffer STD_LOGIC);
end;

architecture test of top is
  component mips
    port(clk, reset:        in  STD_LOGIC;
         pc:                out STD_LOGIC_VECTOR(31 downto 0);
         instr:             in  STD_LOGIC_VECTOR(31 downto 0);
         memwrite:          out STD_LOGIC;
         aluout, writedata: out STD_LOGIC_VECTOR(31 downto 0);
         readdata:          in  STD_LOGIC_VECTOR(31 downto 0));
  end component;
  component imem
    port(a:  in  STD_LOGIC_VECTOR(5 downto 0)
         rd: out STD_LOGIC_VECTOR(31 downto 0));
  end component;
  component dmem
    port(clk, we: in  STD_LOGIC;
         a, wd:   in  STD_LOGIC_VECTOR(31 downto 0);
         rd:      out STD_LOGIC_VECTOR(31 downto 0));
  end component;
  signal pc, instr,
         readdata: STD_LOGIC_VECTOR(31 downto 0);
begin
  -- instantiate processor and memories
  mips1: mips port map(clk, reset, pc, instr, memwrite,
                       dataadr, writedata, readdata);
  imem1: imem port map(pc(7 downto 2), instr);
  dmem1: dmem port map(clk, memwrite, dataadr, writedata,
                       readdata);
end;
```

HDL Example 7.14 MIPS DATA MEMORY

```
module dmem(input         clk, we,
            input  [31:0] a, wd,
            output [31:0] rd);

  reg [31:0] RAM[63:0];

  assign rd = RAM[a[31:2]]; // word aligned

  always @(posedge clk)
    if(we)
      RAM[a[31:2]] <= wd;
endmodule
```

```
library IEEE;
use IEEE.STD_LOGIC_1164.all; use STD.TEXTIO.all;
use IEEE.STD_LOGIC_UNSIGNED.all; use IEEE.STD_LOGIC_ARITH.all;
entity dmem is -- data memory
 port(clk, we: in  STD_LOGIC;
      a, wd:   in  STD_LOGIC_VECTOR(31 downto 0);
      rd:      out STD_LOGIC_VECTOR(31 downto 0));
end;

architecture behave of dmem is
begin
  process is
    type ramtype is array(63 downto 0) of STD_LOGIC_VECTOR
                          (31 downto 0);
    variable mem: ramtype;
  begin
    -- read or write memory
    loop
      if clk'event and clk = '1' then
          if(we = '1') then mem(CONV_INTEGER(a(7 downto
                                             2))) := wd;
          end if;
      end if;
      rd <= mem(CONV_INTEGER(a(7 downto 2)));
      wait on clk, a;
    end loop;

  end process;
end;
```

HDL Example 7.15 MIPS INSTRUCTION MEMORY

Verilog

```
module imem(input  [5:0]  a,
            output [31:0] rd);

  reg [31:0] RAM[63:0];

  initial
    begin
      $readmemh("memfile.dat",RAM);
    end

  assign rd = RAM[a]; // word aligned
endmodule
```

VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.all; use STD.TEXTIO.all;
use IEEE.STD_LOGIC_UNSIGNED.all; use IEEE.STD_LOGIC_ARITH.all;

entity imem is -- instruction memory
  port(a:   in  STD_LOGIC_VECTOR(5 downto 0);
       rd:  out STD_LOGIC_VECTOR(31 downto 0));
end;

architecture behave of imem is
begin
  process is
    file mem_file: TEXT;
    variable L: line;
    variable ch: character;
    variable index, result: integer;
    type ramtype is array(63 downto 0) of STD_LOGIC_VECTOR
                    (31 downto 0);
    variable mem: ramtype;
  begin
    -- initialize memory from file
    for i in 0 to 63 loop -- set all contents low
      mem(conv_integer(i)) := CONV_STD_LOGIC_VECTOR(0, 32);
    end loop;
    index := 0;
    FILE_OPEN(mem_file, "C:/mips/memfile.dat", READ_MODE);
    while not endfile(mem_file) loop
      readline(mem_file, L);
      result := 0;
      for i in 1 to 8 loop
        read(L, ch);
        if '0' <= ch and ch <= '9' then
            result := result*16 + character'pos(ch) -
                      character'pos('0');
        elsif 'a' <= ch and ch <= 'f' then
          result := result*16 + character'pos(ch) -
                    character'pos('a') + 10;
        else report "Format error on line" & integer'image
             (index) severity error;
        end if;
      end loop;
      mem(index) := CONV_STD_LOGIC_VECTOR(result, 32);
      index := index + 1;
    end loop;

    -- read memory
    loop
      rd <= mem(CONV_INTEGER(a));
      wait on a;
    end loop;
  end process;
end;
```

7.7 EXCEPTIONS*

Section 6.7.2 introduced exceptions, which cause unplanned changes in the flow of a program. In this section, we enhance the multicycle processor to support two types of exceptions: undefined instructions

and arithmetic overflow. Supporting exceptions in other microarchitectures follows similar principles.

As described in Section 6.7.2, when an exception takes place, the processor copies the PC to the EPC register and stores a code in the Cause register indicating the source of the exception. Exception causes include 0x28 for undefined instructions and 0x30 for overflow (see Table 6.7). The processor then jumps to the exception handler at memory address 0x80000180. The exception handler is code that responds to the exception. It is part of the operating system.

Also as discussed in Section 6.7.2, the exception registers are part of *Coprocessor 0*, a portion of the MIPS processor that is used for system functions. Coprocessor 0 defines up to 32 special-purpose registers, including Cause and EPC. The exception handler may use the `mfc0` (move from coprocessor 0) instruction to copy these special-purpose registers into a general-purpose register in the register file; the Cause register is Coprocessor 0 register 13, and EPC is register 14.

To handle exceptions, we must add EPC and Cause registers to the datapath and extend the *PCSrc* multiplexer to accept the exception handler address, as shown in Figure 7.62. The two new registers have write enables, *EPCWrite* and *CauseWrite*, to store the PC and exception cause when an exception takes place. The cause is generated by a multiplexer

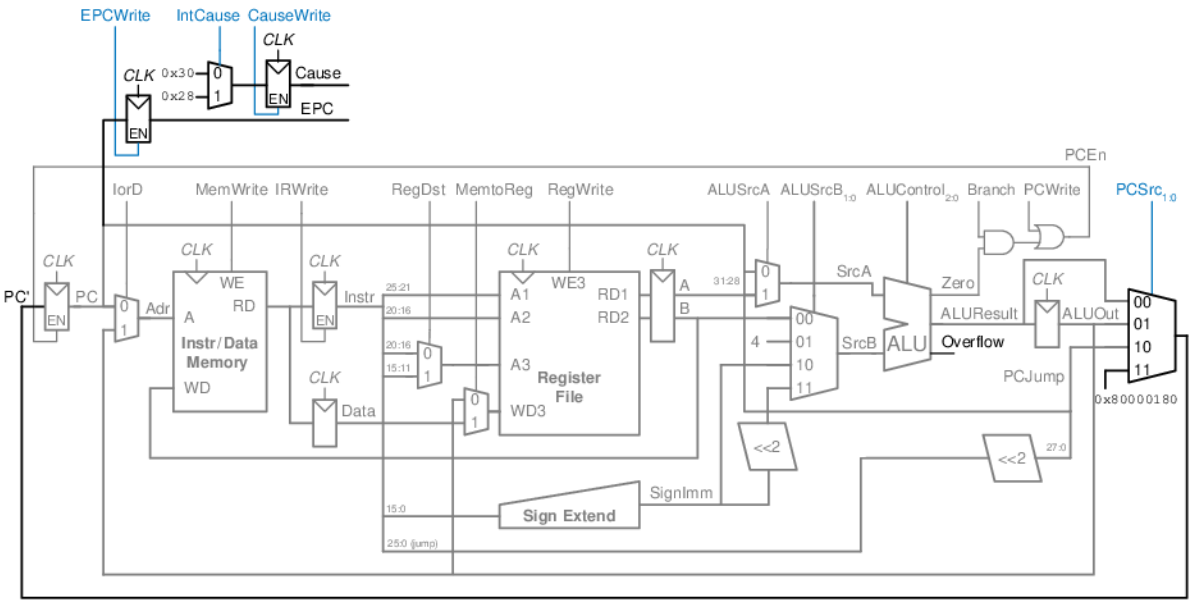


Figure 7.62 Datapath supporting overflow and undefined instruction exceptions

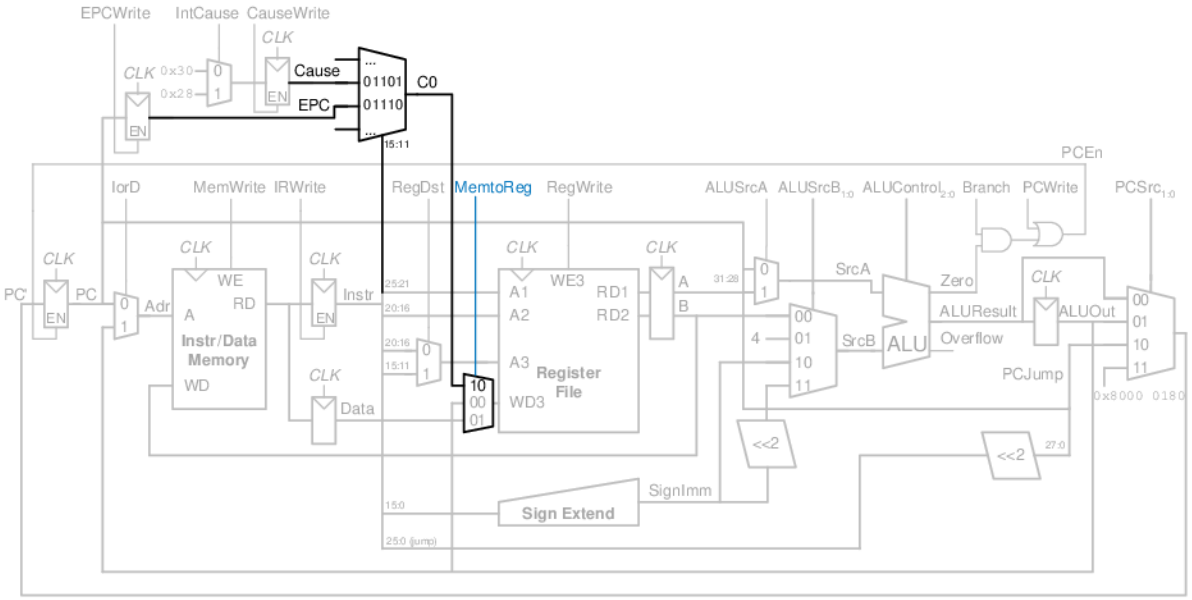


Figure 7.63 Datapath supporting `mfc0`

that selects the appropriate code for the exception. The ALU must also generate an overflow signal, as was discussed in Section 5.2.4.[5]

To support the `mfc0` instruction, we also add a way to select the Coprocessor 0 registers and write them to the register file, as shown in Figure 7.63. The `mfc0` instruction specifies the Coprocessor 0 register by $Instr_{15:11}$; in this diagram, only the Cause and EPC registers are supported. We add another input to the *MemtoReg* multiplexer to select the value from Coprocessor 0.

The modified controller is shown in Figure 7.64. The controller receives the overflow flag from the ALU. It generates three new control signals: one to write the EPC, a second to write the Cause register, and a third to select the Cause. It also includes two new states to support the two exceptions and another state to handle `mfc0`.

If the controller receives an undefined instruction (one that it does not know how to handle), it proceeds to S12, saves the PC in EPC, writes 0x28 to the Cause register, and jumps to the exception handler. Similarly, if the controller detects arithmetic overflow on an `add` or `sub` instruction, it proceeds to S13, saves the PC in EPC, writes 0x30

[5] Strictly speaking, the ALU should assert overflow only for `add` and `sub`, not for other ALU instructions.

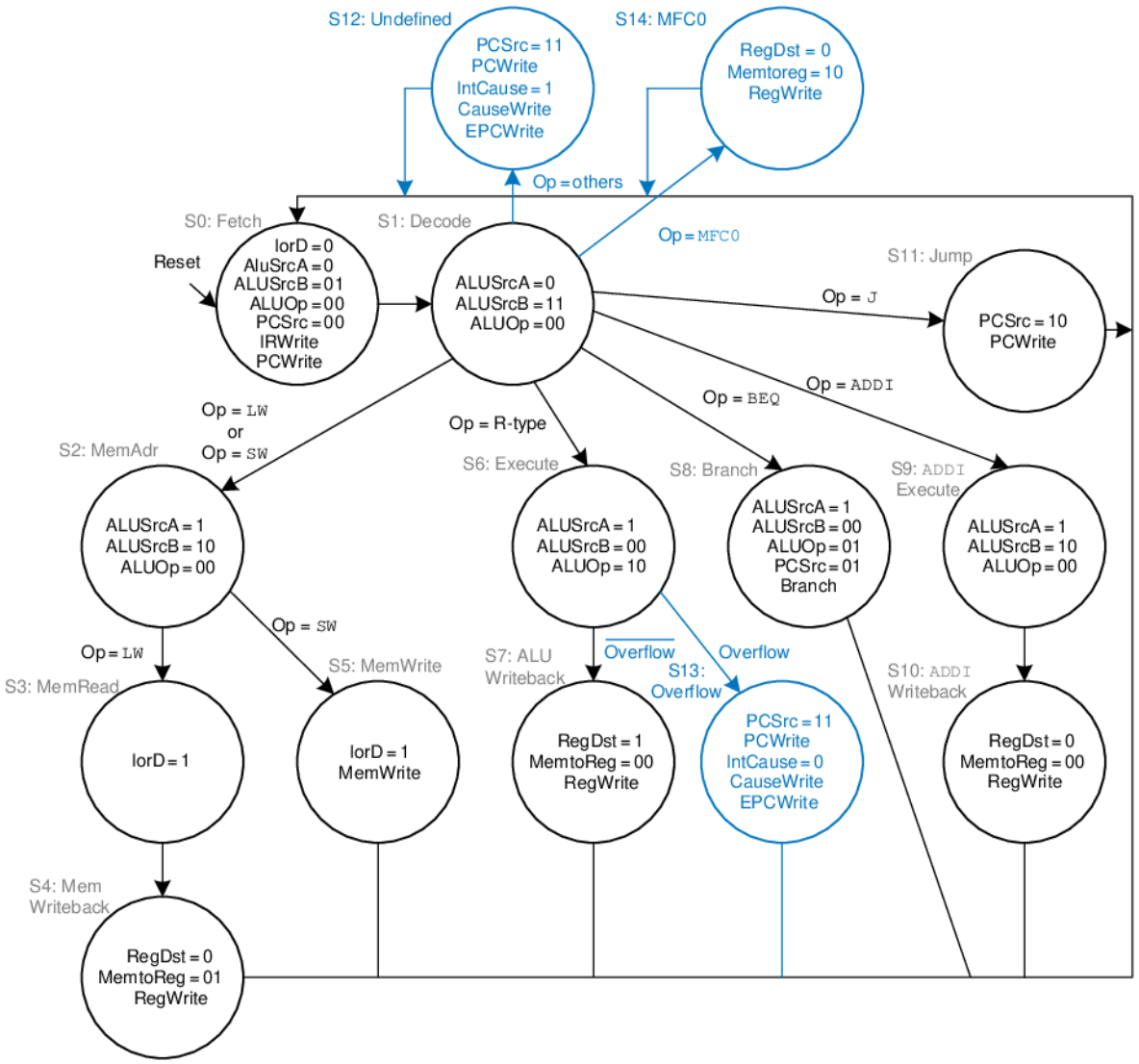


Figure 7.64 Controller supporting exceptions and `mfc0`

in the Cause register, and jumps to the exception handler. Note that, when an exception occurs, the instruction is discarded and the register file is not written. When a `mfc0` instruction is decoded, the processor goes to S14 and writes the appropriate Coprocessor 0 register to the main register file.

7.8 ADVANCED MICROARCHITECTURE*

High-performance microprocessors use a wide variety of techniques to run programs faster. Recall that the time required to run a program is proportional to the period of the clock and to the number of clock cycles per instruction (CPI). Thus, to increase performance we would like to speed up the clock and/or reduce the CPI. This section surveys some existing speedup techniques. The implementation details become quite complex, so we will focus on the concepts. Hennessy & Patterson's *Computer Architecture* text is a definitive reference if you want to fully understand the details.

Every 2 to 3 years, advances in CMOS manufacturing reduce transistor dimensions by 30% in each direction, doubling the number of transistors that can fit on a chip. A manufacturing process is characterized by its *feature size*, which indicates the smallest transistor that can be reliably built. Smaller transistors are faster and generally consume less power. Thus, even if the microarchitecture does not change, the clock frequency can increase because all the gates are faster. Moreover, smaller transistors enable placing more transistors on a chip. Microarchitects use the additional transistors to build more complicated processors or to put more processors on a chip. Unfortunately, power consumption increases with the number of transistors and the speed at which they operate (see Section 1.8). Power consumption is now an essential concern. Microprocessor designers have a challenging task juggling the trade-offs among speed, power, and cost for chips with billions of transistors in some of the most complex systems that humans have ever built.

7.8.1 Deep Pipelines

Aside from advances in manufacturing, the easiest way to speed up the clock is to chop the pipeline into more stages. Each stage contains less logic, so it can run faster. This chapter has considered a classic five-stage pipeline, but 10 to 20 stages are now commonly used.

The maximum number of pipeline stages is limited by pipeline hazards, sequencing overhead, and cost. Longer pipelines introduce more dependencies. Some of the dependencies can be solved by forwarding, but others require stalls, which increase the CPI. The pipeline registers between each stage have sequencing overhead from their setup time and clk-to-Q delay (as well as clock skew). This sequencing overhead makes adding more pipeline stages give diminishing returns. Finally, adding more stages increases the cost because of the extra pipeline registers and hardware required to handle hazards.

Example 7.11 DEEP PIPELINES

Consider building a pipelined processor by chopping up the single-cycle processor into N stages ($N \geq 5$). The single-cycle processor has a propagation delay of 900 ps through the combinational logic. The sequencing overhead of a register is 50 ps. Assume that the combinational delay can be arbitrarily divided into any number of stages and that pipeline hazard logic does not increase the delay. The five-stage pipeline in Example 7.9 has a CPI of 1.15. Assume that each additional stage increases the CPI by 0.1 because of branch mispredictions and other pipeline hazards. How many pipeline stages should be used to make the processor execute programs as fast as possible?

Solution: If the 900-ps combinational logic delay is divided into N stages and each stage also pays 50 ps of sequencing overhead for its pipeline register, the cycle time is $T_c = 900/N + 50$. The CPI is $1.15 + 0.1(N - 5)$. The time per instruction, or instruction time, is the product of the cycle time and the CPI. Figure 7.65 plots the cycle time and instruction time versus the number of stages. The instruction time has a minimum of 231 ps at $N = 11$ stages. This minimum is only slightly better than the 250 ps per instruction achieved with a six-stage pipeline.

In the late 1990s and early 2000s, microprocessors were marketed largely based on clock frequency ($1/T_c$). This pushed microprocessors to use very deep pipelines (20 to 31 stages on the Pentium 4) to maximize the clock frequency, even if the benefits for overall performance were questionable. Power is proportional to clock frequency and also increases with the number of pipeline registers, so now that power consumption is so important, pipeline depths are decreasing.

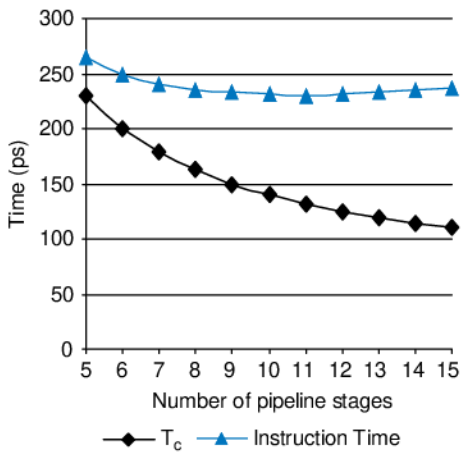


Figure 7.65 Cycle time and instruction time versus the number of pipeline stages

7.8.2 Branch Prediction

An ideal pipelined processor would have a CPI of 1. The branch misprediction penalty is a major reason for increased CPI. As pipelines get deeper, branches are resolved later in the pipeline. Thus, the branch misprediction penalty gets larger, because all the instructions issued after the mispredicted branch must be flushed. To address this problem, most pipelined processors use a *branch predictor* to guess whether the branch should be taken. Recall that our pipeline from Section 7.5.3 simply predicted that branches are never taken.

Some branches occur when a program reaches the end of a loop (e.g., a `for` or `while` statement) and branches back to repeat the loop. Loops tend to be executed many times, so these backward branches are usually taken. The simplest form of branch prediction checks the direction of the branch and predicts that backward branches should be taken. This is called *static branch prediction*, because it does not depend on the history of the program.

Forward branches are difficult to predict without knowing more about the specific program. Therefore, most processors use *dynamic branch predictors*, which use the history of program execution to guess whether a branch should be taken. Dynamic branch predictors maintain a table of the last several hundred (or thousand) branch instructions that the processor has executed. The table, sometimes called a *branch target buffer*, includes the destination of the branch and a history of whether the branch was taken.

To see the operation of dynamic branch predictors, consider the following loop code from Code Example 6.20. The loop repeats 10 times, and the `beq` out of the loop is taken only on the last time.

```
add  $s1, $0, $0       # sum = 0
add  $s0, $0, $0       # i   = 0
addi $t0, $0, 10       # $t0 = 10

for:
  beq  $s0, $t0, done  # if i == 10, branch to done
  add  $s1, $s1, $s0   # sum = sum + i
  addi $s0, $s0, 1     # increment i
  j    for
done:
```

A *one-bit dynamic branch predictor* remembers whether the branch was taken the last time and predicts that it will do the same thing the next time. While the loop is repeating, it remembers that the `beq` was not taken last time and predicts that it should not be taken next time. This is a correct prediction until the last branch of the loop, when the branch does get taken. Unfortunately, if the loop is run again, the branch predictor remembers that the last branch was taken. Therefore,

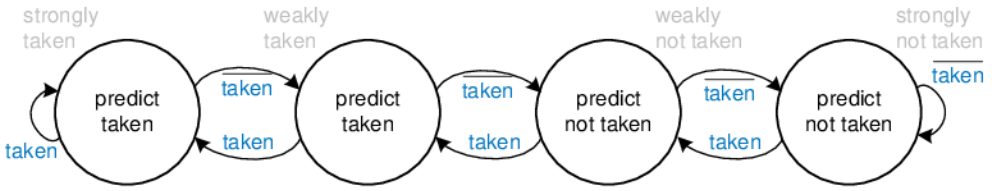


Figure 7.66 2-bit branch predictor state transition diagram

it incorrectly predicts that the branch should be taken when the loop is first run again. In summary, a 1-bit branch predictor mispredicts the first and last branches of a loop.

A 2-bit dynamic branch predictor solves this problem by having four states: *strongly taken*, *weakly taken*, *weakly not taken*, and *strongly not taken*, as shown in Figure 7.66. When the loop is repeating, it enters the "strongly not taken" state and predicts that the branch should not be taken next time. This is correct until the last branch of the loop, which is taken and moves the predictor to the "weakly not taken" state. When the loop is first run again, the branch predictor correctly predicts that the branch should not be taken and reenters the "strongly not taken" state. In summary, a 2-bit branch predictor mispredicts only the last branch of a loop.

As one can imagine, branch predictors may be used to track even more history of the program to increase the accuracy of predictions. Good branch predictors achieve better than 90% accuracy on typical programs.

The branch predictor operates in the Fetch stage of the pipeline so that it can determine which instruction to execute on the next cycle. When it predicts that the branch should be taken, the processor fetches the next instruction from the branch destination stored in the branch target buffer. By keeping track of both branch and jump destinations in the branch target buffer, the processor can also avoid flushing the pipeline during jump instructions.

7.8.3 Superscalar Processor

A *superscalar processor* contains multiple copies of the datapath hardware to execute multiple instructions simultaneously. Figure 7.67 shows a block diagram of a two-way superscalar processor that fetches and executes two instructions per cycle. The datapath fetches two instructions at a time from the instruction memory. It has a six-ported register file to read four source operands and write two results back in each cycle. It also contains two ALUs and a two-ported data memory to execute the two instructions at the same time.

A *scalar* processor acts on one piece of data at a time. A *vector* processor acts on several pieces of data with a single instruction. A *superscalar* processor issues several instructions at a time, each of which operates on one piece of data.

Our MIPS pipelined processor is a scalar processor. Vector processors were popular for supercomputers in the 1980s and 1990s because they efficiently handled the long vectors of data common in scientific computations. Modern high-performance microprocessors are superscalar, because issuing several independent instructions is more flexible than processing vectors.

However, modern processors also include hardware to handle short vectors of data that are common in multimedia and graphics applications. These are called *single instruction multiple data* (*SIMD*) units.

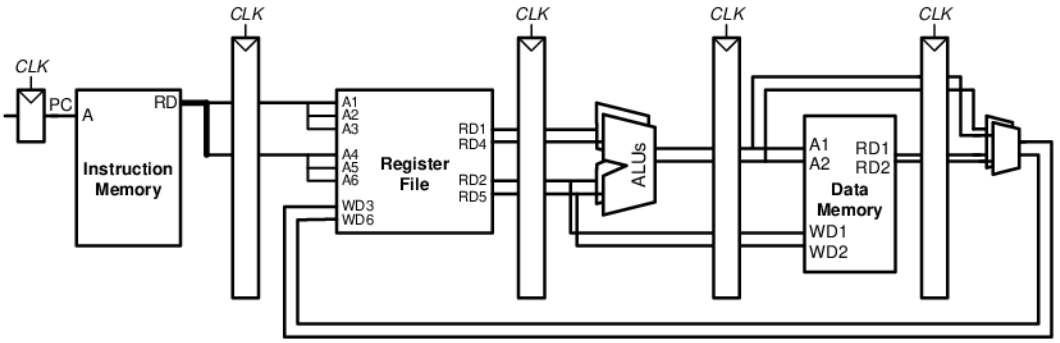


Figure 7.67 Superscalar datapath

Figure 7.68 shows a pipeline diagram illustrating the two-way superscalar processor executing two instructions on each cycle. For this program, the processor has a CPI of 0.5. Designers commonly refer to the reciprocal of the CPI as the *instructions per cycle*, or *IPC*. This processor has an IPC of 2 on this program.

Executing many instructions simultaneously is difficult because of dependencies. For example, Figure 7.69 shows a pipeline diagram running a program with data dependencies. The dependencies in the code are shown in blue. The `add` instruction is dependent on `$t0`, which is produced by the `lw` instruction, so it cannot be issued at the same time as `lw`. Indeed, the `add` instruction stalls for yet another cycle so that `lw` can forward `$t0` to `add` in cycle 5. The other dependencies (between

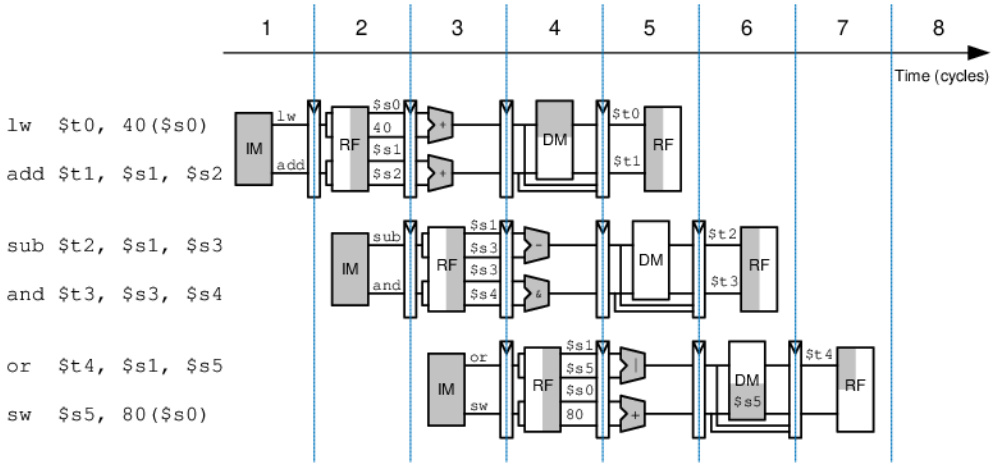


Figure 7.68 Abstract view of a superscalar pipeline in operation

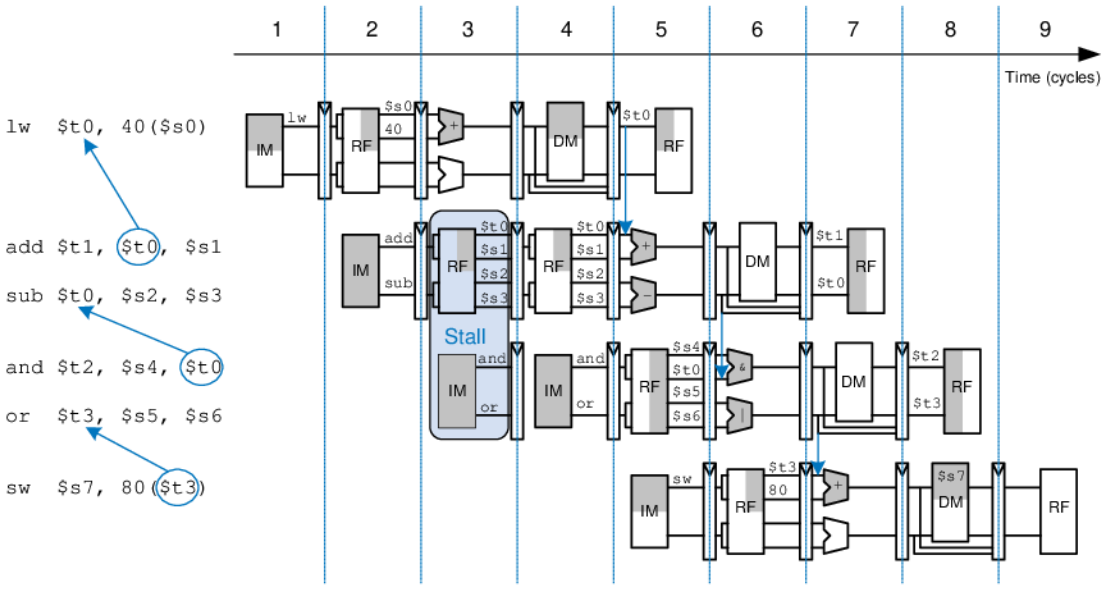


Figure 7.69 Program with data dependencies

`sub` and `and` based on `$t0`, and between `or` and `sw` based on `$t3`) are handled by forwarding results produced in one cycle to be consumed in the next. This program, also given below, requires five cycles to issue six instructions, for an IPC of 1.17.

```
lw  $t0, 40($s0)
add $t1, $t0, $s1
sub $t0, $s2, $s3
and $t2, $s4, $t0
or  $t3, $s5, $s6
sw  $s7, 80($t3)
```

Recall that parallelism comes in temporal and spatial forms. Pipelining is a case of temporal parallelism. Multiple execution units is a case of spatial parallelism. Superscalar processors exploit both forms of parallelism to squeeze out performance far exceeding that of our single-cycle and multicycle processors.

Commercial processors may be three-, four-, or even six-way superscalar. They must handle control hazards such as branches as well as data hazards. Unfortunately, real programs have many dependencies, so wide superscalar processors rarely fully utilize all of the execution units. Moreover, the large number of execution units and complex forwarding networks consume vast amounts of circuitry and power.

7.8.4 Out-of-Order Processor

To cope with the problem of dependencies, an *out-of-order processor* looks ahead across many instructions to *issue*, or begin executing, independent instructions as rapidly as possible. The instructions can be issued in a different order than that written by the programmer, as long as dependencies are honored so that the program produces the intended result.

Consider running the same program from Figure 7.69 on a two-way superscalar out-of-order processor. The processor can issue up to two instructions per cycle from anywhere in the program, as long as dependencies are observed. Figure 7.70 shows the data dependencies and the operation of the processor. The classifications of dependencies as RAW and WAR will be discussed shortly. The constraints on issuing instructions are described below.

- **Cycle 1**
  - The `lw` instruction issues.
  - The `add`, `sub`, and `and` instructions are dependent on `lw` by way of `$t0`, so they cannot issue yet. However, the `or` instruction is independent, so it also issues.

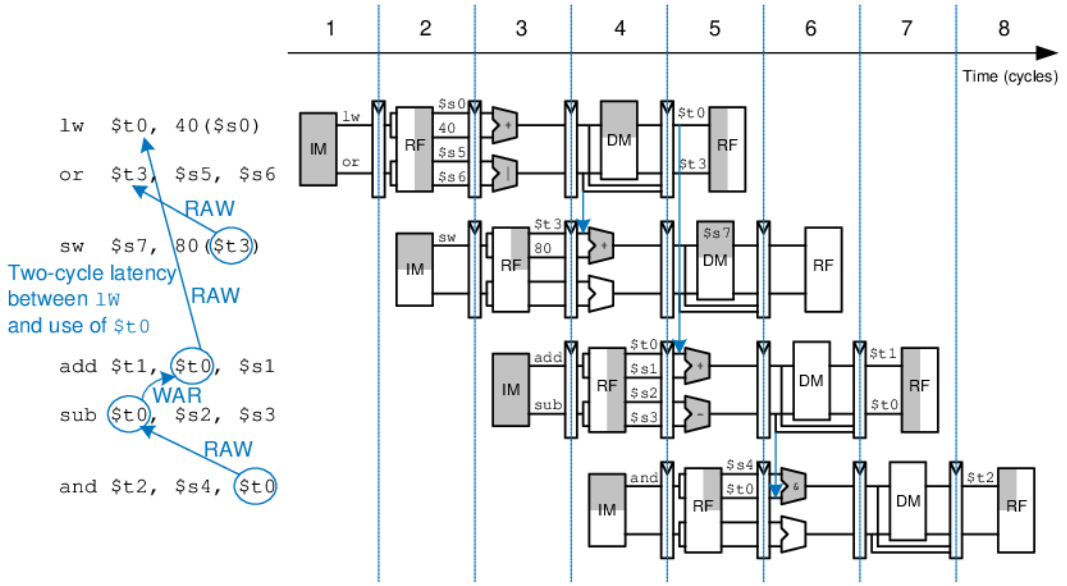


Figure 7.70 Out-of-order execution of a program with dependencies

- Cycle 2
  - Remember that there is a two-cycle latency between when a `lw` instruction issues and when a dependent instruction can use its result, so `add` cannot issue yet because of the `$t0` dependence. `sub` writes `$t0`, so it cannot issue before `add`, lest `add` receive the wrong value of `$t0`. `and` is dependent on `sub`.
  - Only the `sw` instruction issues.
- Cycle 3
  - On cycle 3, `$t0` is available, so `add` issues. `sub` issues simultaneously, because it will not write `$t0` until after `add` consumes `$t0`.
- Cycle 4
  - The `and` instruction issues. `$t0` is forwarded from `sub` to `and`.

The out-of-order processor issues the six instructions in four cycles, for an IPC of 1.5.

The dependence of `add` on `lw` by way of `$t0` is a read after write (RAW) hazard. `add` must not read `$t0` until after `lw` has written it. This is the type of dependency we are accustomed to handling in the pipelined processor. It inherently limits the speed at which the program can run, even if infinitely many execution units are available. Similarly, the dependence of `sw` on `or` by way of `$t3` and of `and` on `sub` by way of `$t0` are RAW dependencies.

The dependence between `sub` and `add` by way of `$t0` is called a *write after read* (*WAR*) hazard or an *antidependence*. `sub` must not write `$t0` before `add` reads `$t0`, so that `add` receives the correct value according to the original order of the program. WAR hazards could not occur in the simple MIPS pipeline, but they may happen in an out-of-order processor if the dependent instruction (in this case, `sub`) is moved too early.

A WAR hazard is not essential to the operation of the program. It is merely an artifact of the programmer's choice to use the same register for two unrelated instructions. If the `sub` instruction had written `$t4` instead of `$t0`, the dependency would disappear and `sub` could be issued before `add`. The MIPS architecture only has 32 registers, so sometimes the programmer is forced to reuse a register and introduce a hazard just because all the other registers are in use.

A third type of hazard, not shown in the program, is called *write after write* (*WAW*) or an *output dependence*. A WAW hazard occurs if an instruction attempts to write a register after a subsequent instruction has already written it. The hazard would result in the wrong value being

written to the register. For example, in the following program, `add` and `sub` both write `$t0`. The final value in `$t0` should come from `sub` according to the order of the program. If an out-of-order processor attempted to execute `sub` first, the WAW hazard would occur.

```
add $t0, $s1, $s2
sub $t0, $s3, $s4
```

WAW hazards are not essential either; again, they are artifacts caused by the programmer's using the same register for two unrelated instructions. If the `sub` instruction were issued first, the program could eliminate the WAW hazard by discarding the result of the `add` instead of writing it to `$t0`. This is called *squashing* the `add`.[6]

Out-of-order processors use a table to keep track of instructions waiting to issue. The table, sometimes called a *scoreboard*, contains information about the dependencies. The size of the table determines how many instructions can be considered for issue. On each cycle, the processor examines the table and issues as many instructions as it can, limited by the dependencies and by the number of execution units (e.g., ALUs, memory ports) that are available.

The *instruction level parallelism* (*ILP*) is the number of instructions that can be executed simultaneously for a particular program and microarchitecture. Theoretical studies have shown that the ILP can be quite large for out-of-order microarchitectures with perfect branch predictors and enormous numbers of execution units. However, practical processors seldom achieve an ILP greater than 2 or 3, even with six-way superscalar datapaths with out-of-order execution.

7.8.5 Register Renaming

Out-of-order processors use a technique called *register renaming* to eliminate WAR hazards. Register renaming adds some nonarchitectural *renaming registers* to the processor. For example, a MIPS processor might add 20 renaming registers, called `$r0-$r19`. The programmer cannot use these registers directly, because they are not part of the architecture. However, the processor is free to use them to eliminate hazards.

For example, in the previous section, a WAR hazard occurred between the `sub` and `add` instructions based on reusing `$t0`. The out-of-order processor could *rename* `$t0` to `$r0` for the `sub` instruction. Then

[6] You might wonder why the `add` needs to be issued at all. The reason is that out-of-order processors must guarantee that all of the same exceptions occur that would have occurred if the program had been executed in its original order. The `add` potentially may produce an overflow exception, so it must be issued to check for the exception, even though the result can be discarded.

`sub` could be executed sooner, because `$r0` has no dependency on the `add` instruction. The processor keeps a table of which registers were renamed so that it can consistently rename registers in subsequent dependent instructions. In this example, `$t0` must also be renamed to `$r0` in the `and` instruction, because it refers to the result of `sub`.

Figure 7.71 shows the same program from Figure 7.70 executing on an out-of-order processor with register renaming. `$t0` is renamed to `$r0` in `sub` and `and` to eliminate the WAR hazard. The constraints on issuing instructions are described below.

- **Cycle 1**
  - The `lw` instruction issues.
  - The `add` instruction is dependent on `lw` by way of `$t0`, so it cannot issue yet. However, the `sub` instruction is independent now that its destination has been renamed to `$r0`, so `sub` also issues.
- **Cycle 2**
  - Remember that there is a two-cycle latency between when a `lw` issues and when a dependent instruction can use its result, so `add` cannot issue yet because of the `$t0` dependence.
  - The `and` instruction is dependent on `sub`, so it can issue. `$r0` is forwarded from `sub` to `and`.
  - The `or` instruction is independent, so it also issues.

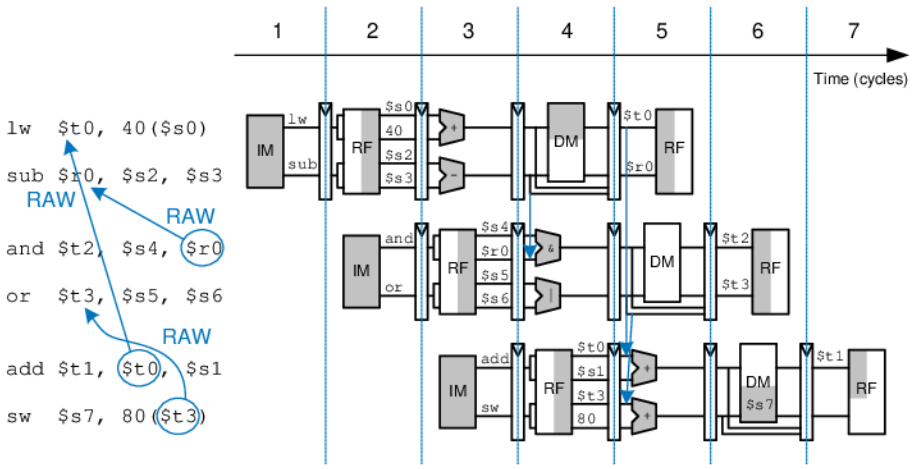


Figure 7.71 Out-of-order execution of a program using register renaming

- Cycle 3
  - On cycle 3, `$t0` is available, so `add` issues. `$t3` is also available, so `sw` issues.

The out-of-order processor with register renaming issues the six instructions in three cycles, for an IPC of 2.

7.8.6 Single Instruction Multiple Data

The term *SIMD* (pronounced "sim-dee") stands for *single instruction multiple data*, in which a single instruction acts on multiple pieces of data in parallel. A common application of SIMD is to perform many short arithmetic operations at once, especially for graphics processing. This is also called *packed* arithmetic.

For example, a 32-bit microprocessor might pack four 8-bit data elements into one 32-bit word. Packed add and subtract instructions operate on all four data elements within the word in parallel. Figure 7.72 shows a packed 8-bit addition summing four pairs of 8-bit numbers to produce four results. The word could also be divided into two 16-bit elements. Performing packed arithmetic requires modifying the ALU to eliminate carries between the smaller data elements. For example, a carry out of $a_0 + b_0$ should not affect the result of $a_1 + b_1$.

Short data elements often appear in graphics processing. For example, a pixel in a digital photo may use 8 bits to store each of the red, green, and blue color components. Using an entire 32-bit word to process one of these components wastes the upper 24 bits. When the components from four adjacent pixels are packed into a 32-bit word, the processing can be performed four times faster.

SIMD instructions are even more helpful for 64-bit architectures, which can pack eight 8-bit elements, four 16-bit elements, or two 32-bit elements into a single 64-bit word. SIMD instructions are also used for floating-point computations; for example, four 32-bit single-precision floating-point values can be packed into a single 128-bit word.

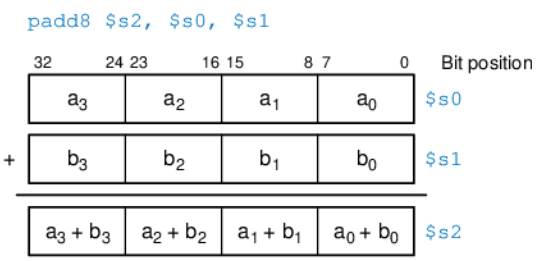


Figure 7.72 Packed arithmetic: four simultaneous 8-bit additions

7.8.7 Multithreading

Because the ILP of real programs tends to be fairly low, adding more execution units to a superscalar or out-of-order processor gives diminishing returns. Another problem, discussed in Chapter 8, is that memory is much slower than the processor. Most loads and stores access a smaller and faster memory, called a *cache*. However, when the instructions or data are not available in the cache, the processor may stall for 100 or more cycles while retrieving the information from the main memory. Multithreading is a technique that helps keep a processor with many execution units busy even if the ILP of a program is low or the program is stalled waiting for memory.

To explain multithreading, we need to define a few new terms. A program running on a computer is called a *process*. Computers can run multiple processes simultaneously; for example, you can play music on a PC while surfing the web and running a virus checker. Each process consists of one or more *threads* that also run simultaneously. For example, a word processor may have one thread handling the user typing, a second thread spell-checking the document while the user works, and a third thread printing the document. In this way, the user does not have to wait, for example, for a document to finish printing before being able to type again.

In a conventional processor, the threads only give the illusion of running simultaneously. The threads actually take turns being executed on the processor under control of the OS. When one thread's turn ends, the OS saves its architectural state, loads the architectural state of the next thread, and starts executing that next thread. This procedure is called *context switching*. As long as the processor switches through all the threads fast enough, the user perceives all of the threads as running at the same time.

A multithreaded processor contains more than one copy of its architectural state, so that more than one thread can be active at a time. For example, if we extended a MIPS processor to have four program counters and 128 registers, four threads could be available at one time. If one thread stalls while waiting for data from main memory, the processor could context switch to another thread without any delay, because the program counter and registers are already available. Moreover, if one thread lacks sufficient parallelism to keep all the execution units busy, another thread could issue instructions to the idle units.

Multithreading does not improve the performance of an individual thread, because it does not increase the ILP. However, it does improve the overall throughput of the processor, because multiple threads can use processor resources that would have been idle when executing a single thread. Multithreading is also relatively inexpensive to implement, because it replicates only the PC and register file, not the execution units and memories.

7.8.8 Multiprocessors

A *multiprocessor* system consists of multiple processors and a method for communication between the processors. A common form of multiprocessing in computer systems is *symmetric multiprocessing* (*SMP*), in which two or more identical processors share a single main memory.

The multiple processors may be separate chips or multiple *cores* on the same chip. Modern processors have enormous numbers of transistors available. Using them to increase the pipeline depth or to add more execution units to a superscalar processor gives little performance benefit and is wasteful of power. Around the year 2005, computer architects made a major shift to build multiple copies of the processor on the same chip; these copies are called cores.

Multiprocessors can be used to run more threads simultaneously or to run a particular thread faster. Running more threads simultaneously is easy; the threads are simply divided up among the processors. Unfortunately typical PC users need to run only a small number of threads at any given time. Running a particular thread faster is much more challenging. The programmer must divide the thread into pieces to perform on each processor. This becomes tricky when the processors need to communicate with each other. One of the major challenges for computer designers and programmers is to effectively use large numbers of processor cores.

Other forms of multiprocessing include asymmetric multiprocessing and clusters. *Asymmetric multiprocessors* use separate specialized microprocessors for separate tasks. For example, a cell phone contains a *digital signal processor* (*DSP*) with specialized instructions to decipher the wireless data in real time and a separate conventional processor to interact with the user, manage the phone book, and play games. In *clustered multiprocessing*, each processor has its own local memory system. Clustering can also refer to a group of PCs connected together on the network running software to jointly solve a large problem.

Scientists searching for signs of extraterrestrial intelligence use the world's largest clustered multiprocessors to analyze radio telescope data for patterns that might be signs of life in other solar systems. The cluster consists of personal computers owned by more than 3.8 million volunteers around the world.

When a computer in the cluster is idle, it fetches a piece of the data from a centralized server, analyzes the data, and sends the results back to the server. You can volunteer your computer's idle time for the cluster by visiting setiathome.berkeley.edu.

7.9 REAL-WORLD PERSPECTIVE: IA-32 MICROARCHITECTURE*

Section 6.8 introduced the IA-32 architecture used in almost all PCs. This section tracks the evolution of IA-32 processors through progressively faster and more complicated microarchitectures. The same principles we have applied to the MIPS microarchitectures are used in IA-32.

Intel invented the first single-chip microprocessor, the 4-bit 4004, in 1971 as a flexible controller for a line of calculators. It contained 2300 transistors manufactured on a 12-mm^2 sliver of silicon in a process with a 10-μm feature size and operated at 750 KHz. A photograph of the chip taken under a microscope is shown in Figure 7.73.

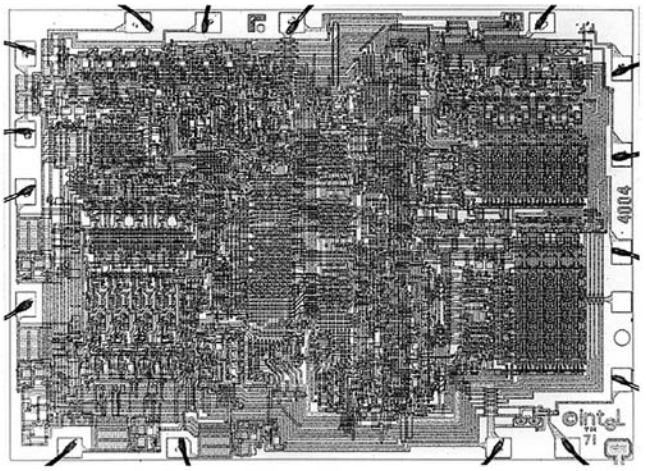


Figure 7.73 4004 microprocessor chip

In places, columns of four similar-looking structures are visible, as one would expect in a 4-bit microprocessor. Around the periphery are *bond wires*, which are used to connect the chip to its package and the circuit board.

The 4004 inspired the 8-bit 8008, then the 8080, which eventually evolved into the 16-bit 8086 in 1978 and the 80286 in 1982. In 1985, Intel introduced the 80386, which extended the 8086 architecture to 32 bits and defined the IA-32 architecture. Table 7.7 summarizes major Intel IA-32 microprocessors. In the 35 years since the 4004, transistor feature size has shrunk 160-fold, the number of transistors

Table 7.7 Evolution of Intel IA-32 microprocessors

Processor	Year	Feature Size (μm)	Transistors	Frequency (MHz)	Microarchitecture
80386	1985	1.5–1.0	275k	16–25	multicycle
80486	1989	1.0–0.6	1.2M	25–100	pipelined
Pentium	1993	0.8–0.35	3.2–4.5M	60–300	superscalar
Pentium II	1997	0.35–0.25	7.5M	233–450	out of order
Pentium III	1999	0.25–0.18	9.5M–28M	450–1400	out of order
Pentium 4	2001	0.18–0.09	42–178M	1400–3730	out of order
Pentium M	2003	0.13–0.09	77–140M	900–2130	out of order
Core Duo	2005	0.065	152M	1500–2160	dual core

on a chip has increased by five orders of magnitude, and the operating frequency has increased by almost four orders of magnitude. No other field of engineering has made such astonishing progress in such a short time.

The 80386 is a multicycle processor. The major components are labeled on the chip photograph in Figure 7.74. The 32-bit datapath is clearly visible on the left. Each of the columns processes one bit of data. Some of the control signals are generated using a *microcode* PLA that steps through the various states of the control FSM. The memory management unit in the upper right controls access to the external memory.

The 80486, shown in Figure 7.75, dramatically improved performance using pipelining. The datapath is again clearly visible, along with the control logic and microcode PLA. The 80486 added an on-chip floating-point unit; previous Intel processors either sent floating-point instructions to a separate coprocessor or emulated them in software. The 80486 was too fast for external memory to keep up, so it incorporated an 8-KB cache onto the chip to hold the most commonly used instructions and data. Chapter 8 describes caches in more detail and revisits the cache systems on Intel IA-32 processors.

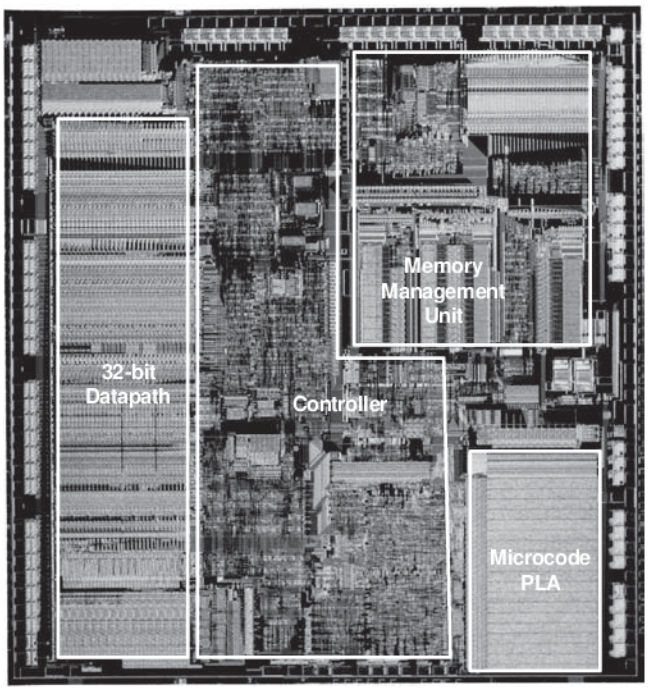


Figure 7.74 80386 microprocessor chip

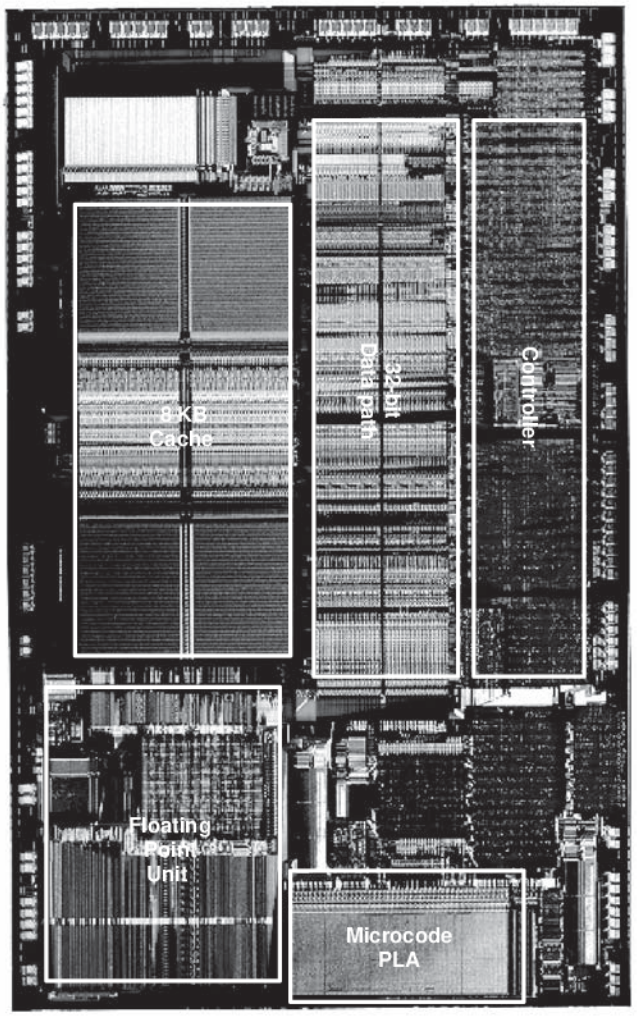


Figure 7.75 80486 microprocessor chip

The Pentium processor, shown in Figure 7.76, is a superscalar processor capable of executing two instructions simultaneously. Intel switched to the name Pentium instead of 80586 because AMD was becoming a serious competitor selling interchangeable 80486 chips, and part numbers cannot be trademarked. The Pentium uses separate instruction and data caches. It also uses a branch predictor to reduce the performance penalty for branches.

The Pentium Pro, Pentium II, and Pentium III processors all share a common out-of-order microarchitecture, code named P6. The complex IA-32 instructions are broken down into one or more micro-ops similar

to MIPS instructions. The micro-ops are then executed on a fast out-of-order execution core with an 11-stage pipeline. Figure 7.77 shows the Pentium III. The 32-bit datapath is called the Integer Execution Unit (IEU). The floating-point datapath is called the Floating Point Unit

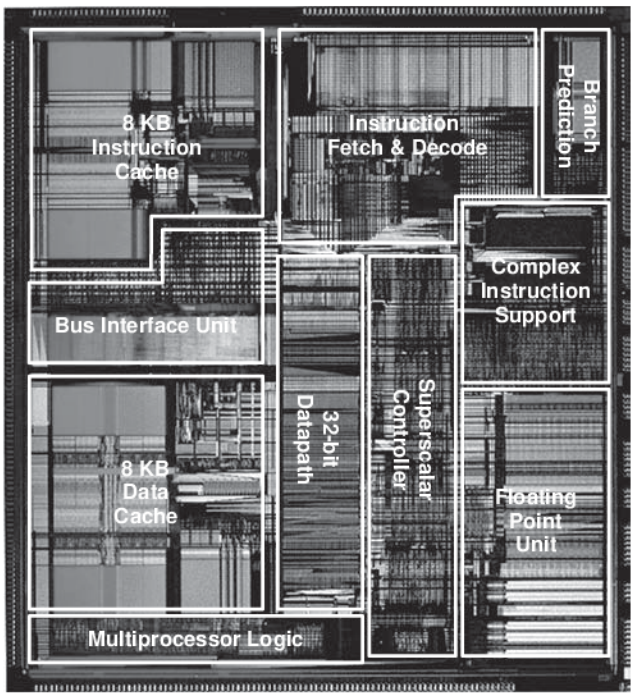


Figure 7.76 Pentium microprocessor chip

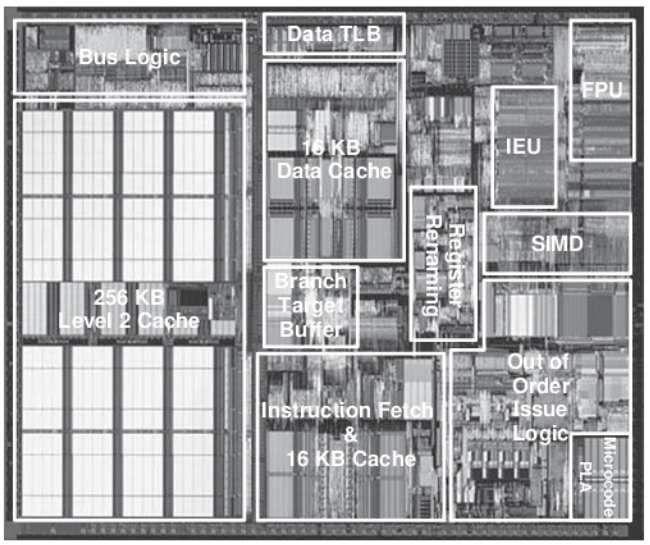


Figure 7.77 Pentium III microprocessor chip

(FPU). The processor also has a SIMD unit to perform packed operations on short integer and floating-point data. A larger portion of the chip is dedicated to issuing instructions out-of-order than to actually executing the instructions. The instruction and data caches have grown to 16 KB each. The Pentium III also has a larger but slower 256-KB second-level cache on the same chip.

By the late 1990s, processors were marketed largely on clock speed. The Pentium 4 is another out-of-order processor with a very deep pipeline to achieve extremely high clock frequencies. It started with 20 stages, and later versions adopted 31 stages to achieve frequencies greater than 3 GHz. The chip, shown in Figure 7.78, packs in 42 to 178 million transistors (depending on the cache size), so even the major execution units are difficult to see on the photograph. Decoding three IA-32 instructions per cycle is impossible at such high clock frequencies because the instruction encodings are so complex and irregular. Instead, the processor predecodes the instructions into simpler micro-ops, then stores the micro-ops in a memory called a *trace cache*. Later versions of the Pentium 4 also perform multithreading to increase the throughput of multiple threads.

The Pentium 4's reliance on deep pipelines and high clock speed led to extremely high power consumption, sometimes more than 100 W. This is unacceptable in laptops and makes cooling of desktops expensive.

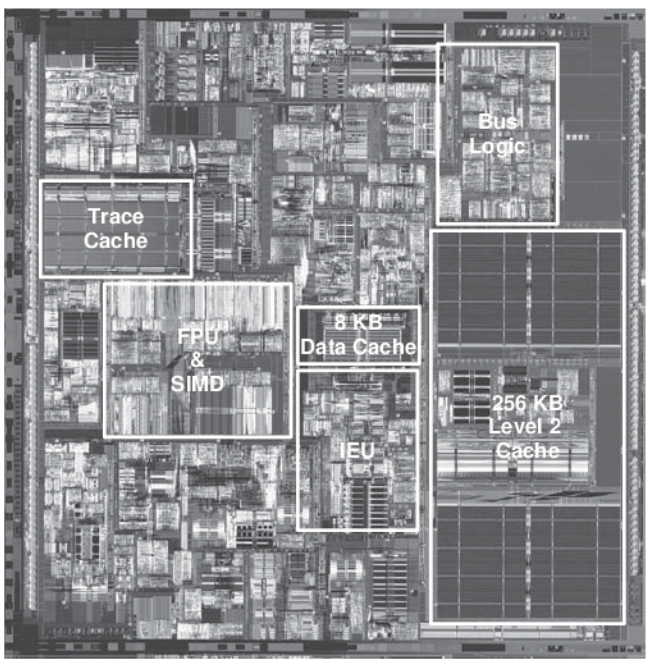


Figure 7.78 Pentium 4 microprocessor chip

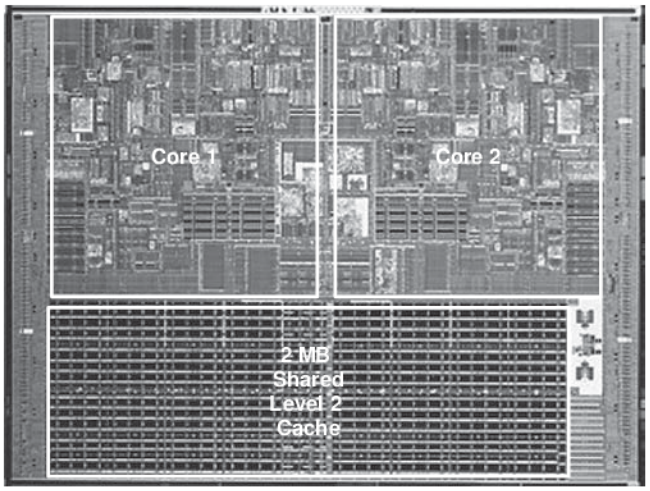


Figure 7.79 Core Duo microprocessor chip

Intel discovered that the older P6 architecture could achieve comparable performance at much lower clock speed and power. The Pentium M uses an enhanced version of the P6 out-of-order microarchitecture with 32-KB instruction and data caches and a 1- to 2-MB second-level cache. The Core Duo is a multicore processor based on two Pentium M cores connected to a shared 2-MB second-level cache. The individual functional units in Figure 7.79 are difficult to see, but the two cores and the large cache are clearly visible.

7.10 SUMMARY

This chapter has described three ways to build MIPS processors, each with different performance and cost trade-offs. We find this topic almost magical: how can such a seemingly complicated device as a microprocessor actually be simple enough to fit in a half-page schematic? Moreover, the inner workings, so mysterious to the uninitiated, are actually reasonably straightforward.

The MIPS microarchitectures have drawn together almost every topic covered in the text so far. Piecing together the microarchitecture puzzle illustrates the principles introduced in previous chapters, including the design of combinational and sequential circuits, covered in Chapters 2 and 3; the application of many of the building blocks described in Chapter 5; and the implementation of the MIPS architecture, introduced in Chapter 6. The MIPS microarchitectures can be described in a few pages of HDL, using the techniques from Chapter 4.

Building the microarchitectures has also heavily used our techniques for managing complexity. The microarchitectural abstraction forms the link between the logic and architecture abstractions, forming

the crux of this book on digital design and computer architecture. We also use the abstractions of block diagrams and HDL to succinctly describe the arrangement of components. The microarchitectures exploit regularity and modularity, reusing a library of common building blocks such as ALUs, memories, multiplexers, and registers. Hierarchy is used in numerous ways. The microarchitectures are partitioned into the datapath and control units. Each of these units is built from logic blocks, which can be built from gates, which in turn can be built from transistors using the techniques developed in the first five chapters.

This chapter has compared single-cycle, multicycle, and pipelined microarchitectures for the MIPS processor. All three microarchitectures implement the same subset of the MIPS instruction set and have the same architectural state. The single-cycle processor is the most straightforward and has a CPI of 1.

The multicycle processor uses a variable number of shorter steps to execute instructions. It thus can reuse the ALU, rather than requiring several adders. However, it does require several nonarchitectural registers to store results between steps. The multicycle design in principle could be faster, because not all instructions must be equally long. In practice, it is generally slower, because it is limited by the slowest steps and by the sequencing overhead in each step.

The pipelined processor divides the single-cycle processor into five relatively fast pipeline stages. It adds pipeline registers between the stages to separate the five instructions that are simultaneously executing. It nominally has a CPI of 1, but hazards force stalls or flushes that increase the CPI slightly. Hazard resolution also costs some extra hardware and design complexity. The clock period ideally could be five times shorter than that of the single-cycle processor. In practice, it is not that short, because it is limited by the slowest stage and by the sequencing overhead in each stage. Nevertheless, pipelining provides substantial performance benefits. All modern high-performance microprocessors use pipelining today.

Although the microarchitectures in this chapter implement only a subset of the MIPS architecture, we have seen that supporting more instructions involves straightforward enhancements of the datapath and controller. Supporting exceptions also requires simple modifications.

A major limitation of this chapter is that we have assumed an ideal memory system that is fast and large enough to store the entire program and data. In reality, large fast memories are prohibitively expensive. The next chapter shows how to get most of the benefits of a large fast memory with a small fast memory that holds the most commonly used information and one or more larger but slower memories that hold the rest of the information.

Exercises

Exercise 7.1 Suppose that one of the following control signals in the single-cycle MIPS processor has a *stuck-at-0 fault*, meaning that the signal is always 0, regardless of its intended value. What instructions would malfunction? Why?

(a) *RegWrite*

(b) $ALUOp_1$

(c) *MemWrite*

Exercise 7.2 Repeat Exercise 7.1, assuming that the signal has a stuck-at-1 fault.

Exercise 7.3 Modify the single-cycle MIPS processor to implement one of the following instructions. See Appendix B for a definition of the instructions. Mark up a copy of Figure 7.11 to indicate the changes to the datapath. Name any new control signals. Mark up a copy of Table 7.8 to show the changes to the main decoder. Describe any other changes that are required.

(a) `sll`

(b) `lui`

(c) `slti`

(d) `blez`

(e) `jal`

(f) `lh`

Table 7.8 Main decoder truth table to mark up with changes

Instruction	Opcode	RegWrite	RegDst	ALUSrc	Branch	MemWrite	MemtoReg	ALUOp
R-type	000000	1	1	0	0	0	0	10
`lw`	100011	1	0	1	0	0	1	00
`sw`	101011	0	X	1	0	1	X	00
`beq`	000100	0	X	0	1	0	X	01

Exercise 7.4 Many processor architectures have a *load with postincrement* instruction, which updates the index register to point to the next memory word after completing the load. `lwinc $rt, imm($rs)` is equivalent to the following two instructions:

```
lw   $rt, imm($rs)
addi $rs, $rs, 4
```

Repeat Exercise 7.3 for the `lwinc` instruction. Is it possible to add the instruction without modifying the register file?

Exercise 7.5 Add a single-precision floating-point unit to the single-cycle MIPS processor to handle `add.s`, `sub.s`, and `mul.s`. Assume that you have single-precision floating-point adder and multiplier units available. Explain what changes must be made to the datapath and the controller.

Exercise 7.6 Your friend is a crack circuit designer. She has offered to redesign one of the units in the single-cycle MIPS processor to have half the delay. Using the delays from Table 7.6, which unit should she work on to obtain the greatest speedup of the overall processor, and what would the cycle time of the improved machine be?

Exercise 7.7 Consider the delays given in Table 7.6. Ben Bitdiddle builds a prefix adder that reduces the ALU delay by 20 ps. If the other element delays stay the same, find the new cycle time of the single-cycle MIPS processor and determine how long it takes to execute a benchmark with 100 billion instructions.

Exercise 7.8 Suppose one of the following control signals in the multicycle MIPS processor has a stuck-at-0 fault, meaning that the signal is always 0, regardless of its intended value. What instructions would malfunction? Why?

(a) *MemtoReg*
(b) $ALUOp_0$
(c) *PCSrc*

Exercise 7.9 Repeat Exercise 7.8, assuming that the signal has a stuck-at-1 fault.

Exercise 7.10 Modify the HDL code for the single-cycle MIPS processor, given in Section 7.6.1, to handle one of the new instructions from Exercise 7.3. Enhance the testbench, given in Section 7.6.3 to test the new instruction.

Exercise 7.11 Modify the multicycle MIPS processor to implement one of the following instructions. See Appendix B for a definition of the instructions. Mark up a copy of Figure 7.27 to indicate the changes to the datapath. Name any new control signals. Mark up a copy of Figure 7.39 to show the changes to the controller FSM. Describe any other changes that are required.

(a) `srlv`
(b) `ori`
(c) `xori`
(d) `jr`
(e) `bne`
(f) `lbu`

Exercise 7.12 Repeat Exercise 7.4 for the multicycle MIPS processor. Show the changes to the multicycle datapath and control FSM. Is it possible to add the instruction without modifying the register file?

Exercise 7.13 Repeat Exercise 7.5 for the multicycle MIPS processor.

Exercise 7.14 Suppose that the floating-point adder and multiplier from Exercise 7.13 each take two cycles to operate. In other words, the inputs are applied at the beginning of one cycle, and the output is available in the second cycle. How does your answer to Exercise 7.13 change?

Exercise 7.15 Your friend, the crack circuit designer, has offered to redesign one of the units in the multicycle MIPS processor to be much faster. Using the delays from Table 7.6, which unit should she work on to obtain the greatest speedup of the overall processor? How fast should it be? (Making it faster than necessary is a waste of your friend's effort.) What is the cycle time of the improved processor?

Exercise 7.16 Repeat Exercise 7.7 for the multicycle processor.

Exercise 7.17 Suppose the multicycle MIPS processor has the component delays given in Table 7.6. Alyssa P. Hacker designs a new register file that has 40% less power but twice as much delay. Should she switch to the slower but lower power register file for her multicycle processor design?

Exercise 7.18 Goliath Corp claims to have a patent on a three-ported register file. Rather than fighting Goliath in court, Ben Bitdiddle designs a new register file that has only a single read/write port (like the combined instruction and data memory). Redesign the MIPS multicycle datapath and controller to use his new register file.

Exercise 7.19 What is the CPI of the redesigned multicycle MIPS processor from Exercise 7.18? Use the instruction mix from Example 7.7.

Exercise 7.20 How many cycles are required to run the following program on the multicycle MIPS processor? What is the CPI of this program?

```
    addi $s0, $0, 5   # sum = 5

while:
    beq  $s0, $0, done# if result > 0, execute the while block
    addi $s0, $s0, -1 # while block: result = result - 1
    j    while

done:
```

Exercise 7.21 Repeat Exercise 7.20 for the following program.

```
      add  $s0, $0, $0   # i = 0
      add  $s1, $0, $0   # sum = 0
      addi $t0, $0, 10   # $t0 = 10

    loop:
      slt  $t1, $s0, $t0 # if (i < 10), $t1 = 1, else $t1 = 0
      beq  $t1, $0, done # if $t1 == 0 (i >= 10), branch to done
      add  $s1, $s1, $s0 # sum = sum + i
      addi $s0, $s0, 1   # increment i
      j loop
    done:
```

Exercise 7.22 Write HDL code for the multicycle MIPS processor. The processor should be compatible with the following top-level module. The mem module is used to hold both instructions and data. Test your processor using the testbench from Section 7.6.3.

```
module top(input         clk, reset,
           output [31:0] writedata, adr,
           output        memwrite);

  wire [31:0] readdata;

  // instantiate processor and memories
  mips mips(clk, reset, adr, writedata, memwrite, readdata);
  mem mem(clk, memwrite, adr, writedata, readdata);

  endmodule

module mem(input         clk, we,
           input  [31:0] a, wd,
           output [31:0] rd);

  reg [31:0] RAM[63:0];

  initial
   begin
    $readmemh("memfile.dat",RAM);
   end

assign rd = RAM[a[31:2]]; // word aligned
  always @ (posedge clk)
    if (we)
     RAM[a[31:2]] <= wd;
endmodule
```

Exercise 7.23 Extend your HDL code for the multicycle MIPS processor from Exercise 7.22 to handle one of the new instructions from Exercise 7.11. Enhance the testbench to test the new instruction.

Exercise 7.24 The pipelined MIPS processor is running the following program. Which registers are being written, and which are being read on the fifth cycle?

```
add $s0, $t0, $t1
sub $s1, $t2, $t3
and $s2, $s0, $s1
or  $s3, $t4, $t5
slt $s4, $s2, $s3
```

Exercise 7.25 Using a diagram similar to Figure 7.52, show the forwarding and stalls needed to execute the following instructions on the pipelined MIPS processor.

```
add $t0, $s0, $s1
sub $t0, $t0, $s2
lw  $t1, 60($t0)
and $t2, $t1, $t0
```

Exercise 7.26 Repeat Exercise 7.25 for the following instructions.

```
add $t0, $s0, $s1
lw  $t1, 60($s2)
sub $t2, $t0, $s3
and $t3, $t1, $t0
```

Exercise 7.27 How many cycles are required for the pipelined MIPS processor to issue all of the instructions for the program in Exercise 7.21? What is the CPI of the processor on this program?

Exercise 7.28 Explain how to extend the pipelined MIPS processor to handle the `addi` instruction.

Exercise 7.29 Explain how to extend the pipelined processor to handle the `j` instruction. Give particular attention to how the pipeline is flushed when a jump takes place.

Exercise 7.30 Examples 7.9 and 7.10 point out that the pipelined MIPS processor performance might be better if branches take place during the Execute stage rather than the Decode stage. Show how to modify the pipelined processor from Figure 7.58 to branch in the Execute stage. How do the stall and flush signals change? Redo Examples 7.9 and 7.10 to find the new CPI, cycle time, and overall time to execute the program.

Exercise 7.31 Your friend, the crack circuit designer, has offered to redesign one of the units in the pipelined MIPS processor to be much faster. Using the delays from Table 7.6 and Example 7.10, which unit should she work on to obtain the greatest speedup of the overall processor? How fast should it be? (Making it faster than necessary is a waste of your friend's effort.) What is the cycle time of the improved processor?

Exercise 7.32 Consider the delays from Table 7.6 and Example 7.10. Now suppose that the ALU were 20% faster. Would the cycle time of the pipelined MIPS processor change? What if the ALU were 20% slower?

Exercise 7.33 Write HDL code for the pipelined MIPS processor. The processor should be compatible with the top-level module from HDL Example 7.13. It should support all of the instructions described in this chapter, including `addi` and `j` (see Exercises 7.28 and 7.29). Test your design using the testbench from HDL Example 7.12.

Exercise 7.34 Design the hazard unit shown in Figure 7.58 for the pipelined MIPS processor. Use an HDL to implement your design. Sketch the hardware that a synthesis tool might generate from your HDL.

Exercise 7.35 A *nonmaskable interrupt* (*NMI*) is triggered by an input pin to the processor. When the pin is asserted, the current instruction should finish, then the processor should set the Cause register to 0 and take an exception. Show how to modify the multicycle processor in Figures 7.63 and 7.64 to handle nonmaskable interrupts.

Interview Questions

The following exercises present questions that have been asked at interviews for digital design jobs.

Question 7.1 Explain the advantages of pipelined microprocessors.

Question 7.2 If additional pipeline stages allow a processor to go faster, why don't processors have 100 pipeline stages?

Question 7.3 Describe what a hazard is in a microprocessor and explain ways in which it can be resolved. What are the pros and cons of each way?

Question 7.4 Describe the concept of a superscalar processor and its pros and cons.

Memory Systems

8

8.1 INTRODUCTION

Computer system performance depends on the memory system as well as the processor microarchitecture. Chapter 7 assumed an ideal memory system that could be accessed in a single clock cycle. However, this would be true only for a very small memory—or a very slow processor! Early processors were relatively slow, so memory was able to keep up. But processor speed has increased at a faster rate than memory speeds. DRAM memories are currently 10 to 100 times slower than processors. The increasing gap between processor and DRAM memory speeds demands increasingly ingenious memory systems to try to approximate a memory that is as fast as the processor. This chapter investigates practical memory systems and considers trade-offs of speed, capacity, and cost.

The processor communicates with the memory system over a *memory interface*. Figure 8.1 shows the simple memory interface used in our multicycle MIPS processor. The processor sends an address over the *Address* bus to the memory system. For a read, *MemWrite* is 0 and the memory returns the data on the *ReadData* bus. For a write, *MemWrite* is 1 and the processor sends data to memory on the *WriteData* bus.

The major issues in memory system design can be broadly explained using a metaphor of books in a library. A library contains many books on the shelves. If you were writing a term paper on the meaning of dreams, you might go to the library[1] and pull Freud's *The Interpretation of Dreams* off the shelf and bring it to your cubicle. After skimming it, you might put it back and pull out Jung's *The Psychology of the*

[1] We realize that library usage is plummeting among college students because of the Internet. But we also believe that libraries contain vast troves of hard-won human knowledge that are not electronically available. We hope that Web searching does not completely displace the art of library research.

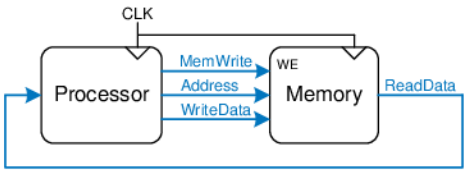
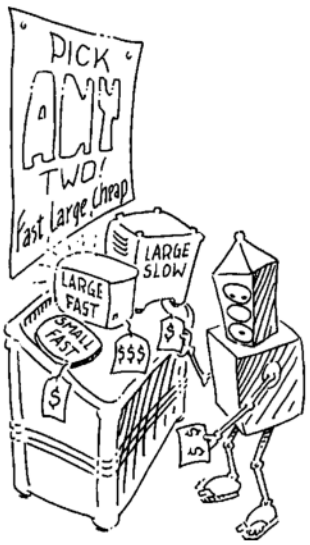


Figure 8.1 The memory interface

Unconscious. You might then go back for another quote from *Interpretation of Dreams*, followed by yet another trip to the stacks for Freud's *The Ego and the Id*. Pretty soon you would get tired of walking from your cubicle to the stacks. If you are clever, you would save time by keeping the books in your cubicle rather than schlepping them back and forth. Furthermore, when you pull a book by Freud, you could also pull several of his other books from the same shelf.

This metaphor emphasizes the principle, introduced in Section 6.2.1, of making the common case fast. By keeping books that you have recently used or might likely use in the future at your cubicle, you reduce the number of time-consuming trips to the stacks. In particular, you use the principles of *temporal* and *spatial locality*. Temporal locality means that if you have used a book recently, you are likely to use it again soon. Spatial locality means that when you use one particular book, you are likely to be interested in other books on the same shelf.

The library itself makes the common case fast by using these principles of locality. The library has neither the shelf space nor the budget to accommodate all of the books in the world. Instead, it keeps some of the lesser-used books in deep storage in the basement. Also, it may have an interlibrary loan agreement with nearby libraries so that it can offer more books than it physically carries.



In summary, you obtain the benefits of both a large collection and quick access to the most commonly used books through a hierarchy of storage. The most commonly used books are in your cubicle. A larger collection is on the shelves. And an even larger collection is available, with advanced notice, from the basement and other libraries. Similarly, memory systems use a hierarchy of storage to quickly access the most commonly used data while still having the capacity to store large amounts of data.

Memory subsystems used to build this hierarchy were introduced in Section 5.5. Computer memories are primarily built from dynamic RAM (DRAM) and static RAM (SRAM). Ideally, the computer memory system is fast, large, and cheap. In practice, a single memory only has two of these three attributes; it is either slow, small, or expensive. But computer systems can approximate the ideal by combining a fast small cheap memory and a slow large cheap memory. The fast memory stores the most commonly used data and instructions, so on average the memory

system appears fast. The large memory stores the remainder of the data and instructions, so the overall capacity is large. The combination of two cheap memories is much less expensive than a single large fast memory. These principles extend to using an entire hierarchy of memories of increasing capacity and decreasing speed.

Computer memory is generally built from DRAM chips. In 2006, a typical PC had a *main memory* consisting of 256 MB to 1 GB of DRAM, and DRAM cost about $100 per gigabyte (GB). DRAM prices have declined at about 30% per year for the last three decades, and memory capacity has grown at the same rate, so the total cost of the memory in a PC has remained roughly constant. Unfortunately, DRAM speed has improved by only about 7% per year, whereas processor performance has improved at a rate of 30 to 50% per year, as shown in Figure 8.2. The plot shows memory and processor speeds with the 1980 speeds as a baseline. In about 1980, processor and memory speeds were the same. But performance has diverged since then, with memories badly lagging.

DRAM could keep up with processors in the 1970s and early 1980's, but it is now woefully too slow. The DRAM access time is one to two orders of magnitude longer than the processor cycle time (tens of nanoseconds, compared to less than one nanosecond).

To counteract this trend, computers store the most commonly used instructions and data in a faster but smaller memory, called a *cache*. The cache is usually built out of SRAM on the same chip as the processor. The cache speed is comparable to the processor speed, because SRAM is inherently faster than DRAM, and because the on-chip memory eliminates lengthy delays caused by traveling to and from a separate chip. In 2006, on-chip SRAM costs were on the order of $10,000/GB, but the cache is relatively small (kilobytes to a few megabytes), so the overall

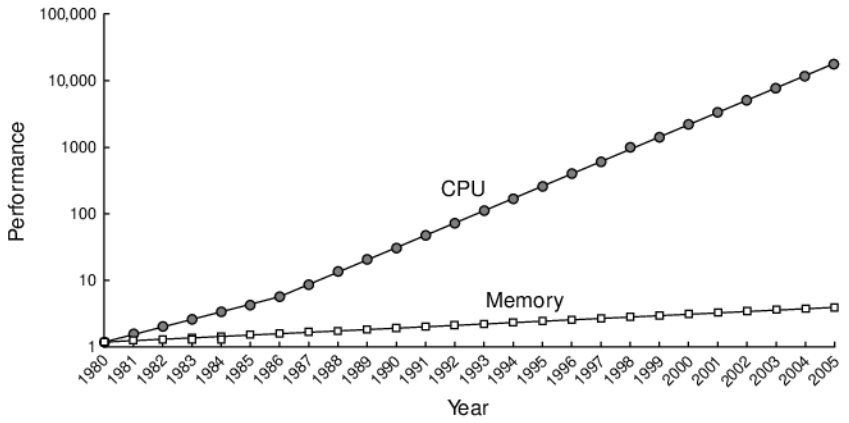


Figure 8.2 Diverging processor and memory performance Adapted with permission from Hennessy and Patterson, *Computer Architecture: A Quantitative Approach*, 3rd ed., Morgan Kaufmann, 2003.

cost is low. Caches can store both instructions and data, but we will refer to their contents generically as "data."

If the processor requests data that is available in the cache, it is returned quickly. This is called a cache *hit*. Otherwise, the processor retrieves the data from main memory (DRAM). This is called a cache *miss*. If the cache hits most of the time, then the processor seldom has to wait for the slow main memory, and the average access time is low.

The third level in the memory hierarchy is the *hard disk*, or *hard drive*. In the same way that a library uses the basement to store books that do not fit in the stacks, computer systems use the hard disk to store data that does not fit in main memory. In 2006, a hard disk cost less than \$1/GB and had an access time of about 10 ms. Hard disk costs have decreased at 60%/year but access times scarcely improved. The hard disk provides an illusion of more capacity than actually exists in the main memory. It is thus called *virtual memory*. Like books in the basement, data in virtual memory takes a long time to access. Main memory, also called *physical memory*, holds a subset of the virtual memory. Hence, the main memory can be viewed as a cache for the most commonly used data from the hard disk.

Figure 8.3 summarizes the memory hierarchy of the computer system discussed in the rest of this chapter. The processor first seeks data in a small but fast cache that is usually located on the same chip. If the data is not available in the cache, the processor then looks in main memory. If the data is not there either, the processor fetches the data from virtual memory on the large but slow hard disk. Figure 8.4 illustrates this capacity and speed trade-off in the memory hierarchy and lists typical costs and access times in 2006 technology. As access time decreases, speed increases.

Section 8.2 introduces memory system performance analysis. Section 8.3 explores several cache organizations, and Section 8.4 delves into virtual memory systems. To conclude, this chapter explores how processors can access input and output devices, such as keyboards and monitors, in much the same way as they access memory. Section 8.5 investigates such memory-mapped I/O.

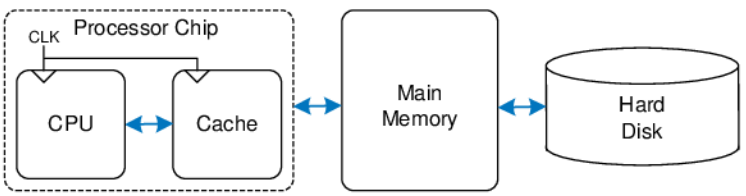


Figure 8.3 **A typical memory hierarchy**

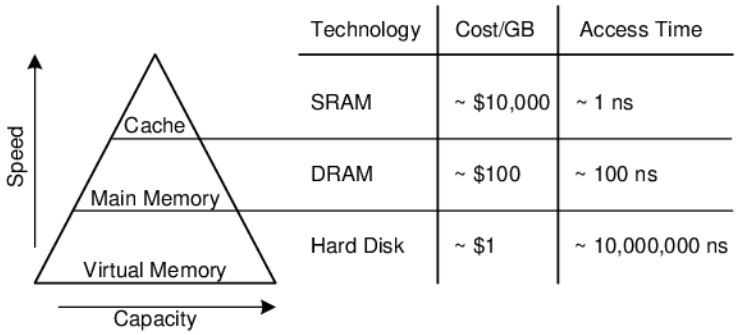


Figure 8.4 Memory hierarchy components, with typical characteristics in 2006

8.2 MEMORY SYSTEM PERFORMANCE ANALYSIS

Designers (and computer buyers) need quantitative ways to measure the performance of memory systems to evaluate the cost-benefit trade-offs of various alternatives. Memory system performance metrics are *miss rate* or *hit rate* and *average memory access time*. Miss and hit rates are calculated as:

$$Miss\ Rate = \frac{Number\ of\ misses}{Number\ of\ total\ memory\ accesses} = 1 - Hit\ Rate$$

$$Hit\ Rate = \frac{Number\ of\ hits}{Number\ of\ total\ memory\ accesses} = 1 - Miss\ Rate \tag{8.1}$$

Example 8.1 CALCULATING CACHE PERFORMANCE

Suppose a program has 2000 data access instructions (loads or stores), and 1250 of these requested data values are found in the cache. The other 750 data values are supplied to the processor by main memory or disk memory. What are the miss and hit rates for the cache?

Solution: The miss rate is 750/2000 = 0.375 = 37.5%. The hit rate is 1250/2000 = 0.625 = 1 − 0.375 = 62.5%.

Average memory access time (AMAT) is the average time a processor must wait for memory per load or store instruction. In the typical computer system from Figure 8.3, the processor first looks for the data in the cache. If the cache misses, the processor then looks in main memory. If the main memory misses, the processor accesses virtual memory on the hard disk. Thus, *AMAT* is calculated as:

$$AMAT = t_{cache} + MR_{cache}(t_{MM} + MR_{MM}t_{VM}) \tag{8.2}$$

where t_{cache}, t_{MM}, and t_{VM} are the access times of the cache, main memory, and virtual memory, and MR_{cache} and MR_{MM} are the cache and main memory miss rates, respectively.

Example 8.2 CALCULATING AVERAGE MEMORY ACCESS TIME

Suppose a computer system has a memory organization with only two levels of hierarchy, a cache and main memory. What is the average memory access time given the access times and miss rates given in Table 8.1?

Solution: The average memory access time is $1 + 0.1(100) = 11$ cycles.

Table 8.1 Access times and miss rates

Memory Level	Access Time (Cycles)	Miss Rate
Cache	1	10%
Main Memory	100	0%

Example 8.3 IMPROVING ACCESS TIME

An 11-cycle average memory access time means that the processor spends ten cycles waiting for data for every one cycle actually using that data. What cache miss rate is needed to reduce the average memory access time to 1.5 cycles given the access times in Table 8.1?

Solution: If the miss rate is m, the average access time is $1 + 100m$. Setting this time to 1.5 and solving for m requires a cache miss rate of 0.5%.

Gene Amdahl, 1922–. Most famous for Amdahl's Law, an observation he made in 1965. While in graduate school, he began designing computers in his free time. This side work earned him his Ph.D. in theoretical physics in 1952. He joined IBM immediately after graduation, and later went on to found three companies, including one called Amdahl Corporation in 1970.

As a word of caution, performance improvements might not always be as good as they sound. For example, making the memory system ten times faster will not necessarily make a computer program run ten times as fast. If 50% of a program's instructions are loads and stores, a ten-fold memory system improvement only means a 1.82-fold improvement in program performance. This general principle is called *Amdahl's Law*, which says that the effort spent on increasing the performance of a subsystem is worthwhile only if the subsystem affects a large percentage of the overall performance.

8.3 CACHES

A cache holds commonly used memory data. The number of data words that it can hold is called the *capacity*, C. Because the capacity

of the cache is smaller than that of main memory, the computer system designer must choose what subset of the main memory is kept in the cache.

When the processor attempts to access data, it first checks the cache for the data. If the cache hits, the data is available immediately. If the cache misses, the processor fetches the data from main memory and places it in the cache for future use. To accommodate the new data, the cache must *replace* old data. This section investigates these issues in cache design by answering the following questions: (1) What data is held in the cache? (2) How is the data found? and (3) What data is replaced to make room for new data when the cache is full?

When reading the next sections, keep in mind that the driving force in answering these questions is the inherent spatial and temporal locality of data accesses in most applications. Caches use spatial and temporal locality to predict what data will be needed next. If a program accesses data in a random order, it would not benefit from a cache.

As we explain in the following sections, caches are specified by their capacity (C), number of sets (S), block size (b), number of blocks (B), and degree of associativity (N).

Although we focus on data cache loads, the same principles apply for fetches from an instruction cache. Data cache store operations are similar and are discussed further in Section 8.3.4.

Cache: a hiding place especially for concealing and preserving provisions or implements.

– Merriam Webster Online Dictionary. 2006. http://www.merriam-webster.com

8.3.1 What Data Is Held in the Cache?

An ideal cache would anticipate all of the data needed by the processor and fetch it from main memory ahead of time so that the cache has a zero miss rate. Because it is impossible to predict the future with perfect accuracy, the cache must guess what data will be needed based on the past pattern of memory accesses. In particular, the cache exploits temporal and spatial locality to achieve a low miss rate.

Recall that temporal locality means that the processor is likely to access a piece of data again soon if it has accessed that data recently. Therefore, when the processor loads or stores data that is not in the cache, the data is copied from main memory into the cache. Subsequent requests for that data hit in the cache.

Recall that spatial locality means that, when the processor accesses a piece of data, it is also likely to access data in nearby memory locations. Therefore, when the cache fetches one word from memory, it may also fetch several adjacent words. This group of words is called a *cache block*. The number of words in the cache block, b, is called the *block size*. A cache of capacity C contains $B = C/b$ blocks.

The principles of temporal and spatial locality have been experimentally verified in real programs. If a variable is used in a program, the

same variable is likely to be used again, creating temporal locality. If an element in an array is used, other elements in the same array are also likely to be used, creating spatial locality.

8.3.2 How Is the Data Found?

A cache is organized into S *sets*, each of which holds one or more blocks of data. The relationship between the address of data in main memory and the location of that data in the cache is called the *mapping*. Each memory address maps to exactly one set in the cache. Some of the address bits are used to determine which cache set contains the data. If the set contains more than one block, the data may be kept in any of the blocks in the set.

Caches are categorized based on the number of blocks in a set. In a *direct mapped* cache, each set contains exactly one block, so the cache has $S = B$ sets. Thus, a particular main memory address maps to a unique block in the cache. In an *N-way set associative* cache, each set contains N blocks. The address still maps to a unique set, with $S = B/N$ sets. But the data from that address can go in any of the N blocks in that set. A *fully associative* cache has only $S = 1$ set. Data can go in any of the B blocks in the set. Hence, a fully associative cache is another name for a B-way set associative cache.

To illustrate these cache organizations, we will consider a MIPS memory system with 32-bit addresses and 32-bit words. The memory is byte-addressable, and each word is four bytes, so the memory consists of 2^{30} words aligned on word boundaries. We analyze caches with an eight-word capacity (C) for the sake of simplicity. We begin with a one-word block size (b), then generalize later to larger blocks.

Direct Mapped Cache

A *direct mapped* cache has one block in each set, so it is organized into $S = B$ sets. To understand the mapping of memory addresses onto cache blocks, imagine main memory as being mapped into b-word blocks, just as the cache is. An address in block 0 of main memory maps to set 0 of the cache. An address in block 1 of main memory maps to set 1 of the cache, and so forth until an address in block $B - 1$ of main memory maps to block $B - 1$ of the cache. There are no more blocks of the cache, so the mapping wraps around, such that block B of main memory maps to block 0 of the cache.

This mapping is illustrated in Figure 8.5 for a direct mapped cache with a capacity of eight words and a block size of one word. The cache has eight sets, each of which contains a one-word block. The bottom two bits of the address are always 00, because they are word aligned. The next $\log_2 8 = 3$ bits indicate the set onto which the memory address maps. Thus, the data at addresses 0x00000004, 0x00000024, . . . ,

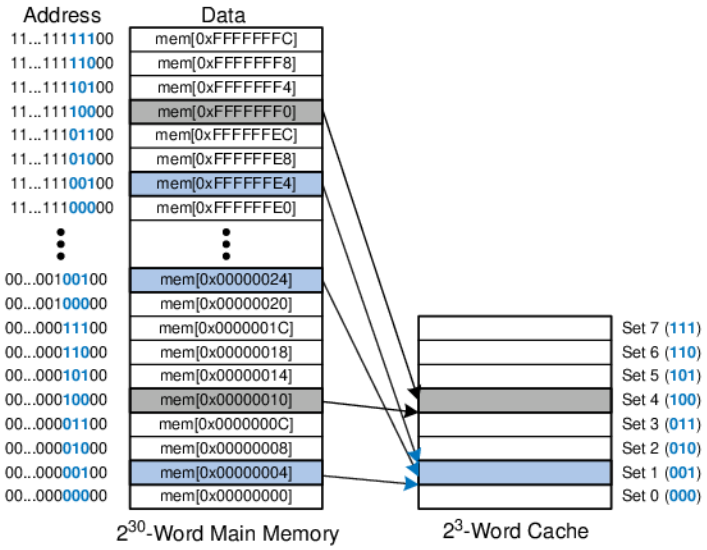


Figure 8.5 Mapping of main memory to a direct mapped cache

0xFFFFFFE4 all map to set 1, as shown in blue. Likewise, data at addresses 0x00000010, . . . , 0xFFFFFFF0 all map to set 4, and so forth. Each main memory address maps to exactly one set in the cache.

Example 8.4 CACHE FIELDS

To what cache set in Figure 8.5 does the word at address 0x00000014 map? Name another address that maps to the same set.

Solution: The two least significant bits of the address are 00, because the address is word aligned. The next three bits are 101, so the word maps to set 5. Words at addresses 0x34, 0x54, 0x74, . . . , 0xFFFFFFF4 all map to this same set.

Because many addresses map to a single set, the cache must also keep track of the address of the data actually contained in each set. The least significant bits of the address specify which set holds the data. The remaining most significant bits are called the *tag* and indicate which of the many possible addresses is held in that set.

In our previous example, the two least significant bits of the 32-bit address are called the *byte offset*, because they indicate the byte within the word. The next three bits are called the *set bits*, because they indicate the set to which the address maps. (In general, the number of set bits is $\log_2 S$.) The remaining 27 tag bits indicate the memory address of the data stored in a given cache set. Figure 8.6 shows the cache fields for address 0xFFFFFFE4. It maps to set 1 and its tag is all 1's.

Figure 8.6 Cache fields for address 0xFFFFFFE4 when mapping to the cache in Figure 8.5

Example 8.5 CACHE FIELDS

Find the number of set and tag bits for a direct mapped cache with 1024 (2^{10}) sets and a one-word block size. The address size is 32 bits.

Solution: A cache with 2^{10} sets requires $\log_2(2^{10}) = 10$ set bits. The two least significant bits of the address are the byte offset, and the remaining $32 - 10 - 2 = 20$ bits form the tag.

Sometimes, such as when the computer first starts up, the cache sets contain no data at all. The cache uses a *valid bit* for each set to indicate whether the set holds meaningful data. If the valid bit is 0, the contents are meaningless.

Figure 8.7 shows the hardware for the direct mapped cache of Figure 8.5. The cache is constructed as an eight-entry SRAM. Each entry, or set, contains one line consisting of 32 bits of data, 27 bits of tag, and 1 valid bit. The cache is accessed using the 32-bit address. The two least significant bits, the byte offset bits, are ignored for word accesses. The next three bits, the set bits, specify the entry or set in the cache. A load instruction reads the specified entry from the cache and checks the tag and valid bits. If the tag matches the most significant

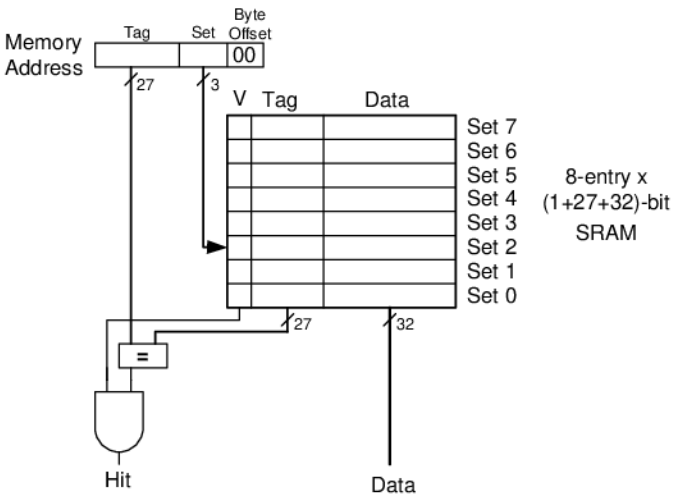


Figure 8.7 Direct mapped cache with 8 sets

27 bits of the address and the valid bit is 1, the cache hits and the data is returned to the processor. Otherwise, the cache misses and the memory system must fetch the data from main memory.

Example 8.6 TEMPORAL LOCALITY WITH A DIRECT MAPPED CACHE

Loops are a common source of temporal and spatial locality in applications. Using the eight-entry cache of Figure 8.7, show the contents of the cache after executing the following silly loop in MIPS assembly code. Assume that the cache is initially empty. What is the miss rate?

```
      addi $t0, $0, 5
loop: beq  $t0, $0, done
      lw   $t1, 0x4($0)
      lw   $t2, 0xC($0)
      lw   $t3, 0x8($0)
      addi $t0, $t0, -1
      j    loop
done:
```

Solution: The program contains a loop that repeats for five iterations. Each iteration involves three memory accesses (loads), resulting in 15 total memory accesses. The first time the loop executes, the cache is empty and the data must be fetched from main memory locations 0x4, 0xC, and 0x8 into cache sets 1, 3, and 2, respectively. However, the next four times the loop executes, the data is found in the cache. Figure 8.8 shows the contents of the cache during the last request to memory address 0x4. The tags are all 0 because the upper 27 bits of the addresses are 0. The miss rate is 3/15 = 20%.

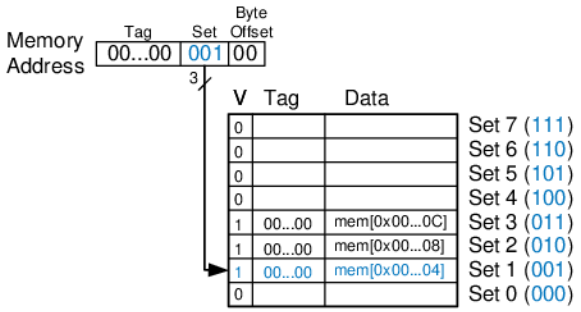


Figure 8.8 Direct mapped cache contents

When two recently accessed addresses map to the same cache block, a *conflict* occurs, and the most recently accessed address *evicts* the previous one from the block. Direct mapped caches have only one block in each set, so two addresses that map to the same set always cause a conflict. The example on the next page illustrates conflicts.

Example 8.7 CACHE BLOCK CONFLICT

What is the miss rate when the following loop is executed on the eight-word direct mapped cache from Figure 8.7? Assume that the cache is initially empty.

```
      addi  $t0, $0, 5
loop: beq   $t0, $0, done
      lw    $t1, 0x4($0)
      lw    $t2, 0x24($0)
      addi  $t0, $t0, -1
      j     loop
done:
```

Solution: Memory addresses 0x4 and 0x24 both map to set 1. During the initial execution of the loop, data at address 0x4 is loaded into set 1 of the cache. Then data at address 0x24 is loaded into set 1, evicting the data from address 0x4. Upon the second execution of the loop, the pattern repeats and the cache must refetch data at address 0x4, evicting data from address 0x24. The two addresses conflict, and the miss rate is 100%.

Multi-way Set Associative Cache

An *N-way set associative* cache reduces conflicts by providing N blocks in each set where data mapping to that set might be found. Each memory address still maps to a specific set, but it can map to any one of the N blocks in the set. Hence, a direct mapped cache is another name for a one-way set associative cache. N is also called the *degree of associativity* of the cache.

Figure 8.9 shows the hardware for a $C = 8$-word, $N = 2$-way set associative cache. The cache now has only $S = 4$ sets rather than 8.

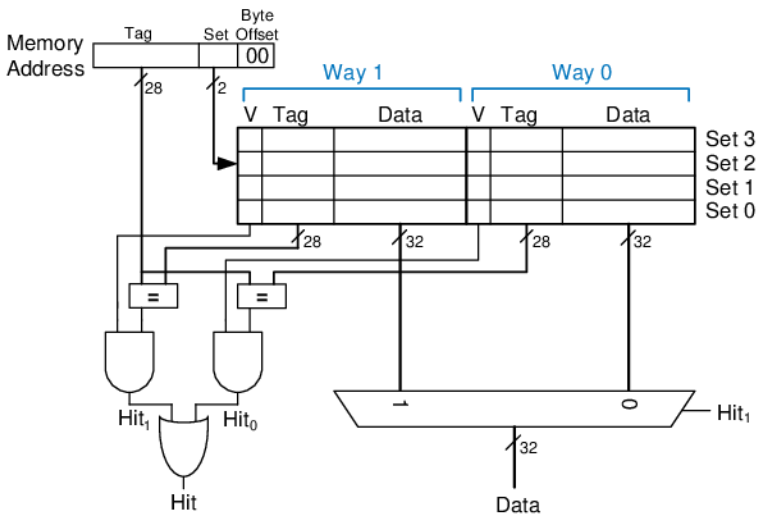


Figure 8.9 Two-way set associative cache

Thus, only $\log_2 4 = 2$ set bits rather than 3 are used to select the set. The tag increases from 27 to 28 bits. Each set contains two *ways* or degrees of associativity. Each way consists of a data block and the valid and tag bits. The cache reads blocks from both ways in the selected set and checks the tags and valid bits for a hit. If a hit occurs in one of the ways, a multiplexer selects data from that way.

Set associative caches generally have lower miss rates than direct mapped caches of the same capacity, because they have fewer conflicts. However, set associative caches are usually slower and somewhat more expensive to build because of the output multiplexer and additional comparators. They also raise the question of which way to replace when both ways are full; this is addressed further in Section 8.3.3. Most commercial systems use set associative caches.

Example 8.8 SET ASSOCIATIVE CACHE MISS RATE

Repeat Example 8.7 using the eight-word two-way set associative cache from Figure 8.9.

Solution: Both memory accesses, to addresses 0x4 and 0x24, map to set 1. However, the cache has two ways, so it can accommodate data from both addresses. During the first loop iteration, the empty cache misses both addresses and loads both words of data into the two ways of set 1, as shown in Figure 8.10. On the next four iterations, the cache hits. Hence, the miss rate is 2/10 = 20%. Recall that the direct mapped cache of the same size from Example 8.7 had a miss rate of 100%.

Figure 8.10 Two-way set associative cache contents

Fully Associative Cache

A *fully associative* cache contains a single set with B ways, where B is the number of blocks. A memory address can map to a block in any of these ways. A fully associative cache is another name for a B-way set associative cache with one set.

Figure 8.11 shows the SRAM array of a fully associative cache with eight blocks. Upon a data request, eight tag comparisons (not shown) must be made, because the data could be in any block. Similarly, an 8:1 multiplexer chooses the proper data if a hit occurs. Fully associative caches tend

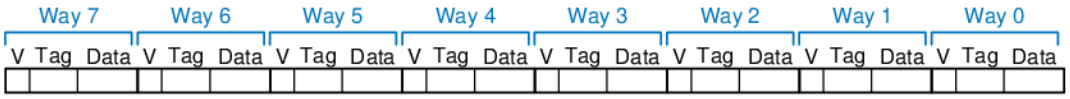


Figure 8.11 Eight-block fully associative cache

to have the fewest conflict misses for a given cache capacity, but they require more hardware for additional tag comparisons. They are best suited to relatively small caches because of the large number of comparators.

Block Size

The previous examples were able to take advantage only of temporal locality, because the block size was one word. To exploit spatial locality, a cache uses larger blocks to hold several consecutive words.

The advantage of a block size greater than one is that when a miss occurs and the word is fetched into the cache, the adjacent words in the block are also fetched. Therefore, subsequent accesses are more likely to hit because of spatial locality. However, a large block size means that a fixed-size cache will have fewer blocks. This may lead to more conflicts, increasing the miss rate. Moreover, it takes more time to fetch the missing cache block after a miss, because more than one data word is fetched from main memory. The time required to load the missing block into the cache is called the *miss penalty*. If the adjacent words in the block are not accessed later, the effort of fetching them is wasted. Nevertheless, most real programs benefit from larger block sizes.

Figure 8.12 shows the hardware for a $C = 8$-word direct mapped cache with a $b = 4$-word block size. The cache now has only $B = C/b = 2$ blocks. A direct mapped cache has one block in each set, so this cache is

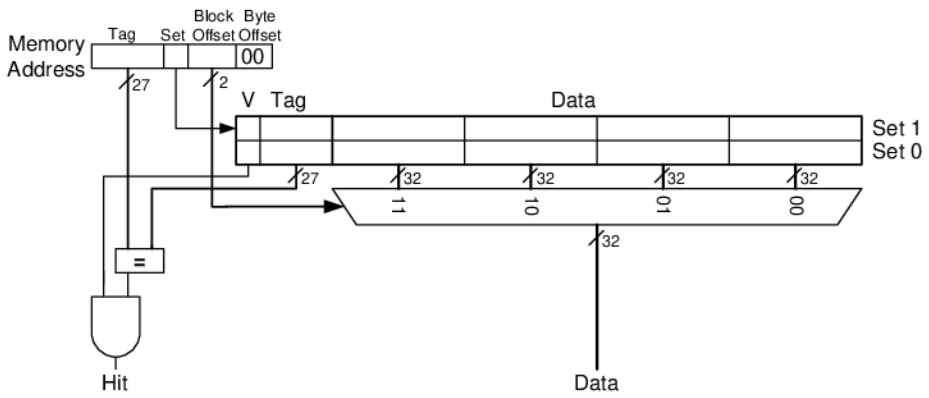


Figure 8.12 Direct mapped cache with two sets and a four-word block size

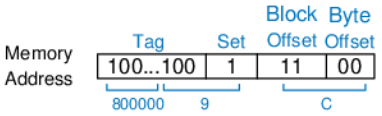


Figure 8.13 Cache fields for address 0x8000009C when mapping to the cache of Figure 8.12

organized as two sets. Thus, only $\log_2 2 = 1$ bit is used to select the set. A multiplexer is now needed to select the word within the block. The multiplexer is controlled by the $\log_2 4 = 2$ *block offset bits* of the address. The most significant 27 address bits form the tag. Only one tag is needed for the entire block, because the words in the block are at consecutive addresses.

Figure 8.13 shows the cache fields for address 0x8000009C when it maps to the direct mapped cache of Figure 8.12. The byte offset bits are always 0 for word accesses. The next $\log_2 b = 2$ block offset bits indicate the word within the block. And the next bit indicates the set. The remaining 27 bits are the tag. Therefore, word 0x8000009C maps to set 1, word 3 in the cache. The principle of using larger block sizes to exploit spatial locality also applies to associative caches.

Example 8.9 SPATIAL LOCALITY WITH A DIRECT MAPPED CACHE

Repeat Example 8.6 for the eight-word direct mapped cache with a four-word block size.

Solution: Figure 8.14 shows the contents of the cache after the first memory access. On the first loop iteration, the cache misses on the access to memory address 0x4. This access loads data at addresses 0x0 through 0xC into the cache block. All subsequent accesses (as shown for address 0xC) hit in the cache. Hence, the miss rate is 1/15 = 6.67%.

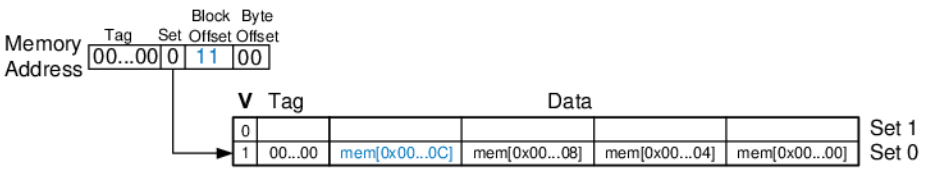


Figure 8.14 Cache contents with a block size (*b*) of four words

Putting It All Together

Caches are organized as two-dimensional arrays. The rows are called sets, and the columns are called ways. Each entry in the array consists of

Table 8.2 Cache organizations

Organization	Number of Ways (N)	Number of Sets (S)
direct mapped	1	B
set associative	$1 < N < B$	B/N
fully associative	B	1

a data block and its associated valid and tag bits. Caches are characterized by

- capacity C
- block size b (and number of blocks, $B = C/b$)
- number of blocks in a set (N)

Table 8.2 summarizes the various cache organizations. Each address in memory maps to only one set but can be stored in any of the ways.

Cache capacity, associativity, set size, and block size are typically powers of 2. This makes the cache fields (tag, set, and block offset bits) subsets of the address bits.

Increasing the associativity, N, usually reduces the miss rate caused by conflicts. But higher associativity requires more tag comparators. Increasing the block size, b, takes advantage of spatial locality to reduce the miss rate. However, it decreases the number of sets in a fixed sized cache and therefore could lead to more conflicts. It also increases the miss penalty.

8.3.3 What Data Is Replaced?

In a direct mapped cache, each address maps to a unique block and set. If a set is full when new data must be loaded, the block in that set is replaced with the new data. In set associative and fully associative caches, the cache must choose which block to evict when a cache set is full. The principle of temporal locality suggests that the best choice is to evict the least recently used block, because it is least likely to be used again soon. Hence, most associative caches have a *least recently used* (*LRU*) replacement policy.

In a two-way set associative cache, a *use bit*, U, indicates which way within a set was least recently used. Each time one of the ways is used, U is adjusted to indicate the other way. For set associative caches with more than two ways, tracking the least recently used way becomes complicated. To simplify the problem, the ways are often divided into two groups and U indicates which *group* of ways was least recently used.

Upon replacement, the new block replaces a random block within the least recently used group. Such a policy is called *pseudo-LRU* and is good enough in practice.

Example 8.10 LRU REPLACEMENT

Show the contents of an eight-word two-way set associative cache after executing the following code. Assume LRU replacement, a block size of one word, and an initially empty cache.

```
lw $t0, 0x04($0)
lw $t1, 0x24($0)
lw $t2, 0x54($0)
```

Solution: The first two instructions load data from memory addresses 0x4 and 0x24 into set 1 of the cache, shown in Figure 8.15(a). $U = 0$ indicates that data in way 0 was the least recently used. The next memory access, to address 0x54, also maps to set 1 and replaces the least recently used data in way 0, as shown in Figure 8.15(b), The use bit, U, is set to 1 to indicate that data in way 1 was the least recently used.

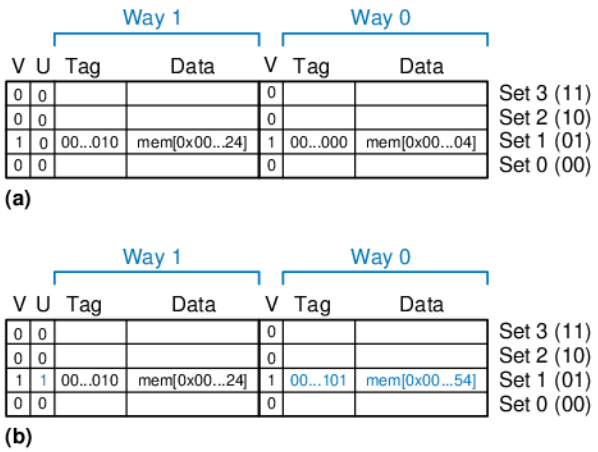


Figure 8.15 Two-way associative cache with LRU replacement

8.3.4 Advanced Cache Design*

Modern systems use multiple levels of caches to decrease memory access time. This section explores the performance of a two-level caching system and examines how block size, associativity, and cache capacity affect miss rate. The section also describes how caches handle stores, or writes, by using a write-through or write-back policy.

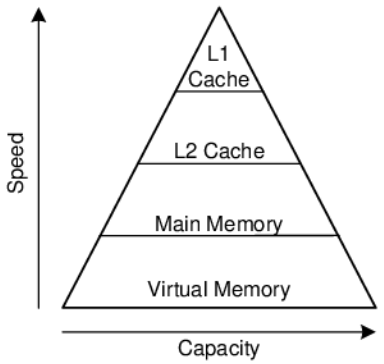


Figure 8.16 Memory hierarchy with two levels of cache

Multiple-Level Caches

Large caches are beneficial because they are more likely to hold data of interest and therefore have lower miss rates. However, large caches tend to be slower than small ones. Modern systems often use two levels of caches, as shown in Figure 8.16. The first-level (L1) cache is small enough to provide a one- or two-cycle access time. The second-level (L2) cache is also built from SRAM but is larger, and therefore slower, than the L1 cache. The processor first looks for the data in the L1 cache. If the L1 cache misses, the processor looks in the L2 cache. If the L2 cache misses, the processor fetches the data from main memory. Some modern systems add even more levels of cache to the memory hierarchy, because accessing main memory is so slow.

Example 8.11 SYSTEM WITH AN L2 CACHE

Use the system of Figure 8.16 with access times of 1, 10, and 100 cycles for the L1 cache, L2 cache, and main memory, respectively. Assume that the L1 and L2 caches have miss rates of 5% and 20%, respectively. Specifically, of the 5% of accesses that miss the L1 cache, 20% of those also miss the L2 cache. What is the average memory access time (*AMAT*)?

Solution: Each memory access checks the L1 cache. When the L1 cache misses (5% of the time), the processor checks the L2 cache. When the L2 cache misses (20% of the time), the processor fetches the data from main memory. Using Equation 8.2, we calculate the average memory access time as follows:
1 cycle + 0.05[10 cycles + 0.2(100 cycles)] = 2.5 cycles

The L2 miss rate is high because it receives only the "hard" memory accesses, those that miss in the L1 cache. If all accesses went directly to the L2 cache, the L2 miss rate would be about 1%.

Reducing Miss Rate

Cache misses can be reduced by changing capacity, block size, and/or associativity. The first step to reducing the miss rate is to understand the causes of the misses. The misses can be classified as compulsory, capacity, and conflict. The first request to a cache block is called a *compulsory miss*, because the block must be read from memory regardless of the cache design. *Capacity misses* occur when the cache is too small to hold all concurrently used data. *Conflict misses* are caused when several addresses map to the same set and evict blocks that are still needed.

Changing cache parameters can affect one or more type of cache miss. For example, increasing cache capacity can reduce conflict and capacity misses, but it does not affect compulsory misses. On the other hand, increasing block size could reduce compulsory misses (due to spatial locality) but might actually *increase* conflict misses (because more addresses would map to the same set and could conflict).

Memory systems are complicated enough that the best way to evaluate their performance is by running benchmarks while varying cache parameters. Figure 8.17 plots miss rate versus cache size and degree of associativity for the SPEC2000 benchmark. This benchmark has a small number of compulsory misses, shown by the dark region near the x-axis. As expected, when cache size increases, capacity misses decrease. Increased associativity, especially for small caches, decreases the number of conflict misses shown along the top of the curve.

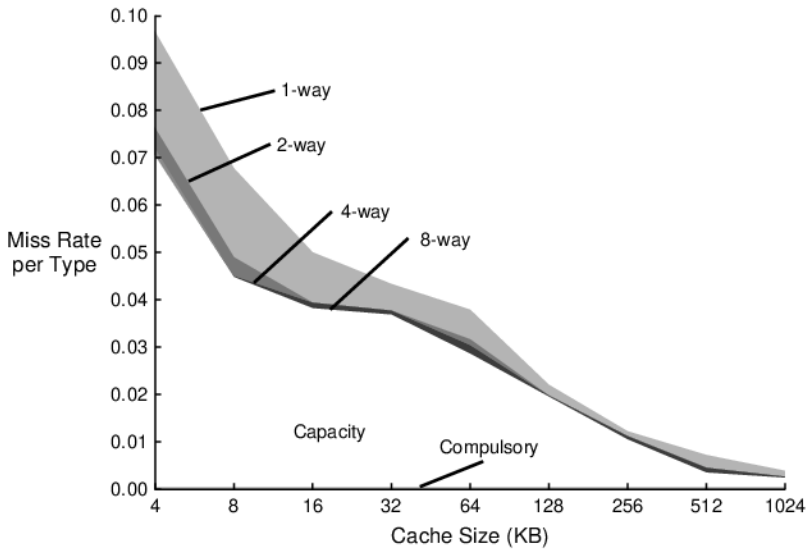


Figure 8.17 Miss rate versus cache size and associativity on SPEC2000 benchmark
Adapted with permission from Hennessy and Patterson, *Computer Architecture: A Quantitative Approach*, 3rd ed., Morgan Kaufmann, 2003.

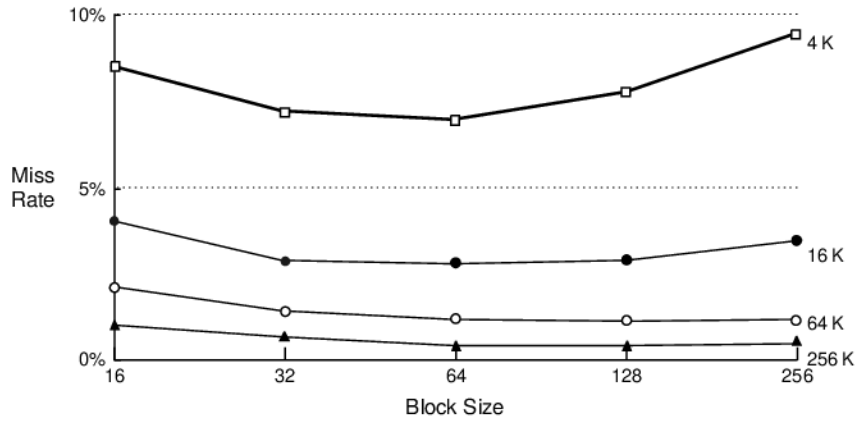


Figure 8.18 Miss rate versus block size and cache size on SPEC92 benchmark Adapted with permission from Hennessy and Patterson, *Computer Architecture: A Quantitative Approach*, 3rd ed., Morgan Kaufmann, 2003.

Increasing associativity beyond four or eight ways provides only small decreases in miss rate.

As mentioned, miss rate can also be decreased by using larger block sizes that take advantage of spatial locality. But as block size increases, the number of sets in a fixed size cache decreases, increasing the probability of conflicts. Figure 8.18 plots miss rate versus block size (in number of bytes) for caches of varying capacity. For small caches, such as the 4-KB cache, increasing the block size beyond 64 bytes *increases* the miss rate because of conflicts. For larger caches, increasing the block size does not change the miss rate. However, large block sizes might still increase execution time because of the larger miss penalty, the time required to fetch the missing cache block from main memory on a miss.

Write Policy

The previous sections focused on memory loads. Memory stores, or writes, follow a similar procedure as loads. Upon a memory store, the processor checks the cache. If the cache misses, the cache block is fetched from main memory into the cache, and then the appropriate word in the cache block is written. If the cache hits, the word is simply written to the cache block.

Caches are classified as either write-through or write-back. In a *write-through* cache, the data written to a cache block is simultaneously written to main memory. In a *write-back* cache, a *dirty bit* (D) is associated with each cache block. D is 1 when the cache block has been written and 0 otherwise. Dirty cache blocks are written back to main memory only when they are evicted from the cache. A write-through

cache requires no dirty bit but usually requires more main memory writes than a write-back cache. Modern caches are usually write-back, because main memory access time is so large.

Example 8.12 WRITE-THROUGH VERSUS WRITE-BACK

Suppose a cache has a block size of four words. How many main memory accesses are required by the following code when using each write policy: write-through or write-back?

```
sw $t0, 0x0($0)
sw $t0, 0xC($0)
sw $t0, 0x8($0)
sw $t0, 0x4($0)
```

Solution: All four store instructions write to the same cache block. With a write-through cache, each store instruction writes a word to main memory, requiring four main memory writes. A write-back policy requires only one main memory access, when the dirty cache block is evicted.

8.3.5 The Evolution of MIPS Caches*

Table 8.3 traces the evolution of cache organizations used by the MIPS processor from 1985 to 2004. The major trends are the introduction of multiple levels of cache, larger cache capacity, and increased associativity. These trends are driven by the growing disparity between CPU frequency and main memory speed and the decreasing cost of transistors. The growing difference between CPU and memory speeds necessitates a lower miss rate to avoid the main memory bottleneck, and the decreasing cost of transistors allows larger cache sizes.

Table 8.3 MIPS cache evolution*

Year	CPU	MHz	L1 Cache	L2 Cache
1985	R2000	16.7	none	none
1990	R3000	33	32 KB direct mapped	none
1991	R4000	100	8 KB direct mapped	1 MB direct mapped
1995	R10000	250	32 KB two-way	4 MB two-way
2001	R14000	600	32 KB two-way	16 MB two-way
2004	R16000A	800	64 KB two-way	16 MB two-way

* Adapted from D. Sweetman, *See MIPS Run*, Morgan Kaufmann, 1999.

8.4 VIRTUAL MEMORY

Most modern computer systems use a *hard disk* (also called a *hard drive*) as the lowest level in the memory hierarchy (see Figure 8.4). Compared with the ideal large, fast, cheap memory, a hard disk is large and cheap but terribly slow. The disk provides a much larger capacity than is possible with a cost-effective main memory (DRAM). However, if a significant fraction of memory accesses involve the disk, performance is dismal. You may have encountered this on a PC when running too many programs at once.

Figure 8.19 shows a hard disk with the lid of its case removed. As the name implies, the hard disk contains one or more rigid disks or *platters*, each of which has a *read/write head* on the end of a long triangular arm. The head moves to the correct location on the disk and reads or writes data magnetically as the disk rotates beneath it. The head takes several milliseconds to *seek* the correct location on the disk, which is fast from a human perspective but millions of times slower than the processor.

Figure 8.19 Hard disk

The objective of adding a hard disk to the memory hierarchy is to inexpensively give the illusion of a very large memory while still providing the speed of faster memory for most accesses. A computer with only 128 MB of DRAM, for example, could effectively provide 2 GB of memory using the hard disk. This larger 2-GB memory is called *virtual memory*, and the smaller 128-MB main memory is called *physical memory*. We will use the term physical memory to refer to main memory throughout this section.

A computer with 32-bit addresses can access a maximum of 2^{32} bytes = 4 GB of memory. This is one of the motivations for moving to 64-bit computers, which can access far more memory.

Programs can access data anywhere in virtual memory, so they must use *virtual addresses* that specify the location in virtual memory. The physical memory holds a subset of most recently accessed virtual memory. In this way, physical memory acts as a cache for virtual memory. Thus, most accesses hit in physical memory at the speed of DRAM, yet the program enjoys the capacity of the larger virtual memory.

Virtual memory systems use different terminologies for the same caching principles discussed in Section 8.3. Table 8.4 summarizes the analogous terms. Virtual memory is divided into *virtual pages*, typically 4 KB in size. Physical memory is likewise divided into *physical pages* of the same size. A virtual page may be located in physical memory (DRAM) or on the disk. For example, Figure 8.20 shows a virtual memory that is larger than physical memory. The rectangles indicate pages. Some virtual pages are present in physical memory, and some are located on the disk. The process of determining the physical address from the virtual address is called *address translation*. If the processor attempts to access a virtual address that is not in physical memory, a *page fault* occurs, and the operating system loads the page from the hard disk into physical memory.

To avoid page faults caused by conflicts, any virtual page can map to any physical page. In other words, physical memory behaves as a fully associative cache for virtual memory. In a conventional fully associative cache, every cache block has a comparator that checks the most significant address bits against a tag to determine whether the request hits in

Table 8.4 Analogous cache and virtual memory terms

Cache	Virtual Memory
Block	Page
Block size	Page size
Block offset	Page offset
Miss	Page fault
Tag	Virtual page number

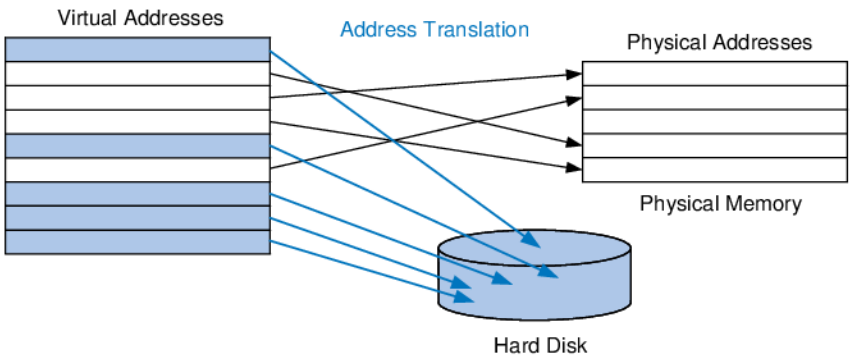


Figure 8.20 Virtual and physical pages

the block. In an analogous virtual memory system, each physical page would need a comparator to check the most significant virtual address bits against a tag to determine whether the virtual page maps to that physical page.

A realistic virtual memory system has so many physical pages that providing a comparator for each page would be excessively expensive. Instead, the virtual memory system uses a page table to perform address translation. A page table contains an entry for each virtual page, indicating its location in physical memory or that it is on the disk. Each load or store instruction requires a page table access followed by a physical memory access. The page table access translates the virtual address used by the program to a physical address. The physical address is then used to actually read or write the data.

The page table is usually so large that it is located in physical memory. Hence, each load or store involves two physical memory accesses: a page table access, and a data access. To speed up address translation, a translation lookaside buffer (TLB) caches the most commonly used page table entries.

The remainder of this section elaborates on address translation, page tables, and TLBs.

8.4.1 Address Translation

In a system with virtual memory, programs use virtual addresses so that they can access a large memory. The computer must translate these virtual addresses to either find the address in physical memory or take a page fault and fetch the data from the hard disk.

Recall that virtual memory and physical memory are divided into pages. The most significant bits of the virtual or physical address specify the virtual or physical *page number*. The least significant bits specify the word within the page and are called the *page offset*.

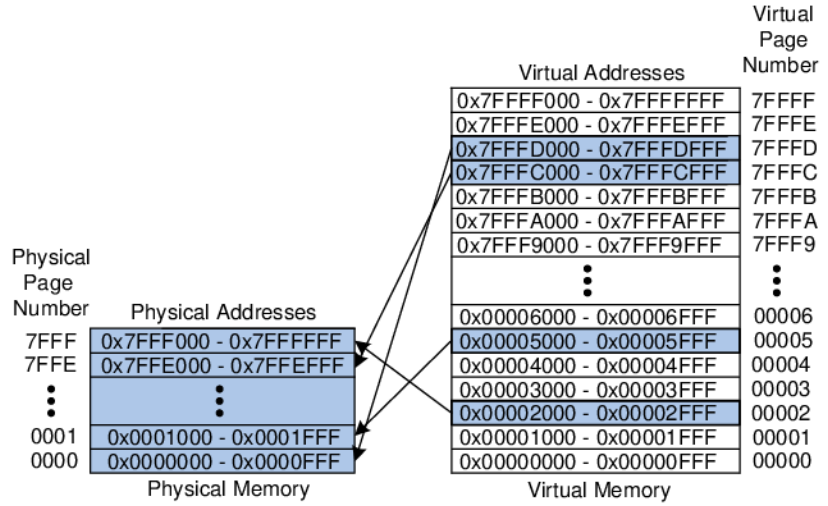


Figure 8.21 **Physical and virtual pages**

Figure 8.21 illustrates the page organization of a virtual memory system with 2 GB of virtual memory and 128 MB of physical memory divided into 4-KB pages. MIPS accommodates 32-bit addresses. With a 2-GB = 2^{31}-byte virtual memory, only the least significant 31 virtual address bits are used; the 32nd bit is always 0. Similarly, with a 128-MB = 2^{27}-byte physical memory, only the least significant 27 physical address bits are used; the upper 5 bits are always 0.

Because the page size is 4 KB = 2^{12} bytes, there are $2^{31}/2^{12} = 2^{19}$ virtual pages and $2^{27}/2^{12} = 2^{15}$ physical pages. Thus, the virtual and physical page numbers are 19 and 15 bits, respectively. Physical memory can only hold up to 1/16th of the virtual pages at any given time. The rest of the virtual pages are kept on disk.

Figure 8.21 shows virtual page 5 mapping to physical page 1, virtual page 0x7FFFC mapping to physical page 0x7FFE, and so forth. For example, virtual address 0x53F8 (an offset of 0x3F8 within virtual page 5) maps to physical address 0x13F8 (an offset of 0x3F8 within physical page 1). The least significant 12 bits of the virtual and physical addresses are the same (0x3F8) and specify the page offset within the virtual and physical pages. Only the page number needs to be translated to obtain the physical address from the virtual address.

Figure 8.22 illustrates the translation of a virtual address to a physical address. The least significant 12 bits indicate the page offset and require no translation. The upper 19 bits of the virtual address specify the *virtual page number* (*VPN*) and are translated to a 15-bit *physical page number* (*PPN*). The next two sections describe how page tables and TLBs are used to perform this address translation.

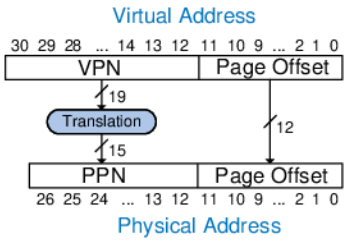


Figure 8.22 Translation from virtual address to physical address

Example 8.13 VIRTUAL ADDRESS TO PHYSICAL ADDRESS TRANSLATION

Find the physical address of virtual address 0x247C using the virtual memory system shown in Figure 8.21.

Solution: The 12-bit page offset (0x47C) requires no translation. The remaining 19 bits of the virtual address give the virtual page number, so virtual address 0x247C is found in virtual page 0x2. In Figure 8.21, virtual page 0x2 maps to physical page 0x7FFF. Thus, virtual address 0x247C maps to physical address 0x7FFF47C.

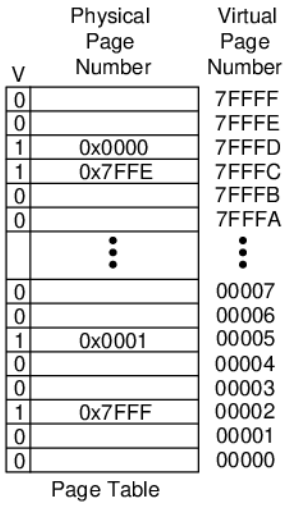


Figure 8.23 The page table for Figure 8.21

8.4.2 The Page Table

The processor uses a *page table* to translate virtual addresses to physical addresses. Recall that the page table contains an entry for each virtual page. This entry contains a physical page number and a valid bit. If the valid bit is 1, the virtual page maps to the physical page specified in the entry. Otherwise, the virtual page is found on disk.

Because the page table is so large, it is stored in physical memory. Let us assume for now that it is stored as a contiguous array, as shown in Figure 8.23. This page table contains the mapping of the memory system of Figure 8.21. The page table is indexed with the virtual page number (VPN). For example, entry 5 specifies that virtual page 5 maps to physical page 1. Entry 6 is invalid ($V = 0$), so virtual page 6 is located on disk.

Example 8.14 USING THE PAGE TABLE TO PERFORM ADDRESS TRANSLATION

Find the physical address of virtual address 0x247C using the page table shown in Figure 8.23.

Solution: Figure 8.24 shows the virtual address to physical address translation for virtual address 0x247C. The 12-bit page offset requires no translation. The remaining 19 bits of the virtual address are the virtual page number, 0x2, and

give the index into the page table. The page table maps virtual page 0x2 to physical page 0x7FFF. So, virtual address 0x247C maps to physical address 0x7FFF47C. The least significant 12 bits are the same in both the physical and the virtual address.

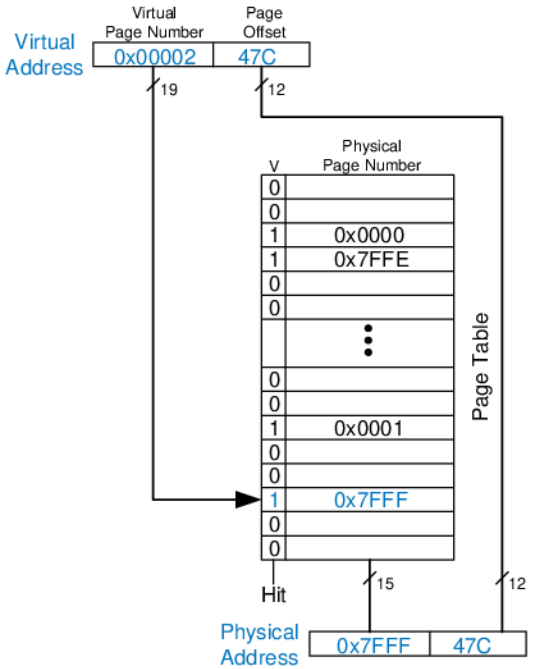


Figure 8.24 Address translation using the page table

The page table can be stored anywhere in physical memory, at the discretion of the OS. The processor typically uses a dedicated register, called the *page table register*, to store the base address of the page table in physical memory.

To perform a load or store, the processor must first translate the virtual address to a physical address and then access the data at that physical address. The processor extracts the virtual page number from the virtual address and adds it to the page table register to find the physical address of the page table entry. The processor then reads this page table entry from physical memory to obtain the physical page number. If the entry is valid, it merges this physical page number with the page offset to create the physical address. Finally, it reads or writes data at this physical address. Because the page table is stored in physical memory, each load or store involves two physical memory accesses.

8.4.3 The Translation Lookaside Buffer

Virtual memory would have a severe performance impact if it required a page table read on every load or store, doubling the delay of loads and stores. Fortunately, page table accesses have great temporal locality. The temporal and spatial locality of data accesses and the large page size mean that many consecutive loads or stores are likely to reference the same page. Therefore, if the processor remembers the last page table entry that it read, it can probably reuse this translation without rereading the page table. In general, the processor can keep the last several page table entries in a small cache called a *translation lookaside buffer* (*TLB*). The processor "looks aside" to find the translation in the TLB before having to access the page table in physical memory. In real programs, the vast majority of accesses hit in the TLB, avoiding the time-consuming page table reads from physical memory.

A TLB is organized as a fully associative cache and typically holds 16 to 512 entries. Each TLB entry holds a virtual page number and its corresponding physical page number. The TLB is accessed using the virtual page number. If the TLB hits, it returns the corresponding physical page number. Otherwise, the processor must read the page table in physical memory. The TLB is designed to be small enough that it can be accessed in less than one cycle. Even so, TLBs typically have a hit rate of greater than 99%. The TLB decreases the number of memory accesses required for most load or store instructions from two to one.

Example 8.15 USING THE TLB TO PERFORM ADDRESS TRANSLATION

Consider the virtual memory system of Figure 8.21. Use a two-entry TLB or explain why a page table access is necessary to translate virtual addresses 0x247C and 0x5FB0 to physical addresses. Suppose the TLB currently holds valid translations of virtual pages 0x2 and 0x7FFD.

Solution: Figure 8.25 shows the two-entry TLB with the request for virtual address 0x247C. The TLB receives the virtual page number of the incoming address, 0x2, and compares it to the virtual page number of each entry. Entry 0 matches and is valid, so the request hits. The translated physical address is the physical page number of the matching entry, 0x7FFF, concatenated with the page offset of the virtual address. As always, the page offset requires no translation.

The request for virtual address 0x5FB0 misses in the TLB. So, the request is forwarded to the page table for translation.

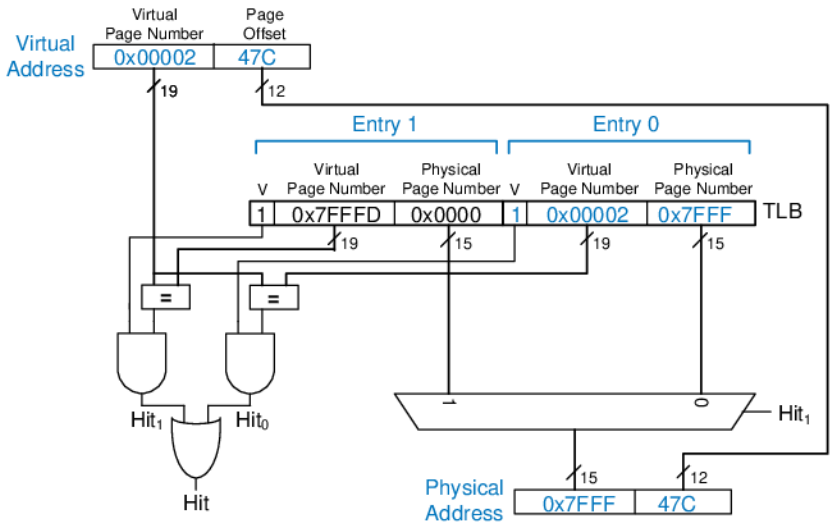


Figure 8.25 Address translation using a two-entry TLB

8.4.4 Memory Protection

So far this section has focused on using virtual memory to provide a fast, inexpensive, large memory. An equally important reason to use virtual memory is to provide protection between concurrently running programs.

As you probably know, modern computers typically run several programs or *processes* at the same time. All of the programs are simultaneously present in physical memory. In a well-designed computer system, the programs should be protected from each other so that no program can crash or hijack another program. Specifically, no program should be able to access another program's memory without permission. This is called *memory protection*.

Virtual memory systems provide memory protection by giving each program its own *virtual address space*. Each program can use as much memory as it wants in that virtual address space, but only a portion of the virtual address space is in physical memory at any given time. Each program can use its entire virtual address space without having to worry about where other programs are physically located. However, a program can access only those physical pages that are mapped in its page table. In this way, a program cannot accidentally or maliciously access another program's physical pages, because they are not mapped in its page table. In some cases, multiple programs access common instructions or data. The operating system adds control bits to each page table entry to determine which programs, if any, can write to the shared physical pages.

8.4.5 Replacement Policies*

Virtual memory systems use write-back and an approximate least recently used (LRU) replacement policy. A write-through policy, where each write to physical memory initiates a write to disk, would be impractical. Store instructions would operate at the speed of the disk instead of the speed of the processor (milliseconds instead of nanoseconds). Under the write-back policy, the physical page is written back to disk only when it is evicted from physical memory. Writing the physical page back to disk and reloading it with a different virtual page is called *swapping*, so the disk in a virtual memory system is sometimes called *swap space*. The processor swaps out one of the least recently used physical pages when a page fault occurs, then replaces that page with the missing virtual page. To support these replacement policies, each page table entry contains two additional status bits: a dirty bit, D, and a use bit, U.

The dirty bit is 1 if any store instructions have changed the physical page since it was read from disk. When a physical page is swapped out, it needs to be written back to disk only if its dirty bit is 1; otherwise, the disk already holds an exact copy of the page.

The use bit is 1 if the physical page has been accessed recently. As in a cache system, exact LRU replacement would be impractically complicated. Instead, the OS approximates LRU replacement by periodically resetting all the use bits in the page table. When a page is accessed, its use bit is set to 1. Upon a page fault, the OS finds a page with $U = 0$ to swap out of physical memory. Thus, it does not necessarily replace the least recently used page, just one of the least recently used pages.

8.4.6 Multilevel Page Tables*

Page tables can occupy a large amount of physical memory. For example, the page table from the previous sections for a 2 GB virtual memory with 4 KB pages would need 2^{19} entries. If each entry is 4 bytes, the page table is $2^{19} \times 2^2$ bytes $= 2^{21}$ bytes $= 2$ MB.

To conserve physical memory, page tables can be broken up into multiple (usually two) levels. The first-level page table is always kept in physical memory. It indicates where small second-level page tables are stored in virtual memory. The second-level page tables each contain the actual translations for a range of virtual pages. If a particular range of translations is not actively used, the corresponding second-level page table can be swapped out to the hard disk so it does not waste physical memory.

In a two-level page table, the virtual page number is split into two parts: the *page table number* and the *page table offset*, as shown in Figure 8.26. The page table number indexes the first-level page table, which must reside in physical memory. The first-level page table entry gives the base address of the second-level page table or indicates that

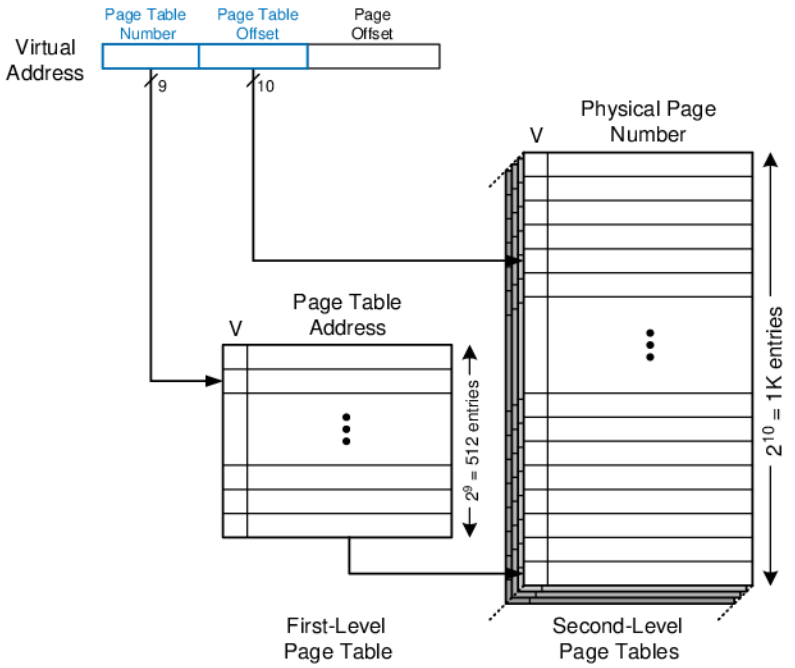


Figure 8.26 Hierarchical page tables

it must be fetched from disk when V is 0. The page table offset indexes the second-level page table. The remaining 12 bits of the virtual address are the page offset, as before, for a page size of $2^{12} = 4$ KB.

In Figure 8.26 the 19-bit virtual page number is broken into 9 and 10 bits, to indicate the page table number and the page table offset, respectively. Thus, the first-level page table has $2^9 = 512$ entries. Each of these 512 second-level page tables has $2^{10} = 1$ K entries. If each of the first- and second-level page table entries is 32 bits (4 bytes) and only two second-level page tables are present in physical memory at once, the hierarchical page table uses only (512 × 4 bytes) + 2 × (1 K × 4 bytes) = 10 KB of physical memory. The two-level page table requires a fraction of the physical memory needed to store the entire page table (2 MB). The drawback of a two-level page table is that it adds yet another memory access for translation when the TLB misses.

Example 8.16 USING A MULTILEVEL PAGE TABLE FOR ADDRESS TRANSLATION

Figure 8.27 shows the possible contents of the two-level page table from Figure 8.26. The contents of only one second-level page table are shown. Using this two-level page table, describe what happens on an access to virtual address 0x003FEFB0.

Solution: As always, only the virtual page number requires translation. The most significant nine bits of the virtual address, 0x0, give the page table number, the index into the first-level page table. The first-level page table at entry 0x0 indicates that the second-level page table is resident in memory ($V = 1$) and its physical address is 0x2375000.

The next ten bits of the virtual address, 0x3FE, are the page table offset, which gives the index into the second-level page table. Entry 0 is at the bottom of the second-level page table, and entry 0x3FF is at the top. Entry 0x3FE in the second-level page table indicates that the virtual page is resident in physical memory ($V = 1$) and that the physical page number is 0x23F1. The physical page number is concatenated with the page offset to form the physical address, 0x23F1FB0.

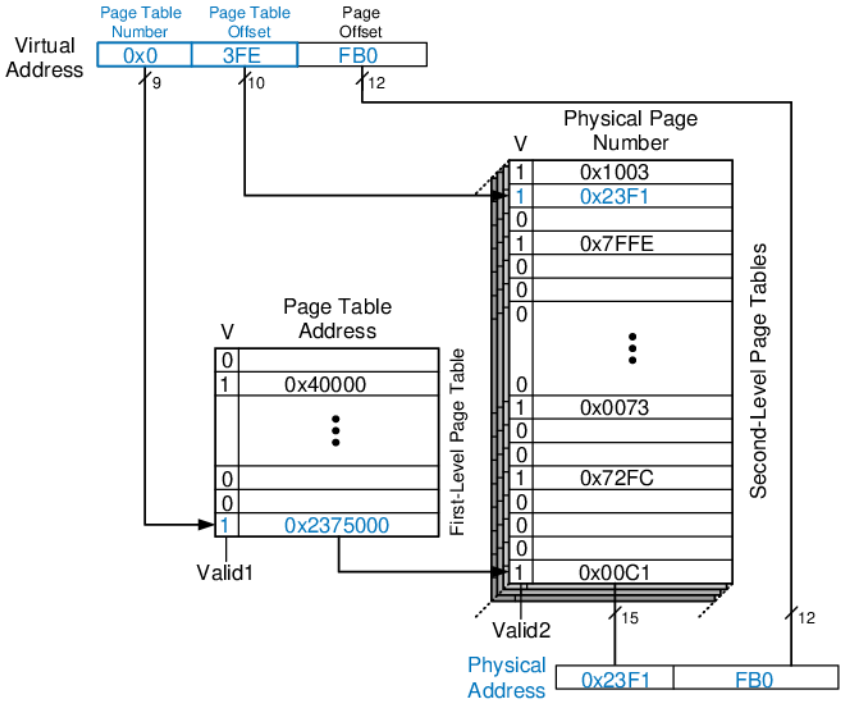


Figure 8.27 Address translation using a two-level page table

8.5 MEMORY-MAPPED I/O*

Processors also use the memory interface to communicate with *input/output (I/O) devices* such as keyboards, monitors, and printers. A processor accesses an I/O device using the address and data busses in the same way that it accesses memory.

A portion of the address space is dedicated to I/O devices rather than memory. For example, suppose that addresses in the range

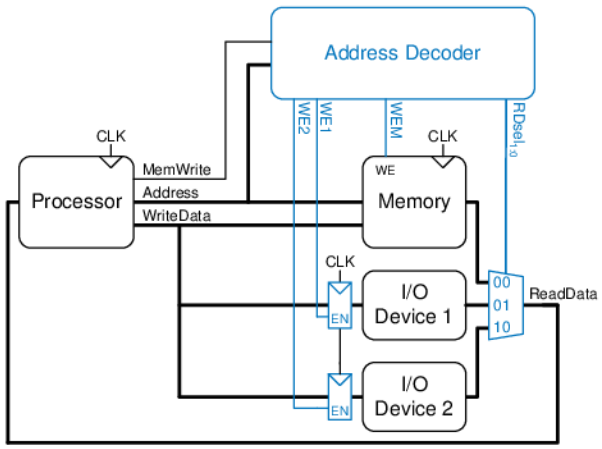


Figure 8.28 Support hardware for memory-mapped I/O

0xFFFF0000 to 0xFFFFFFFF are used for I/O. Recall from Section 6.6.1 that these addresses are in a reserved portion of the memory map. Each I/O device is assigned one or more memory addresses in this range. A store to the specified address sends data to the device. A load receives data from the device. This method of communicating with I/O devices is called *memory-mapped I/O*.

In a system with memory-mapped I/O, a load or store may access either memory or an I/O device. Figure 8.28 shows the hardware needed to support two memory-mapped I/O devices. An *address decoder* determines which device communicates with the processor. It uses the *Address* and *MemWrite* signals to generate control signals for the rest of the hardware. The *ReadData* multiplexer selects between memory and the various I/O devices. Write-enabled registers hold the values written to the I/O devices.

Some architectures, notably IA-32, use specialized instructions instead of memory-mapped I/O to communicate with I/O devices. These instructions are of the following form, where `device1` and `device2` are the unique ID of the peripheral device:

```
lwio $t0, device1
swio $t0, device2
```

This type of communication with I/O devices is called *programmed I/O*.

Example 8.17 COMMUNICATING WITH I/O DEVICES

Suppose I/O Device 1 in Figure 8.28 is assigned the memory address 0xFFFFFFF4. Show the MIPS assembly code for writing the value 7 to I/O Device 1 and for reading the output value from I/O Device 1.

Solution: The following MIPS assembly code writes the value 7 to I/O Device 1.[2]

```
addi $t0, $0, 7
sw   $t0, 0xFFF4($0)
```

The address decoder asserts *WE1* because the address is 0xFFFFFFF4 and *MemWrite* is TRUE. The value on the *WriteData* bus, 7, is written into the register connected to the input pins of I/O Device 1.

---

[2] Recall that the 16-bit immediate 0xFFF4 is sign-extended to the 32-bit value 0xFFFFFFF4.

To read from I/O Device 1, the processor performs the following MIPS assembly code.

```
lw $t1, 0xFFF4($0)
```

The address decoder sets $RDsel_{1:0}$ to 01, because it detects the address 0xFFFFFFF4 and *MemWrite* is FALSE. The output of I/O Device 1 passes through the multiplexer onto the *ReadData* bus and is loaded into `$t1` in the processor.

Software that communicates with an I/O device is called a *device driver*. You have probably downloaded or installed device drivers for your printer or other I/O device. Writing a device driver requires detailed knowledge about the I/O device hardware. Other programs call functions in the device driver to access the device without having to understand the low-level device hardware.

To illustrate memory-mapped I/O hardware and software, the rest of this section describes interfacing a commercial speech synthesizer chip to a MIPS processor.

Speech Synthesizer Hardware

See `www.speechchips.com` for more information about the SP0256 and the allophone encodings.

The Radio Shack SP0256 speech synthesizer chip generates robot-like speech. Words are composed of one or more *allophones*, the fundamental units of sound. For example, the word "hello" uses five allophones represented by the following symbols in the SP0256 speech chip: HH1 EH LL AX OW. The speech synthesizer uses 6-bit codes to represent 64 different allophones that appear in the English language. For example, the five allophones for the word "hello" correspond to the hexadecimal values 0x1B, 0x07, 0x2D, 0x0F, 0x20, respectively. The processor sends a series of allophones to the speech synthesizer, which drives a speaker to blabber the sounds.

Figure 8.29 shows the pinout of the SP0256 speech chip. The I/O pins highlighted in blue are used to interface with the MIPS processor to produce speech. Pins $A_{6:1}$ receive the 6-bit allophone encoding from the processor. The allophone sound is produced on the *Digital Out* pin. The *Digital Out* signal is first amplified and then sent to a speaker. The other two highlighted pins, *SBY* and $\overline{ALD}$, are status and control pins. When the *SBY* output is 1, the speech chip is standing by and is ready to receive a new allophone. On the falling edge of the address load input $\overline{ALD}$, the speech chip reads the allophone specified by $A_{6:1}$. Other pins, such as power and ground (V_{DD} and V_{SS}) and the clock (*OSC1*), must be connected as shown but are not driven by the processor.

Figure 8.30 shows the speech synthesizer interfaced to the MIPS processor. The processor uses three memory-mapped I/O addresses to communicate with the speech synthesizer. We arbitrarily have chosen

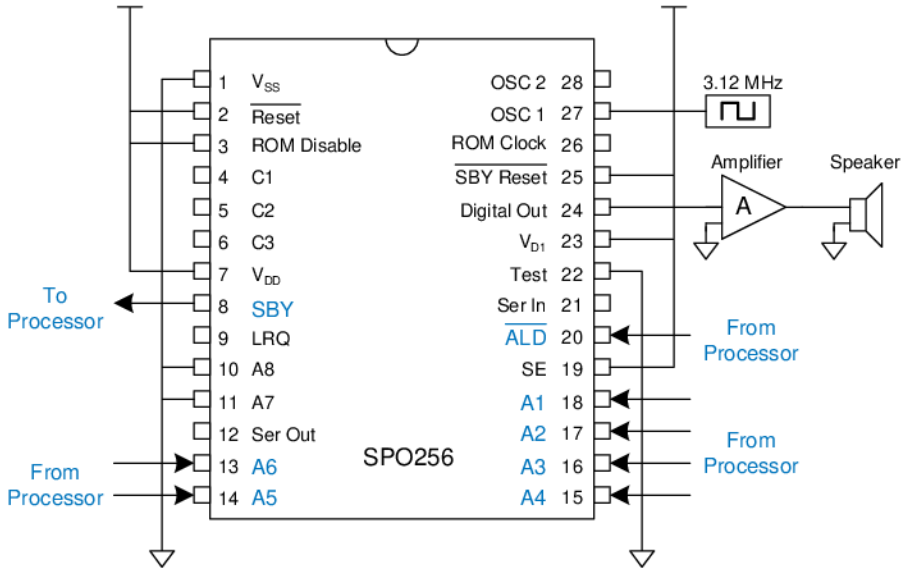


Figure 8.29 SP0256 speech synthesizer chip pinout

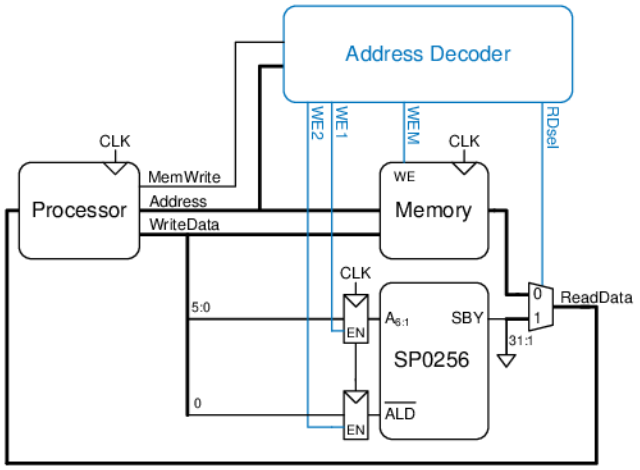


Figure 8.30 Hardware for driving the SP0256 speech synthesizer

that the $A_{6:1}$ port is mapped to address 0xFFFFFF00, $\overline{ALD}$ to 0xFFFFFF04, and *SBY* to 0xFFFFFF08. Although the *WriteData* bus is 32 bits, only the least significant 6 bits are used for $A_{6:1}$, and the least significant bit is used for $\overline{ALD}$; the other bits are ignored. Similarly, *SBY* is read on the least significant bit of the *ReadData* bus; the other bits are 0.

Speech Synthesizer Device Driver

The device driver controls the speech synthesizer by sending an appropriate series of allophones over the memory-mapped I/O interface. It follows the protocol expected by the SPO256 chip, given below:

- Set $\overline{ALD}$ to 1
- Wait until the chip asserts *SBY* to indicate that it is finished speaking the previous allophone and is ready for the next
- Write a 6-bit allophone to $A_{6:1}$
- Reset $\overline{ALD}$ to 0 to initiate speech

This sequence can be repeated for any number of allophones. The MIPS assembly in Code Example 8.1 writes five allophones to the speech chip. The allophone encodings are stored as 32-bit values in a five-entry array starting at memory address 0x10000000.

Code Example 8.1 SPEECH CHIP DEVICE DRIVER

```
init:
      addi $t1, $0, 1      # $t1 = 1 (value to write to ALD)
      addi $t2, $0, 20     # $t2 = array size * 4
      lui  $t3, 0x1000     # $t3 = array base address
      addi $t4, $0, 0      # $t4 = 0 (array index)

start:
      sw   $t1, 0xFF04($0) # ALD =1
loop:
      lw   $t5, 0xFF08($0) # $t5 = SBY
      beq  $0,  $t5, loop  # loop until SBY == 1

      add  $t5, $t3, $t4   # $t5 = address of allophone
      lw   $t5, 0($t5)     # $t5 = allophone
      sw   $t5, 0xFF00($0) # A6:1 = allophone
      sw   $0,  0xFF04($0) # ALD = 0 to initiate speech
      addi $t4, $t4, 4     # increment array index
      beq  $t4, $t2, done  # last allophone in array?
      j    start           # repeat

done:
```

The assembly code in Code Example 8.1 *polls*, or repeatedly checks, the *SBY* signal to determine when the speech chip is ready to receive a new allophone. The code functions correctly but wastes valuable processor cycles that could be used to perform useful work. Instead of polling, the processor could use an *interrupt* connected to *SBY*. When *SBY* rises, the processor stops what it is doing and jumps to code that handles the interrupt. In the case of the speech synthesizer, the interrupt handler

would send the next allophone, then let the processor resume what it was doing before the interrupt. As described in Section 6.7.2, the processor handles interrupts like any other exception.

8.6 REAL-WORLD PERSPECTIVE: IA-32 MEMORY AND I/O SYSTEMS*

As processors get faster, they need ever more elaborate memory hierarchies to keep a steady supply of data and instructions flowing. This section describes the memory systems of IA-32 processors to illustrate the progression. Section 7.9 contained photographs of the processors, highlighting the on-chip caches. IA-32 also has an unusual programmed I/O system that differs from the more common memory-mapped I/O.

8.6.1 IA-32 Cache Systems

The 80386, initially produced in 1985, operated at 16 MHz. It lacked a cache, so it directly accessed main memory for all instructions and data. Depending on the speed of the memory, the processor might get an immediate response, or it might have to pause for one or more cycles for the memory to react. These cycles are called *wait states*, and they increase the CPI of the processor. Microprocessor clock frequencies have increased by at least 25% per year since then, whereas memory latency has scarcely diminished. The delay from when the processor sends an address to main memory until the memory returns the data can now exceed 100 processor clock cycles. Therefore, caches with a low miss rate are essential to good performance. Table 8.5 summarizes the evolution of cache systems on Intel IA-32 processors.

The 80486 introduced a unified write-through cache to hold both instructions and data. Most high-performance computer systems also provided a larger second-level cache on the motherboard using commercially available SRAM chips that were substantially faster than main memory.

The Pentium processor introduced separate instruction and data caches to avoid contention during simultaneous requests for data and instructions. The caches used a write-back policy, reducing the communication with main memory. Again, a larger second-level cache (typically 256–512 KB) was usually offered on the motherboard.

The P6 series of processors (Pentium Pro, Pentium II, and Pentium III) were designed for much higher clock frequencies. The second-level cache on the motherboard could not keep up, so it was moved closer to the processor to improve its latency and throughput. The Pentium Pro was packaged in a *multichip module* (*MCM*) containing both the processor chip and a second-level cache chip, as shown in Figure 8.31. Like the Pentium, the processor had separate 8-KB level 1 instruction and data

Table 8.5 Evolution of Intel IA-32 microprocessor memory systems

Processor	Year	Frequency (MHz)	Level 1 Data Cache	Level 1 Instruction Cache	Level 2 Cache
80386	1985	16–25	none	none	none
80486	1989	25–100	8 KB unified		none on chip
Pentium	1993	60–300	8 KB	8 KB	none on chip
Pentium Pro	1995	150–200	8 KB	8 KB	256 KB–1 MB on MCM
Pentium II	1997	233–450	16 KB	16 KB	256–512 KB on cartridge
Pentium III	1999	450–1400	16 KB	16 KB	256–512 KB on chip
Pentium 4	2001	1400–3730	8–16 KB	12 K op trace cache	256 KB–2 MB on chip
Pentium M	2003	900–2130	32 KB	32 KB	1–2 MB on chip
Core Duo	2005	1500–2160	32 KB/core	32 KB/core	2 MB shared on chip

caches. However, these caches were *nonblocking*, so that the out-of-order processor could continue executing subsequent cache accesses even if the cache missed a particular access and had to fetch data from main memory. The second-level cache was 256 KB, 512 KB, or 1 MB in size and could operate at the same speed as the processor. Unfortunately, the MCM packaging proved too expensive for high-volume manufacturing. Therefore, the Pentium II was sold in a lower-cost cartridge containing the processor and the second-level cache. The level 1 caches were doubled in size to compensate for the fact that the second-level cache operated at half the processor's speed. The Pentium III integrated a full-speed second-level cache directly onto the same chip as the processor. A cache on the same chip can operate at better latency and throughput, so it is substantially more effective than an off-chip cache of the same size.

The Pentium 4 offered a nonblocking level 1 data cache. It switched to a *trace cache* to store instructions after they had been decoded into micro-ops, avoiding the delay of redecoding each time instructions were fetched from the cache.

The Pentium M design was adapted from the Pentium III. It further increased the level 1 caches to 32 KB each and featured a 1- to 2-MB level 2 cache. The Core Duo contains two modified Pentium M processors and

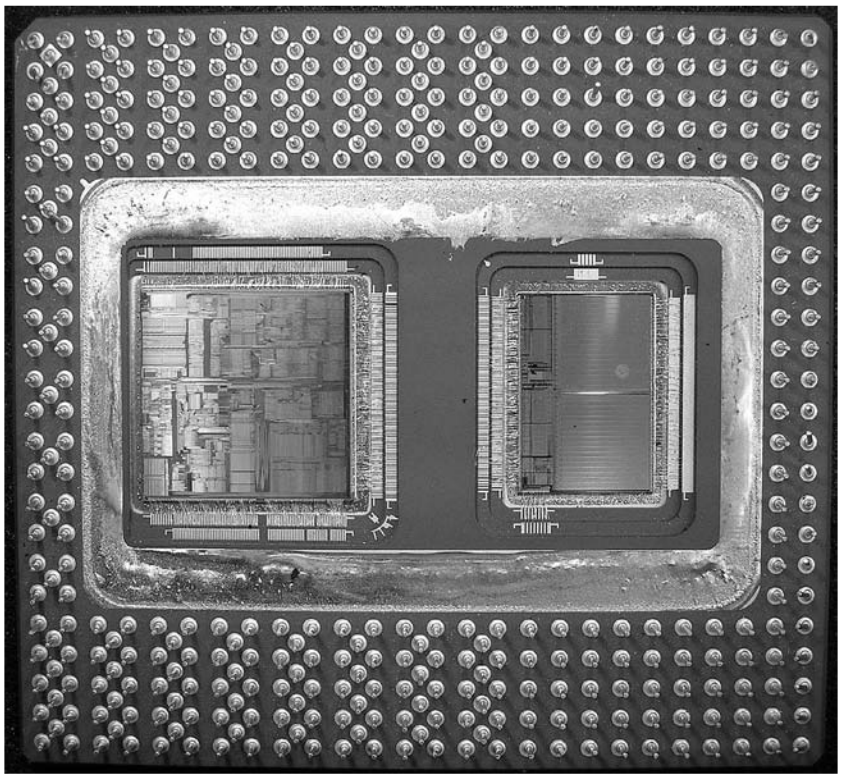


Figure 8.31 Pentium Pro multichip module with processor (left) and 256-KB cache (right) in a pin grid array (PGA) package (Courtesy Intel.)

a shared 2-MB cache on one chip. The shared cache is used for communication between the processors: one can write data to the cache, and the other can read it.

8.6.2 IA-32 Virtual Memory

IA-32 processors operate in either real mode or protected mode. *Real mode* is backward compatible with the original 8086. It only uses 20 bits of addresses, limiting memory to 1 MB, and it does not allow virtual memory.

Protected mode was introduced with the 80286 and extended to 32-bit addresses with the 80386. It supports virtual memory with 4-KB pages. It also provides memory protection so that one program cannot access the pages belonging to other programs. Hence, a buggy or malicious program cannot crash or corrupt other programs. All modern operating systems now use protected mode.

A 32-bit address permits up to 4 GB of memory. Processors since the Pentium Pro have bumped the memory capacity to 64 GB using a

Although memory protection became available in the hardware in the early 1980s, Microsoft Windows took almost 15 years to take advantage of the feature and prevent bad programs from crashing the entire computer. Until the release of Windows 2000, consumer versions of Windows were notoriously unstable. The lag between hardware features and software support can be extremely long.

technique called *physical address extension*. Each process uses 32-bit addresses. The virtual memory system maps these addresses onto a larger 36-bit virtual memory space. It uses different page tables for each process, so that each process can have its own address space of up to 4 GB.

8.6.3 IA-32 Programmed I/O

Most architectures use memory-mapped I/O, described in Section 8.5, in which programs access I/O devices by reading and writing memory locations. IA-32 uses *programmed I/O*, in which special IN and OUT instructions are used to read and write I/O devices. IA-32 defines 2^{16} I/O ports. The IN instruction reads one, two, or four bytes from the port specified by DX into AL, AX, or EAX. OUT is similar, but writes the port.

Connecting a peripheral device to a programmed I/O system is similar to connecting it to a memory-mapped system. When accessing an I/O port, the processor sends the port number rather than the memory address on the 16 least significant bits of the address bus. The device reads or writes data from the data bus. The major difference is that the processor also produces an $M/\overline{IO}$ signal. When $M/\overline{IO} = 1$, the processor is accessing memory. When it is 0, the process is accessing one of the I/O devices. The address decoder must also look at $M/\overline{IO}$ to generate the appropriate enables for main memory and for the I/O devices. I/O devices can also send interrupts to the processor to indicate that they are ready to communicate.

8.7 SUMMARY

Memory system organization is a major factor in determining computer performance. Different memory technologies, such as DRAM, SRAM, and hard disks, offer trade-offs in capacity, speed, and cost. This chapter introduced cache and virtual memory organizations that use a hierarchy of memories to approximate an ideal large, fast, inexpensive memory. Main memory is typically built from DRAM, which is significantly slower than the processor. A cache reduces access time by keeping commonly used data in fast SRAM. Virtual memory increases the memory capacity by using a hard disk to store data that does not fit in the main memory. Caches and virtual memory add complexity and hardware to a computer system, but the benefits usually outweigh the costs. All modern personal computers use caches and virtual memory. Most processors also use the memory interface to communicate with I/O devices. This is called memory-mapped I/O. Programs use load and store operations to access the I/O devices.

EPILOGUE

This chapter brings us to the end of our journey together into the realm of digital systems. We hope this book has conveyed the beauty and thrill of the art as well as the engineering knowledge. You have learned to design combinational and sequential logic using schematics and hardware description languages. You are familiar with larger building blocks such as multiplexers, ALUs, and memories. Computers are one of the most fascinating applications of digital systems. You have learned how to program a MIPS processor in its native assembly language and how to build the processor and memory system using digital building blocks. Throughout, you have seen the application of abstraction, discipline, hierarchy, modularity, and regularity. With these techniques, we have pieced together the puzzle of a microprocessor's inner workings. From cell phones to digital television to Mars rovers to medical imaging systems, our world is an increasingly digital place.

Imagine what Faustian bargain Charles Babbage would have made to take a similar journey a century and a half ago. He merely aspired to calculate mathematical tables with mechanical precision. Today's digital systems are yesterday's science fiction. Might Dick Tracy have listened to iTunes on his cell phone? Would Jules Verne have launched a constellation of global positioning satellites into space? Could Hippocrates have cured illness using high-resolution digital images of the brain? But at the same time, George Orwell's nightmare of ubiquitous government surveillance becomes closer to reality each day. And rogue states develop nuclear weapons using laptop computers more powerful than the room-sized supercomputers that simulated Cold War bombs. The microprocessor revolution continues to accelerate. The changes in the coming decades will surpass those of the past. You now have the tools to design and build these new systems that will shape our future. With your newfound power comes profound responsibility. We hope that you will use it, not just for fun and riches, but also for the benefit of humanity.

Exercises

Exercise 8.1 In less than one page, describe four everyday activities that exhibit temporal or spatial locality. List two activities for each type of locality, and be specific.

Exercise 8.2 In one paragraph, describe two short computer applications that exhibit temporal and/or spatial locality. Describe how. Be specific.

Exercise 8.3 Come up with a sequence of addresses for which a direct mapped cache with a size (capacity) of 16 words and block size of 4 words outperforms a fully associative cache with least recently used (LRU) replacement that has the same capacity and block size.

Exercise 8.4 Repeat Exercise 8.3 for the case when the fully associative cache outperforms the direct mapped cache.

Exercise 8.5 Describe the trade-offs of increasing each of the following cache parameters while keeping the others the same:

(a) block size

(b) associativity

(c) cache size

Exercise 8.6 Is the miss rate of a two-way set associative cache always, usually, occasionally, or never better than that of a direct mapped cache of the same capacity and block size? Explain.

Exercise 8.7 Each of the following statements pertains to the miss rate of caches. Mark each statement as true or false. Briefly explain your reasoning; present a counterexample if the statement is false.

(a) A two-way set associative cache always has a lower miss rate than a direct mapped cache with the same block size and total capacity.

(b) A 16-KB direct mapped cache always has a lower miss rate than an 8-KB direct mapped cache with the same block size.

(c) An instruction cache with a 32-byte block size usually has a lower miss rate than an instruction cache with an 8-byte block size, given the same degree of associativity and total capacity.

Exercise 8.8 A cache has the following parameters: b, block size given in numbers of words; S, number of sets; N, number of ways; and A, number of address bits.

(a) In terms of the parameters described, what is the cache capacity, C?

(b) In terms of the parameters described, what is the total number of bits required to store the tags?

(c) What are S and N for a fully associative cache of capacity C words with block size b?

(d) What is S for a direct mapped cache of size C words and block size b?

Exercise 8.9 A 16-word cache has the parameters given in Exercise 8.8. Consider the following repeating sequence of `lw` addresses (given in hexadecimal):

40 44 48 4C 70 74 78 7C 80 84 88 8C 90 94 98 9C 0 4 8 C 10 14 18 1C 20

Assuming least recently used (LRU) replacement for associative caches, determine the effective miss rate if the sequence is input to the following caches, ignoring startup effects (i.e., compulsory misses).

(a) direct mapped cache, $S = 16$, $b = 1$ word

(b) fully associative cache, $N = 16$, $b = 1$ word

(c) two-way set associative cache, $S = 8$, $b = 1$ word

(d) direct mapped cache, $S = 8$, $b = 2$ words

Exercise 8.10 Suppose you are running a program with the following data access pattern. The pattern is executed only once.

0x0, 0x8, 0x10, 0x18, 0x20, 0x28

(a) If you use a direct mapped cache with a cache size of 1 KB and a block size of 8 bytes (2 words), how many sets are in the cache?

(b) With the same cache and block size as in part (a), what is the miss rate of the direct mapped cache for the given memory access pattern?

(c) For the given memory access pattern, which of the following would decrease the miss rate the most? (Cache capacity is kept constant.) Circle one.

  (i) Increasing the degree of associativity to 2.

  (ii) Increasing the block size to 16 bytes.

(iii) Either (i) or (ii).

(iv) Neither (i) nor (ii).

Exercise 8.11 You are building an instruction cache for a MIPS processor. It has a total capacity of $4C = 2^{c+2}$ bytes. It is $N = 2^n$-way set associative ($N \geq 8$), with a block size of $b = 2^{b'}$ bytes ($b \geq 8$). Give your answers to the following questions in terms of these parameters.

(a) Which bits of the address are used to select a word within a block?

(b) Which bits of the address are used to select the set within the cache?

(c) How many bits are in each tag?

(d) How many tag bits are in the entire cache?

Exercise 8.12 Consider a cache with the following parameters: N (associativity) = 2, b (block size) = 2 words, W (word size) = 32 bits, C (cache size) = 32 K words, A (address size) = 32 bits. You need consider only word addresses.

(a) Show the tag, set, block offset, and byte offset bits of the address. State how many bits are needed for each field.

(b) What is the size of *all* the cache tags in bits?

(c) Suppose each cache block also has a valid bit (V) and a dirty bit (D). What is the size of each cache set, including data, tag, and status bits?

(d) Design the cache using the building blocks in Figure 8.32 and a small number of two-input logic gates. The cache design must include tag storage, data storage, address comparison, data output selection, and any other parts you feel are relevant. Note that the multiplexer and comparator blocks may be any size (n or p bits wide, respectively), but the SRAM blocks must be 16 K × 4 bits. Be sure to include a neatly labeled block diagram.

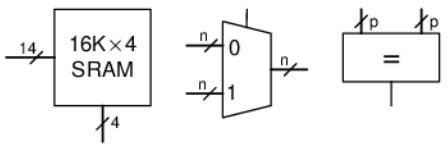


Figure 8.32 Building blocks

Exercise 8.13 You've joined a hot new Internet startup to build wrist watches with a built-in pager and Web browser. It uses an embedded processor with a multilevel cache scheme depicted in Figure 8.33. The processor includes a small on-chip cache in addition to a large off-chip second-level cache. (Yes, the watch weighs 3 pounds, but you should see it surf!)

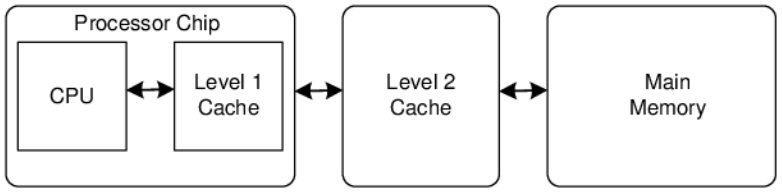


Figure 8.33 Computer system

Assume that the processor uses 32-bit physical addresses but accesses data only on word boundaries. The caches have the characteristics given in Table 8.6. The DRAM has an access time of t_m and a size of 512 MB.

Table 8.6 Memory characteristics

Characteristic	On-chip Cache	Off-chip Cache
organization	four-way set associative	direct mapped
hit rate	A	B
access time	t_a	t_b
block size	16 bytes	16 bytes
number of blocks	512	256K

(a) For a given word in memory, what is the total number of locations in which it might be found in the on-chip cache and in the second-level cache?

(b) What is the size, in bits, of each tag for the on-chip cache and the second-level cache?

(c) Give an expression for the average memory read access time. The caches are accessed in sequence.

(d) Measurements show that, for a particular problem of interest, the on-chip cache hit rate is 85% and the second-level cache hit rate is 90%. However, when the on-chip cache is disabled, the second-level cache hit rate shoots up to 98.5%. Give a brief explanation of this behavior.

Exercise 8.14 This chapter described the least recently used (LRU) replacement policy for multiway associative caches. Other, less common, replacement policies include first-in-first-out (FIFO) and random policies. FIFO replacement evicts the block that has been there the longest, regardless of how recently it was accessed. Random replacement randomly picks a block to evict.

(a) Discuss the advantages and disadvantages of each of these replacement policies.

(b) Describe a data access pattern for which FIFO would perform better than LRU.

Exercise 8.15 You are building a computer with a hierarchical memory system that consists of separate instruction and data caches followed by main memory. You are using the MIPS multicycle processor from Figure 7.41 running at 1 GHz.

(a) Suppose the instruction cache is perfect (i.e., always hits) but the data cache has a 5% miss rate. On a cache miss, the processor stalls for 60 ns to access main memory, then resumes normal operation. Taking cache misses into account, what is the average memory access time?

(b) How many clock cycles per instruction (CPI) on average are required for load and store word instructions considering the non-ideal memory system?

(c) Consider the benchmark application of Example 7.7 that has 25% loads, 10% stores, 11% branches, 2% jumps, and 52% R-type instructions.[3] Taking the non-ideal memory system into account, what is the average CPI for this benchmark?

(d) Now suppose that the instruction cache is also non-ideal and has a 7% miss rate. What is the average CPI for the benchmark in part (c)? Take into account both instruction and data cache misses.

Exercise 8.16 If a computer uses 64-bit virtual addresses, how much virtual memory can it access? Note that 2^{40} bytes = 1 *terabyte*, 2^{50} bytes = 1 *petabyte*, and 2^{60} bytes = 1 *exabyte*.

Exercise 8.17 A supercomputer designer chooses to spend $1 million on DRAM and the same amount on hard disks for virtual memory. Using the prices from Figure 8.4, how much physical and virtual memory will the computer have? How many bits of physical and virtual addresses are necessary to access this memory?

---

[3] Data from Patterson and Hennessy, *Computer Organization and Design*, 3rd Edition, Morgan Kaufmann, 2005. Used with permission.

Exercise 8.18 Consider a virtual memory system that can address a total of 2^{32} bytes. You have unlimited hard disk space, but are limited to only 8 MB of semiconductor (physical) memory. Assume that virtual and physical pages are each 4 KB in size.

(a) How many bits is the physical address?

(b) What is the maximum number of virtual pages in the system?

(c) How many physical pages are in the system?

(d) How many bits are the virtual and physical page numbers?

(e) Suppose that you come up with a direct mapped scheme that maps virtual pages to physical pages. The mapping uses the least significant bits of the virtual page number to determine the physical page number. How many virtual pages are mapped to each physical page? Why is this "direct mapping" a bad plan?

(f) Clearly, a more flexible and dynamic scheme for translating virtual addresses into physical addresses is required than the one described in part (d). Suppose you use a page table to store mappings (translations from virtual page number to physical page number). How many page table entries will the page table contain?

(g) Assume that, in addition to the physical page number, each page table entry also contains some status information in the form of a valid bit (V) and a dirty bit (D). How many bytes long is each page table entry? (Round up to an integer number of bytes.)

(h) Sketch the layout of the page table. What is the total size of the page table in bytes?

Exercise 8.19 You decide to speed up the virtual memory system of Exercise 8.18 by using a translation lookaside buffer (TLB). Suppose your memory system has the characteristics shown in Table 8.7. The TLB and cache miss rates indicate how often the requested entry is not found. The main memory miss rate indicates how often page faults occur.

Table 8.7 Memory characteristics

Memory Unit	Access Time (Cycles)	Miss Rate
TLB	1	0.05%
cache	1	2%
main memory	100	0.0003%
disk	1,000,000	0%

(a) What is the average memory access time of the virtual memory system before and after adding the TLB? Assume that the page table is always resident in physical memory and is never held in the data cache.

(b) If the TLB has 64 entries, how big (in bits) is the TLB? Give numbers for data (physical page number), tag (virtual page number), and valid bits of each entry. Show your work clearly.

(c) Sketch the TLB. Clearly label all fields and dimensions.

(d) What size SRAM would you need to build the TLB described in part (c)? Give your answer in terms of depth × width.

Exercise 8.20 Suppose the MIPS multicycle processor described in Section 7.4 uses a virtual memory system.

(a) Sketch the location of the TLB in the multicycle processor schematic.

(b) Describe how adding a TLB affects processor performance.

Exercise 8.21 The virtual memory system you are designing uses a single-level page table built from dedicated hardware (SRAM and associated logic). It supports 25-bit virtual addresses, 22-bit physical addresses, and 2^{16}-byte (64 KB) pages. Each page table entry contains a physical page number, a valid bit (V) and a dirty bit (D).

(a) What is the total size of the page table, in bits?

(b) The operating system team proposes reducing the page size from 64 to 16 KB, but the hardware engineers on your team object on the grounds of added hardware cost. Explain their objection.

(c) The page table is to be integrated on the processor chip, along with the on-chip cache. The on-chip cache deals only with physical (not virtual) addresses. Is it possible to access the appropriate set of the on-chip cache concurrently with the page table access for a given memory access? Explain briefly the relationship that is necessary for concurrent access to the cache set and page table entry.

(d) Is it possible to perform the tag comparison in the on-chip cache concurrently with the page table access for a given memory access? Explain briefly.

Exercise 8.22 Describe a scenario in which the virtual memory system might affect how an application is written. Be sure to include a discussion of how the page size and physical memory size affect the performance of the application.

Exercise 8.23 Suppose you own a personal computer (PC) that uses 32-bit virtual addresses.

(a) What is the maximum amount of virtual memory space each program can use?

(b) How does the size of your PC's hard disk affect performance?

(c) How does the size of your PC's physical memory affect performance?

Exercise 8.24 Use MIPS memory-mapped I/O to interact with a user. Each time the user presses a button, a pattern of your choice displays on five light-emitting diodes (LEDs). Suppose the input button is mapped to address 0xFFFFFF10 and the LEDs are mapped to address 0xFFFFFF14. When the button is pushed, its output is 1; otherwise it is 0.

(a) Write MIPS code to implement this functionality.

(b) Draw a schematic similar to Figure 8.30 for this memory-mapped I/O system.

(c) Write HDL code to implement the address decoder for your memory-mapped I/O system.

Exercise 8.25 Finite state machines (FSMs), like the ones you built in Chapter 3, can also be implemented in software.

(a) Implement the traffic light FSM from Figure 3.25 using MIPS assembly code. The inputs (T_A and T_B) are memory-mapped to bit 1 and bit 0, respectively, of address 0xFFFFF000. The two 3-bit outputs (L_A and L_B) are mapped to bits 0–2 and bits 3–5, respectively, of address 0xFFFFF004. Assume one-hot output encodings for each light, L_A and L_B; red is 100, yellow is 010, and green is 001.

(b) Draw a schematic similar to Figure 8.30 for this memory-mapped I/O system.

(c) Write HDL code to implement the address decoder for your memory-mapped I/O system.

Interview Questions

The following exercises present questions that have been asked on interviews.

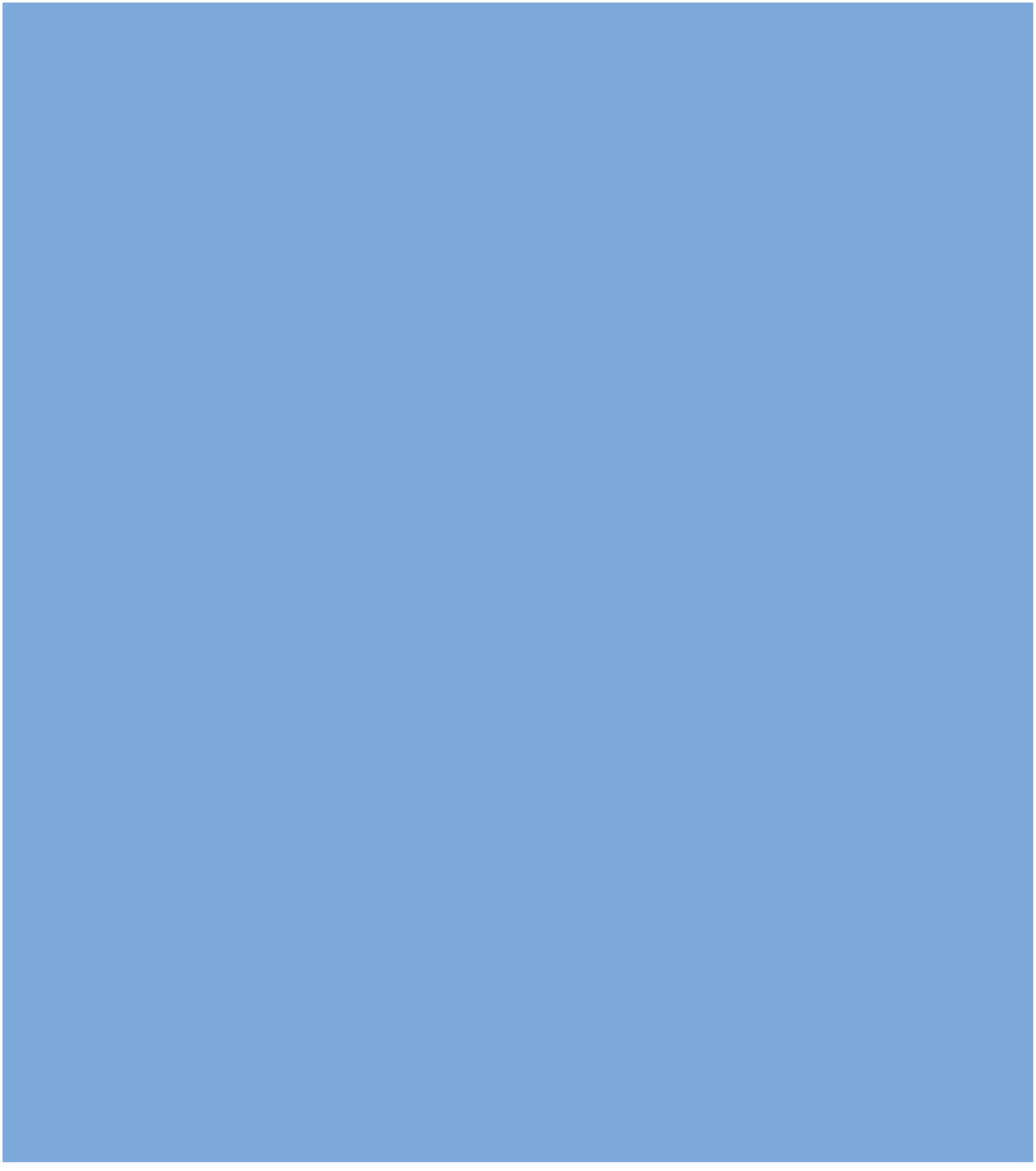
Question 8.1 Explain the difference between direct mapped, set associative, and fully associative caches. For each cache type, describe an application for which that cache type will perform better than the other two.

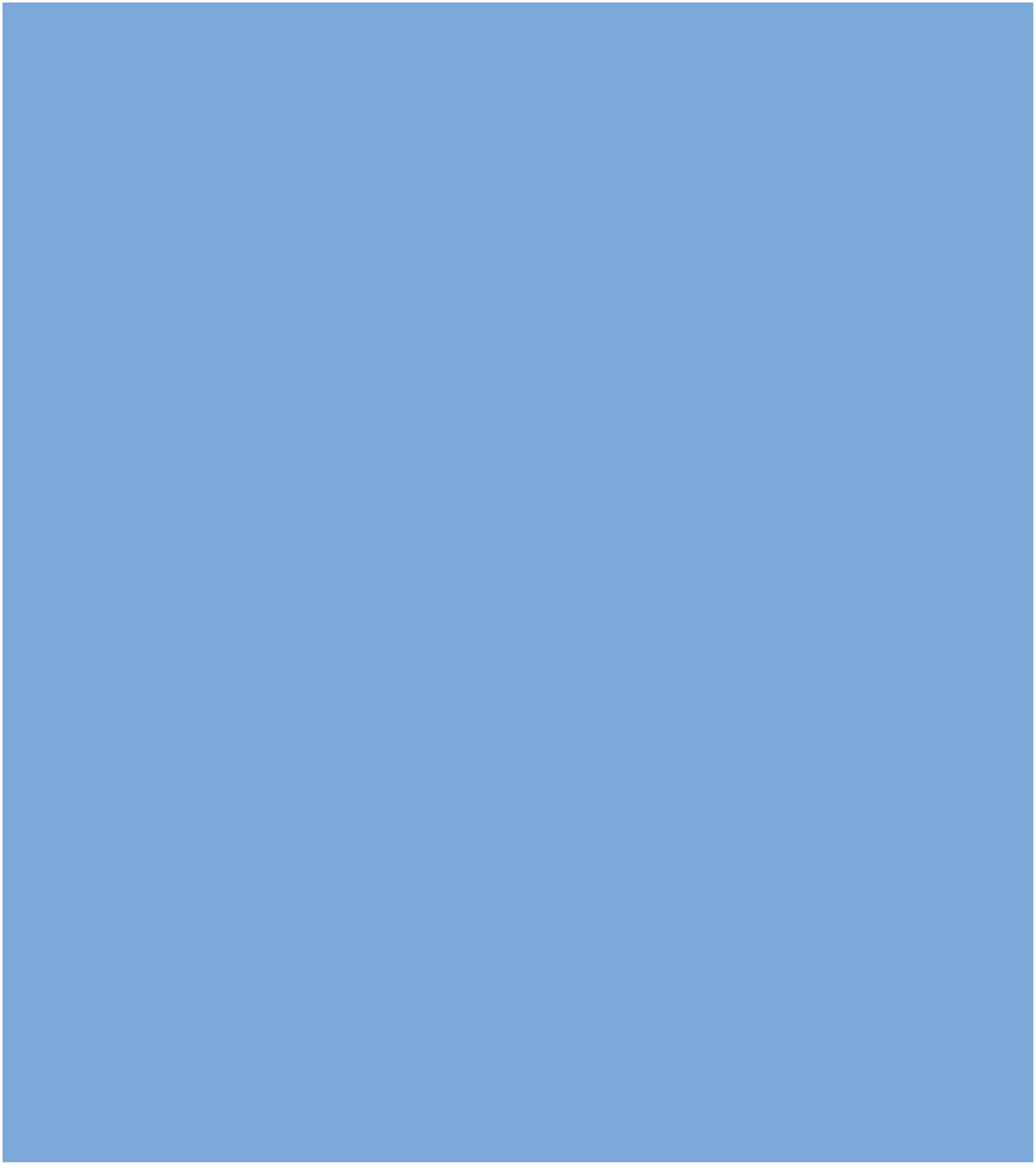
Question 8.2 Explain how virtual memory systems work.

Question 8.3 Explain the advantages and disadvantages of using a virtual memory system.

Question 8.4 Explain how cache performance might be affected by the virtual page size of a memory system.

Question 8.5 Can addresses used for memory-mapped I/O be cached? Explain why or why not.





Digital System Implementation

A

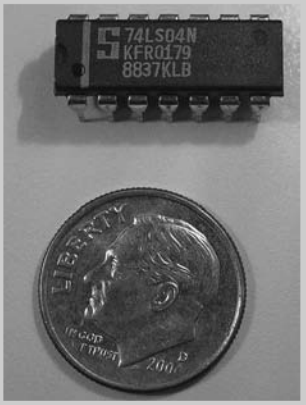
A.1 INTRODUCTION

This appendix introduces practical issues in the design of digital systems. The material in this appendix is not necessary for understanding the rest of the book. However, it seeks to demystify the process of building real digital systems. Moreover, we believe that the best way to understand digital systems is to build and debug them yourself in the laboratory.

Digital systems are usually built using one or more chips. One strategy is to connect together chips containing individual logic gates or larger elements such as arithmetic/logical units (ALUs) or memories. Another is to use programmable logic, which contains generic arrays of circuitry that can be programmed to perform specific logic functions. Yet a third is to design a custom integrated circuit containing the specific logic necessary for the system. These three strategies offer trade-offs in cost, speed, power consumption, and design time that are explored in the following sections. This appendix also examines the physical packaging and assembly of circuits, the transmission lines that connect the chips, and the economics of digital systems.

A.2 74XX LOGIC

In the 1970s and 1980s, many digital systems were built from simple chips, each containing a handful of logic gates. For example, the 7404 chip contains six NOT gates, the 7408 contains four AND gates, and the 7474 contains two flip-flops. These chips are collectively referred to as *74xx-series* logic. They were sold by many manufacturers, typically for 10 to 25 cents per chip. These chips are now largely obsolete, but they are still handy for simple digital systems or class projects, because they are so inexpensive and easy to use. 74xx-series chips are commonly sold in 14-pin *dual inline packages* (*DIPs*).



74LS04 inverter chip in a 14-pin dual inline package. The part number is on the first line. LS indicates the logic family (see Section A.6). The N suffix indicates a DIP package. The large S is the logo of the manufacturer, Signetics. The bottom two lines of gibberish are codes indicating the batch in which the chip was manufactured.

A.2.1 Logic Gates

Figure A.1 shows the pinout diagrams for a variety of popular 74xx-series chips containing basic logic gates. These are sometimes called *small-scale integration* (*SSI*) chips, because they are built from a few transistors. The 14-pin packages typically have a notch at the top or a dot on the top left to indicate orientation. Pins are numbered starting with 1 in the upper left and going counterclockwise around the package. The chips need to receive power (V_{DD} = 5 V) and ground (GND = 0 V) at pins 14 and 7, respectively. The number of logic gates on the chip is determined by the number of pins. Note that pins 3 and 11 of the 7421 chip are not connected (NC) to anything. The 7474 flip-flop has the usual *D*, *CLK*, and *Q* terminals. It also has a complementary output, $\overline{Q}$. Moreover, it receives asynchronous set (also called preset, or *PRE*) and reset (also called clear, or *CLR*) signals. These are active low; in other words, the flop sets when $\overline{PRE} = 0$, resets when $\overline{CLR} = 0$, and operates normally when $\overline{PRE} = \overline{CLR} = 1$.

A.2.2 Other Functions

The 74xx series also includes somewhat more complex logic functions, including those shown in Figures A.2 and A.3. These are called *medium-scale integration* (*MSI*) chips. Most use larger packages to accommodate more inputs and outputs. Power and ground are still provided at the upper right and lower left, respectively, of each chip. A general functional description is provided for each chip. See the manufacturer's data sheets for complete descriptions.

A.3 PROGRAMMABLE LOGIC

Programmable logic consists of arrays of circuitry that can be configured to perform specific logic functions. We have already introduced three forms of programmable logic: programmable read only memories (PROMs), programmable logic arrays (PLAs), and field programmable gate arrays (FPGAs). This section shows chip implementations for each of these. Configuration of these chips may be performed by blowing on-chip fuses to connect or disconnect circuit elements. This is called *one-time programmable* (*OTP*) logic because, once a fuse is blown, it cannot be restored. Alternatively, the configuration may be stored in a memory that can be reprogrammed at will. Reprogrammable logic is convenient in the laboratory, because the same chip can be reused during development.

A.3.1 PROMs

As discussed in Section 5.5.7, PROMs can be used as lookup tables. A 2^N-word × M-bit PROM can be programmed to perform any combinational function of N inputs and M outputs. Design changes simply

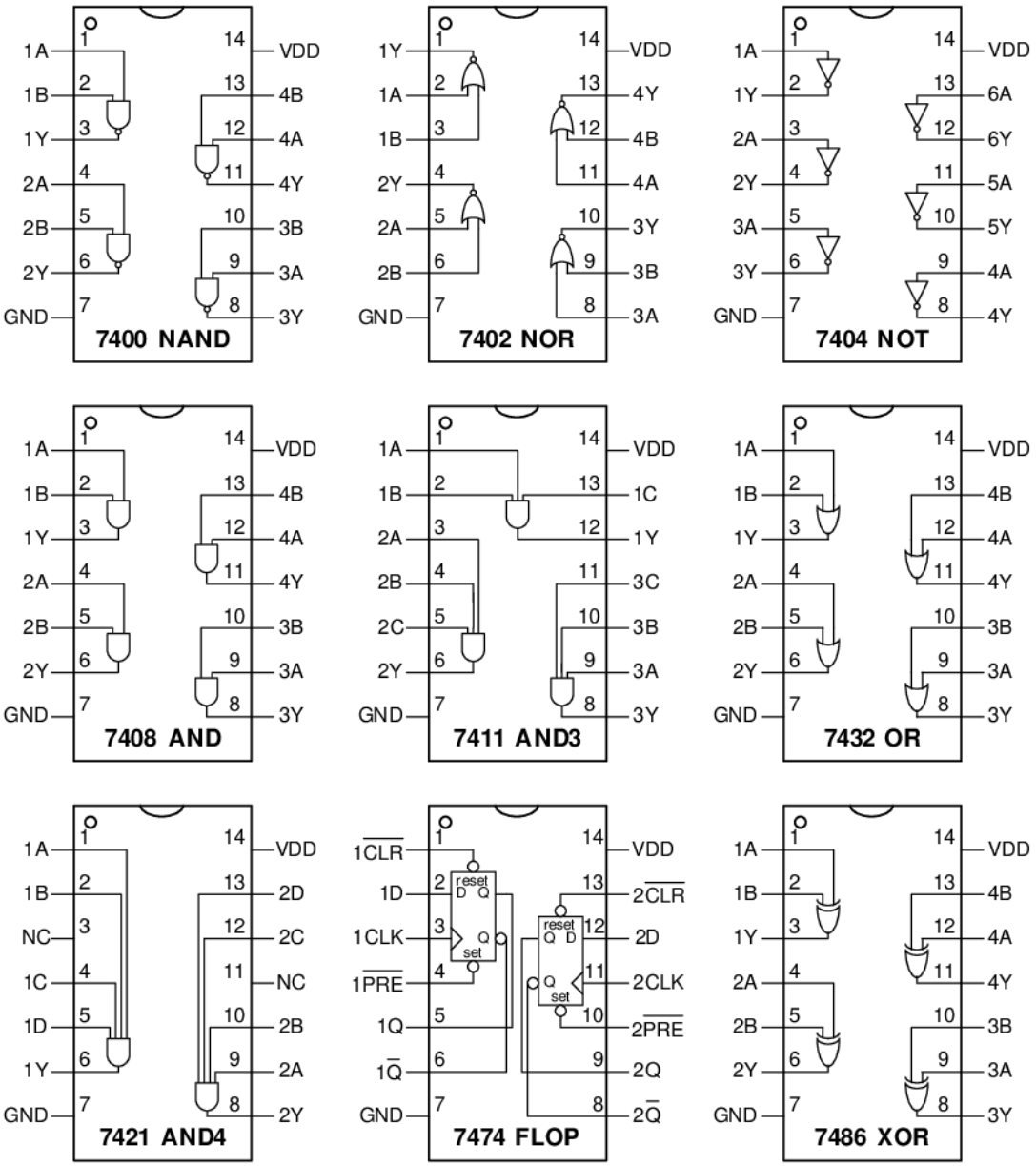
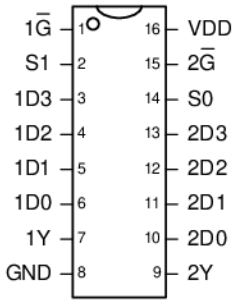


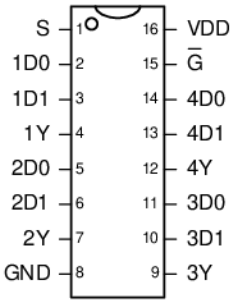
Figure A.1 Common 74xx-series logic gates



74153 4 :1 Mux

Two 4:1 Multiplexers
$D_{3:0}$: data
$S_{1:0}$: select
Y: output
Gb: enable

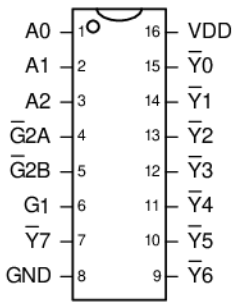
```
always @(1Gb, S, 1D)
  if (1Gb) 1Y = 0;
  else     1Y = 1D[S];
always @(2Gb, S, 2D)
  if (2Gb) 2Y = 0;
  else     2Y = 2D[S];
```



74157 2 :1 Mux

Four 2:1 Multiplexers
$D_{1:0}$: data
S: select
Y: output
Gb: enable

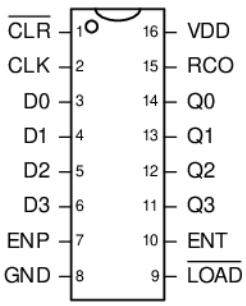
```
always @(*)
  if (~Gb) 1Y = 0;
  else     1Y = S ? 1D[1] : 1D[0];
  if (~Gb) 2Y = 0;
  else     2Y = S ? 2D[1] : 2D[0];
  if (~Gb) 3Y = 0;
  else     3Y = S ? 3D[1] : 3D[0];
  if (~Gb) 4Y = 0;
  else     4Y = S ? 4D[1] : 4D[0];
```



74138 3 :8 Decoder

3:8 Decoder
$A_{2:0}$: address
$Yb_{7:0}$: output
G1: active high enable
G2: active low enables

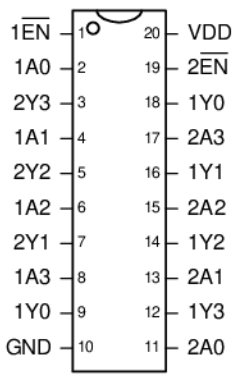
G1	G2A	G2B	A2:0	Y7:0
0	x	x	xxx	11111111
1	1	x	xxx	11111111
1	0	1	xxx	11111111
1	0	0	000	11111110
1	0	0	001	11111101
1	0	0	010	11111011
1	0	0	011	11110111
1	0	0	100	11101111
1	0	0	101	11011111
1	0	0	110	10111111
1	0	0	111	01111111



74161/163 Counter

4-bit Counter
CLK: clock
$Q_{3:0}$: counter output
$D_{3:0}$: parallel input
CLRb: async reset (161)
sync reset (163)
LOADb: load Q from D
ENP, ENT: enables
RCO: ripple carry out

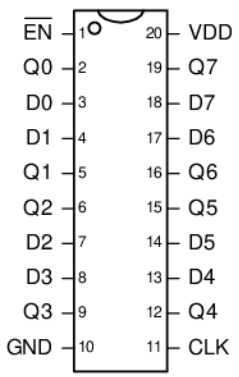
```
always @(posedge CLK) // 74163
  if (~CLRb) Q <= 4'b0000;
  else if (~LOADb) Q <= D;
  else if (ENP & ENT) Q <= Q+1;

assign RCO = (Q == 4'b1111) & ENT;
```



74244 Tristate Buffer

8-bit Tristate Buffer
$A_{3:0}$: input
$Y_{3:0}$: output
ENb: enable

```
assign 1Y =
  1ENb ? 4'bzzzz : 1A;
assign 2Y =
  2ENB ? 4'bzzzz : 2A;
```



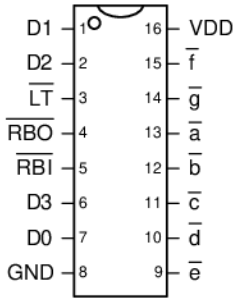
74377 Register

8-bit Enableable Register
CLK: clock
$D_{7:0}$: data
$Q_{7:0}$: output
ENb: enable

```
always @(posedge clk)
  if (~ENb) Q <= D;
```

Note: Verilog variable names cannot start with numbers, but the names in the example code in Figure A.2 are chosen to match the manufacturer's data sheet.

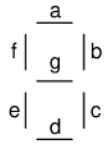
Figure A.2 Medium-scale integration chips



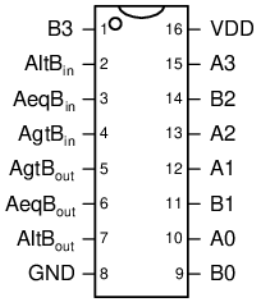
7447 7-Segment Decoder

7-segment Display Decoder

$D_{3:0}$: data
a...f: segments (low = ON)
LTb: light test
RBIb: ripple blanking in
RBOb: ripple blanking out



RBO	LT	RBI	D3:0	a	b	c	d	e	f	g
0	x	x	x	1	1	1	1	1	1	1
1	0	x	x	0	0	0	0	0	0	0
x	1	0	0000	1	1	1	1	1	1	1
1	1	1	0000	0	0	0	0	0	0	1
1	1	1	0001	1	0	0	1	1	1	1
1	1	1	0010	0	0	1	0	0	1	0
1	1	1	0011	0	0	0	0	1	1	0
1	1	1	0100	1	0	0	1	1	0	0
1	1	1	0101	0	1	0	0	1	0	0
1	1	1	0110	1	1	0	0	0	0	0
1	1	1	0111	0	0	0	1	1	1	1
1	1	1	1000	0	0	0	0	0	0	0
1	1	1	1001	0	0	0	1	1	0	0
1	1	1	1010	1	1	1	0	0	1	0
1	1	1	1011	1	1	0	0	1	1	0
1	1	1	1100	1	0	1	1	1	0	0
1	1	1	1101	0	1	1	0	1	0	0
1	1	1	1110	0	0	0	1	1	1	1
1	1	1	1111	0	0	0	0	0	0	0

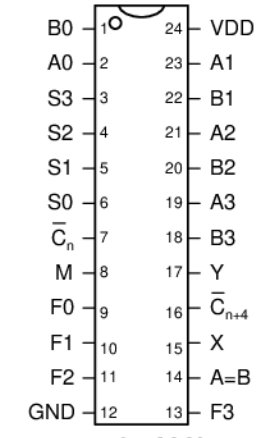


7485 Comparator

4-bit Comparator

$A_{3:0}$, $B_{3:0}$: data
rel_{in}: input relation
rel_{out}: output relation

```
always @(*)
  if (A > B | (A == B & AgtBin)) begin
    AgtBout = 1; AeqBout = 0; AltBout = 0;
  end
  else if (A < B | (A == B & AltBin) begin
    AgtBout = 0; AeqBout = 0; AltBout = 1;
  end else begin
    AgtBout = 0; AeqBout = 1; AltBout = 0;
  end
```



74181 ALU

4-bit ALU

$A_{3:0}$, $B_{3:0}$: inputs
$Y_{3:0}$: output
$F_{3:0}$: function select
M: mode select
Cb_n: carry in
Cb_{nplus4}: carry out
AeqB: equality (in some modes)
X,Y: carry lookahead adder outputs

```
always @(*)
  case (F)
    0000: Y = M ? ~A       :  A                  + ~Cbn;
    0001: Y = M ? ~(A | B) :  A       +  B       + ~Cbn;
    0010: Y = M ? (~A) & B :  A       + ~B       + ~Cbn;
    0011: Y = M ? 4'b0000  :  4'b1111            + ~Cbn;
    0100: Y = M ? ~(A & B) :  A       + (A & ~B) + ~Cbn;
    0101: Y = M ? ~B       : (A | B)  + (A & ~B) + ~Cbn;
    0110: Y = M ? A ^ B    :  A       - B        - Cbn;
    0111: Y = M ? A & ~B   : (A & ~B)            - Cbn;
    1000: Y = M ? ~A + B   :  A       + (A & B)  + ~Cbn;
    1001: Y = M ? ~(A ^ B) :  A       +  B       + ~Cbn;
    1010: Y = M ? B        : (A | ~B) + (A & B)  + ~Cbn;
    1011: Y = M ? A & B    : (A & B)             + ~Cbn;
    1100: Y = M ? 1        :  A       +  A       + ~Cbn;
    1101: Y = M ? A | ~B   : (A | B)  +  A       + ~Cbn;
    1110: Y = M ? A | B    : (A | ~B) +  A       + ~Cbn;
    1111: Y = M ? A        :  A                  - Cbn;
  endcase
```

Figure A.3 More medium-scale integration (MSI) chips

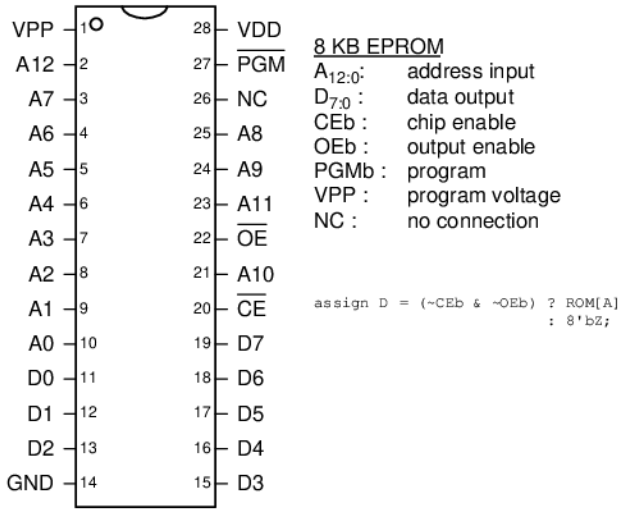


Figure A.4 2764 8KB EPROM

involve replacing the contents of the PROM rather than rewiring connections between chips. Lookup tables are useful for small functions but become prohibitively expensive as the number of inputs grows.

For example, the classic 2764 8-KB (64-Kb) erasable PROM (EPROM) is shown in Figure A.4. The EPROM has 13 address lines to specify one of the 8K words and 8 data lines to read the byte of data at that word. The chip enable and output enable must both be asserted for data to be read. The maximum propagation delay is 200 ps. In normal operation, $\overline{PGM} = 1$ and *VPP* is not used. The EPROM is usually programmed on a special programmer that sets $\overline{PGM} = 0$, applies 13 V to *VPP*, and uses a special sequence of inputs to configure the memory.

Modern PROMs are similar in concept but have much larger capacities and more pins. Flash memory is the cheapest type of PROM, selling for about $30 per gigabyte in 2006. Prices have historically declined by 30 to 40% per year.

A.3.2 PLAs

As discussed in Section 5.6.1, PLAs contain AND and OR planes to compute any combinational function written in sum-of-products form. The AND and OR planes can be programmed using the same techniques for PROMs. A PLA has two columns for each input and one column for each output. It has one row for each minterm. This organization is more efficient than a PROM for many functions, but the array still grows excessively large for functions with numerous I/Os and minterms.

Many different manufacturers have extended the basic PLA concept to build *programmable logic devices* (*PLDs*) that include registers.

The 22V10 is one of the most popular classic PLDs. It has 12 dedicated input pins and 10 outputs. The outputs can come directly from the PLA or from clocked registers on the chip. The outputs can also be fed back into the PLA. Thus, the 22V10 can directly implement FSMs with up to 12 inputs, 10 outputs, and 10 bits of state. The 22V10 costs about $2 in quantities of 100. PLDs have been rendered mostly obsolete by the rapid improvements in capacity and cost of FPGAs.

A.3.3 FPGAs

As discussed in Section 5.6.2, FPGAs consist of arrays of *configurable logic blocks* (*CLBs*) connected together with programmable wires. The CLBs contain small lookup tables and flip-flops. FPGAs scale gracefully to extremely large capacities, with thousands of lookup tables. Xilinx and Altera are two of the leading FPGA manufacturers.

Lookup tables and programmable wires are flexible enough to implement any logic function. However, they are an order of magnitude less efficient in speed and cost (chip area) than hard-wired versions of the same functions. Thus, FPGAs often include specialized blocks, such as memories, multipliers, and even entire microprocessors.

Figure A.5 shows the design process for a digital system on an FPGA. The design is usually specified with a hardware description language (HDL), although some FPGA tools also support schematics. The design is then simulated. Inputs are applied and compared against expected outputs to *verify* that the logic is correct. Usually some debugging is required. Next, logic *synthesis* converts the HDL into Boolean functions. Good synthesis tools produce a schematic of the functions, and the prudent designer examines these schematics, as well as any warnings produced during synthesis, to ensure that the desired logic was produced. Sometimes sloppy coding leads to circuits that are much larger than intended or to circuits with asynchronous logic. When the synthesis results are good, the FPGA tool *maps* the functions onto the CLBs of a specific chip. The *place and route* tool determines which functions go in which lookup tables and how they are wired together. Wire delay increases with length, so critical circuits should be placed close together. If the design is too big to fit on the chip, it must be reengineered. *Timing analysis* compares the timing constraints (e.g., an intended clock speed of 100 MHz) against the actual circuit delays and reports any errors. If the logic is too slow, it may have to be redesigned or pipelined differently. When the design is correct, a file is generated specifying the contents of all the CLBs and the programming of all the wires on the FPGA. Many FPGAs store this *configuration* information in static RAM that must be reloaded each time the FPGA is turned on. The FPGA can download this information from a computer in the

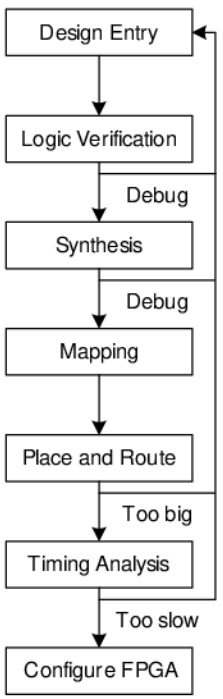


Figure A.5 FPGA design flow

laboratory, or can read it from a nonvolatile ROM when power is first applied.

Example A.1 FPGA TIMING ANALYSIS

Alyssa P. Hacker is using an FPGA to implement an M&M sorter with a color sensor and motors to put red candy in one jar and green candy in another. Her design is implemented as an FSM, and she is using a Spartan XC3S200 FPGA, a chip from the Spartan 3 series family. According to the data sheet, the FPGA has the timing characteristics shown in Table A.1. Assume that the design is small enough that wire delay is negligible.

Alyssa would like her FSM to run at 100 MHz. What is the maximum number of CLBs on the critical path? What is the fastest speed at which her FSM could possibly run?

SOLUTION: At 100 MHz, the cycle time, T_c, is 10 ns. Alyssa uses Equation 3.13 figure to out the minimum combinational propagation delay, t_{pd}, at this cycle time:

$$t_{pd} \leq 10 \text{ ns} - (0.72 \text{ ns} + 0.53 \text{ ns}) = 8.75 \text{ ns} \quad \text{(A.1)}$$

Alyssa's FSM can use at most 14 consecutive CLBs (8.75/0.61) to implement the next-state logic.

The fastest speed at which an FSM will run on a Spartan 3 FPGA is when it is using a single CLB for the next state logic. The minimum cycle time is

$$T_c \geq 0.61 \text{ ns} + 0.72 \text{ ns} + 0.53 \text{ ns} = 1.86 \text{ ns} \quad \text{(A.2)}$$

Therefore, the maximum frequency is 538 MHz.

Table A.1 Spartan 3 XC3S200 timing

name	value (ns)
t_{pcq}	0.72
t_{setup}	0.53
t_{hold}	0
t_{pd} (per CLB)	0.61
t_{skew}	0

Xilinx advertises the XC3S100E FPGA with 1728 lookup tables and flip-flops for $2 in quantities of 500,000 in 2006. In more modest quantities, medium-sized FPGAs typically cost about $10, and the largest

FPGAs cost hundreds or even thousands of dollars. The cost has declined at approximately 30% per year, so FPGAs are becoming extremely popular.

A.4 APPLICATION-SPECIFIC INTEGRATED CIRCUITS

Application-specific integrated circuits (*ASICs*) are chips designed for a particular purpose. Graphics accelerators, network interface chips, and cell phone chips are common examples of ASICS. The ASIC designer places transistors to form logic gates and wires the gates together. Because the ASIC is hardwired for a specific function, it is typically several times faster than an FPGA and occupies an order of magnitude less chip area (and hence cost) than an FPGA with the same function. However, the *masks* specifying where transistors and wires are located on the chip cost hundreds of thousands of dollars to produce. The fabrication process usually requires 6 to 12 weeks to manufacture, package, and test the ASICs. If errors are discovered after the ASIC is manufactured, the designer must correct the problem, generate new masks, and wait for another batch of chips to be fabricated. Hence, ASICs are suitable only for products that will be produced in large quantities and whose function is well defined in advance.

Figure A.6 shows the ASIC design process, which is similar to the FPGA design process of Figure A.5. Logic verification is especially important because correction of errors after the masks are produced is expensive. Synthesis produces a *netlist* consisting of logic gates and connections between the gates; the gates in this netlist are placed, and the wires are routed between gates. When the design is satisfactory, masks are generated and used to fabricate the ASIC. A single speck of dust can ruin an ASIC, so the chips must be tested after fabrication. The fraction of manufactured chips that work is called the *yield*; it is typically 50 to 90%, depending on the size of the chip and the maturity of the manufacturing process. Finally, the working chips are placed in packages, as will be discussed in Section A.7.

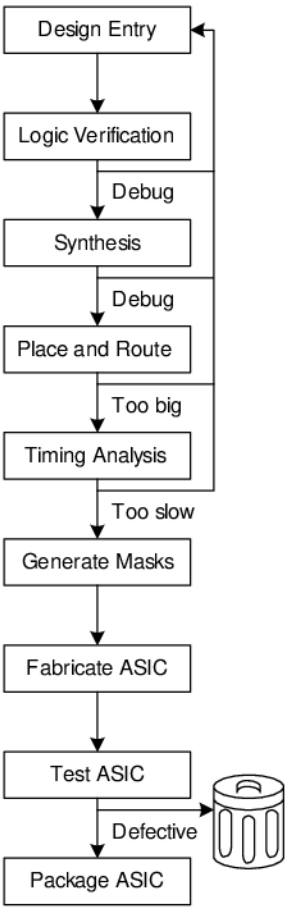


Figure A.6 ASIC design flow

A.5 DATA SHEETS

Integrated circuit manufacturers publish *data sheets* that describe the functions and performance of their chips. It is essential to read and understand the data sheets. One of the leading sources of errors in digital systems comes from misunderstanding the operation of a chip.

Data sheets are usually available from the manufacturer's Web site. If you cannot locate the data sheet for a part and do not have clear documentation from another source, don't use the part. Some of the entries in the data sheet may be cryptic. Often the manufacturer publishes data books containing data sheets for many related parts. The

beginning of the data book has additional explanatory information. This information can usually be found on the Web with a careful search.

This section dissects the Texas Instruments (TI) data sheet for a 74HC04 inverter chip. The data sheet is relatively simple but illustrates many of the major elements. TI still manufacturers a wide variety of 74xx-series chips. In the past, many other companies built these chips too, but the market is consolidating as the sales decline.

Figure A.7 shows the first page of the data sheet. Some of the key sections are highlighted in blue. The title is SN54HC04, SN74HC04 HEX INVERTERS. HEX INVERTERS means that the chip contains six inverters. SN indicates that TI is the manufacturer. Other manufacture codes include MC for Motorola and DM for National Semiconductor. You can generally ignore these codes, because all of the manufacturers build compatible 74xx-series logic. HC is the logic family (high speed CMOS). The logic family determines the speed and power consumption of the chip, but not the function. For example, the 7404, 74HC04, and 74LS04 chips all contain six inverters, but they differ in performance and cost. Other logic families are discussed in Section A.6. The 74xx chips operate across the commercial or industrial temperature range (0 to 70 °C or −40 to 85 °C, respectively), whereas the 54xx chips operate across the military temperature range (−55 to 125 °C) and sell for a higher price but are otherwise compatible.

The 7404 is available in many different packages, and it is important to order the one you intended when you make a purchase. The packages are distinguished by a suffix on the part number. N indicates a *plastic dual inline package* (*PDIP*), which fits in a breadboard or can be soldered in through-holes in a printed circuit board. Other packages are discussed in Section A.7.

The function table shows that each gate inverts its input. If A is HIGH (H), Y is LOW (L) and vice versa. The table is trivial in this case but is more interesting for more complex chips.

Figure A.8 shows the second page of the data sheet. The logic diagram indicates that the chip contains inverters. The *absolute maximum* section indicates conditions beyond which the chip could be destroyed. In particular, the power supply voltage (V_{CC}, also called V_{DD} in this book) should not exceed 7 V. The continuous output current should not exceed 25 mA. The *thermal resistance* or impedance, θ_{JA}, is used to calculate the temperature rise caused by the chip's dissipating power. If the *ambient* temperature in the vicinity of the chip is T_A and the chip dissipates P_{chip}, then the temperature on the chip itself at its *junction* with the package is

$$T_J = T_A + P_{\text{chip}}\,\theta_{JA} \tag{A.3}$$

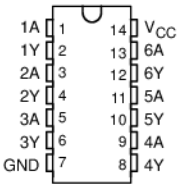
For example, if a 7404 chip in a plastic DIP package is operating in a hot box at 50 °C and consumes 20 mW, the junction temperature will

SN54HC04, SN74HC04
HEX INVERTERS

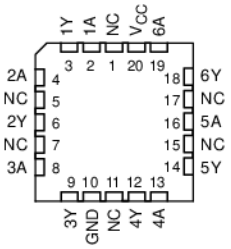
SCLS078D – DECEMBER 1982 – REVISED JULY 2003

- Wide Operating Voltage Range of 2 V to 6 V
- Outputs Can Drive Up To 10 LSTTL Loads
- Low Power Consumption, 20-μA Max I_{CC}
- Typical tpd = 8 ns
- ±4-mA Output Drive at 5 V
- Low Input Current of 1 μA Max

SN54HC04 . . . J OR W PACKAGE
SN74HC04 . . . D, N, NS, OR PW PACKAGE
(TOPVIEW)



SN54HC04 . . . FK PACKAGE
(TOPVIEW)



NC – No internal connection

description/ordering information

The 'HC04 devices contain six independent inverters. They perform the Boolean function $Y = \overline{A}$ in positive logic.

ORDERING INFORMATION

T_A	PACKAGE†		ORDERABLE PARTNUMBER	TOP-SIDE MARKING
–40°C to 85°C	PDIP – N	Tube of 25	SN74HC04N	SN74HC04N
	SOIC – D	Tube of 50	SN74HC04D	HC04
		Reel of 2500	SN74HC04DR	
		Reel of 250	SN74HC04DT	
	SOP – NS	Reel of 2000	SN74HC04NSR	HC04
	TSSOP – PW	Tube of 90	SN74HC04PW	HC04
		Reel of 2000	SN74HC04PWR	
		Reel of 250	SN74HC04PWT	
–55°C to 125°C	CDIP – J	Tube of 25	SNJ54HC04J	SNJ54HC04J
	CFP – W	Tube of 150	SNJ54HC04W	SNJ54HC04W
	LCCC – FK	Tube of 55	SNJ54HC04FK	SNJ54HC04FK

† Package drawings, standard packing quantities, thermal data, symbolization, and PCB design guidelines are available at www.ti.com/sc/package.

FUNCTION TABLE
(each inverter)

INPUT A	OUTPUT Y
H	L
L	H

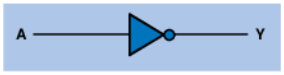
TEXAS INSTRUMENTS
POST OFFICE BOX 655303 • DALLAS, TEXAS 75265

Figure A.7 7404 data sheet page 1

SN54HC04, SN74HC04
HEX INVERTERS

SCLS078D – DECEMBER 1982 – REVISED JULY 2003

logic diagram (positive logic)



absolute maximum ratings over operating free-air temperature range (unless otherwise noted)†

Supply voltage range, V_{CC} . . . . . –0.5 V to 7 V
Input clamp current, I_{IK} ($V_I < 0$ or $V_I > V_{CC}$) (see Note 1) . . . . . ±20 mA
Output clamp current, I_{OK} ($V_O < 0$ or $V_O > V_{CC}$) (see Note 1) . . . . . ±20 mA
Continuous output current, I_O ($V_O = 0$ to V_{CC}) . . . . . ±25 mA
Continuous current through V_{CC} or GND . . . . . ±50 mA
Package thermal impedance, θ_{JA} (see Note 2): D package . . . . . 86° C/W
N package . . . . . 80° C/W
NS package . . . . . 76° C/W
PW package . . . . . 131° C/W
Storage temperature range, T_{stg} . . . . . –65° C to 150° C

† Stresses beyond those listed under "absolute maximum ratings" may cause permanent damage to the device. These are stress ratings only, and functional operation of the device at these or any other conditions beyond those indicated under "recommended operating conditions" is not implied. Exposure to absolute-maximum-rated conditions for extended periods may affect device reliability.

NOTES: 1. The input and output voltage ratings may be exceeded if the input and output current ratings are observed.
2. The package thermal impedance is calculated in accordance with JESD 51-7.

recommended operating conditions (see Note 3)

			SN54HC04			SN74HC04			UNIT
			MIN	NOM	MAX	MIN	NOM	MAX	
V_{CC}	Supply voltage		2	5	6	2	5	6	V
V_{IH}	High-level input voltage	V_{CC} = 2 V	1.5			1.5			V
		V_{CC} = 4.5 V	3.15			3.15			
		V_{CC} = 6 V	4.2			4.2			
V_{IL}	Low-level input voltage	V_{CC} = 2 V			0.5			0.5	V
		V_{CC} = 4.5 V			1.35			1.35	
		V_{CC} = 6 V			1.8			1.8	
V_I	Input voltage		0		V_{CC}	0		V_{CC}	V
V_O	Output voltage		0		V_{CC}	0		V_{CC}	V
$\Delta t/\Delta v$	Input transition rise/fall time	V_{CC} = 2 V			1000			1000	ns
		V_{CC} = 4.5 V			500			500	
		V_{CC} = 6 V			400			400	
T_A	Operating free-air temperature		–55		125	–40		85	°C

NOTE 3: All unused inputs of the device must be held at V_{CC} or GND to ensure proper device operation. Refer to the TI application report, *Implications of Slow or Floating CMOS Inputs*, literature number SCBA004.

Figure A.8 7404 datasheet page 2

climb to 50°C + 0.02 W × 80°C/W = 51.6°C. Internal power dissipation is seldom important for 74xx-series chips, but it becomes important for modern chips that dissipate tens of watts or more.

The *recommended operating conditions* define the environment in which the chip should be used. Within these conditions, the chip should meet specifications. These conditions are more stringent than the absolute maximums. For example, the power supply voltage should be between 2 and 6 V. The input logic levels for the HC logic family depend on V_{DD}. Use the 4.5 V entries when $V_{DD} = 5$ V, to allow for a 10% droop in the power supply caused by noise in the system.

Figure A.9 shows the third page of the data sheet. The *electrical characteristics* describe how the device performs when used within the recommended operating conditions if the inputs are held constant. For example, if $V_{CC} = 5$ V (and droops to 4.5 V) and the output current, I_{OH}/I_{OL} does not exceed 20 μA, $V_{OH} = 4.4$ V and $V_{OL} = 0.1$ V in the worst case. If the output current increases, the output voltages become less ideal, because the transistors on the chip struggle to provide the current. The HC logic family uses CMOS transistors that draw very little current. The current into each input is guaranteed to be less than 1000 nA and is typically only 0.1 nA at room temperature. The *quiescent* power supply current (I_{DD}) drawn while the chip is idle is less than 20 μA. Each input has less than 10 pF of capacitance.

The *switching characteristics* define how the device performs when used within the recommended operating conditions if the inputs change. The *propagation delay*, t_{pd}, is measured from when the input passes through 0.5 V_{CC} to when the output passes through 0.5 V_{CC}. If V_{CC} is nominally 5 V and the chip drives a capacitance of less than 50 pF, the propagation delay will not exceed 24 ns (and typically will be much faster). Recall that each input may present 10 pF, so the chip cannot drive more than five identical chips at full speed. Indeed, stray capacitance from the wires connecting chips cuts further into the useful load. The *transition time*, also called the rise/fall time, is measured as the output transitions between 0.1 V_{CC} and 0.9 V_{CC}.

Recall from Section 1.8 that chips consume both *static* and *dynamic power*. Static power is low for HC circuits. At 85 °C, the maximum quiescent supply current is 20 μA. At 5 V, this gives a static power consumption of 0.1 mW. The dynamic power depends on the capacitance being driven and the switching frequency. The 7404 has an internal power dissipation capacitance of 20 pF per inverter. If all six inverters on the 7404 switch at 10 MHz and drive external loads of 25 pF, then the dynamic power given by Equation 1.4 is $\frac{1}{2}(6)(20\text{ pF} + 25\text{ pF})(5^2)(10\text{ MHz}) = 33.75$ mW and the maximum total power is 33.85 mW.

electrical characteristics over recommended operating free-air temperature range (unless otherwise noted)

PARAMETER	TEST CONDITIONS		V_{CC}	T_A = 25°C			SN54HC04		SN74HC04		UNIT
				MIN	TYP	MAX	MIN	MAX	MIN	MAX	
V_{OH}	$V_I = V_{IH}$ or V_{IL}	I_{OH} = –20 µA	2 V	1.9	1.998		1.9		1.9		V
			4.5 V	4.4	4.499		4.4		4.4		
			6 V	5.9	5.999		5.9		5.9		
		I_{OH} = –4 mA	4.5 V	3.98	4.3		3.7		3.84		
		I_{OH} = –5.2 mA	6 V	5.48	5.8		5.2		5.34		
V_{OL}	$V_I = V_{IH}$ or V_{IL}	I_{OL} = 20 µA	2 V		0.002	0.1		0.1		0.1	V
			4.5 V		0.001	0.1		0.1		0.1	
			6 V		0.001	0.1		0.1		0.1	
		I_{OL} = 4 mA	4.5 V		0.17	0.26		0.4		0.33	
		I_{OL} = 5.2 mA	6 V		0.15	0.26		0.4		0.33	
I_I	$V_I = V_{CC}$ or 0		6 V		±0.1	±100		±1000		±1000	nA
I_{CC}	$V_I = V_{CC}$ or 0, I_O = 0		6 V			2		40		20	µA
C_i			2 V to 6 V		3	10		10		10	pF

switching characteristics over recommended operating free-air temperature range, CL = 50 pF (unless otherwise noted) (see Figure 1)

PARAMETER	FROM (INPUT)	TO (OUTPUT)	V_{CC}	T_A = 25°C			SN54HC04		SN74HC04		UNIT
				MIN	TYP	MAX	MIN	MAX	MIN	MAX	
t_{pd}	A	Y	2 V		45	95		145		120	ns
			4.5 V		9	19		29		24	
			6 V		8	16		25		20	
t_t		Y	2 V		38	75		110		95	ns
			4.5 V		8	15		22		19	
			6 V		6	13		19		16	

operating characteristics, T_A = 25°C

PARAMETER		TEST CONDITIONS	TYP	UNIT
C_{pd}	Power dissipation capacitance per inverter	No load	20	pF

Figure A.9 7404 datasheet page 3

A.6 LOGIC FAMILIES

The 74xx-series logic chips have been manufactured using many different technologies, called *logic families*, that offer different speed, power, and logic level trade-offs. Other chips are usually designed to be compatible with some of these logic families. The original chips, such as the 7404, were built using bipolar transistors in a technology called *Transistor-Transistor Logic* (*TTL*). Newer technologies add one or more letters after the 74 to indicate the logic family, such as 74LS04, 74HC04, or 74AHCT04. Table A.2 summarizes the most common 5-V logic families.

Advances in bipolar circuits and process technology led to the *Schottky* (*S*) and *Low-Power Schottky* (*LS*) families. Both are faster than TTL. Schottky draws more power, whereas Low-power Schottky draws less. *Advanced Schottky* (*AS*) and *Advanced Low-Power Schottky* (*ALS*) have improved speed and power compared to S and LS. *Fast* (*F*) logic is faster and draws less power than AS. All of these families provide more current for LOW outputs than for HIGH outputs and hence have asymmetric logic levels. They conform to the "TTL" logic levels: $V_{IH} = 2$ V, $V_{IL} = 0.8$ V, $V_{OH} > 2.4$ V, and $V_{OL} < 0.5$ V.

Table A.2 Typical specifications for 5-V logic families

	Bipolar / TTL						CMOS		CMOS / TTL Compatible	
Characteristic	TTL	S	LS	AS	ALS	F	HC	AHC	HCT	AHCT
t_{pd} (ns)	22	9	12	7.5	10	6	21	7.5	30	7.7
V_{IH} (V)	2	2	2	2	2	2	3.15	3.15	2	2
V_{IL} (V)	0.8	0.8	0.8	0.8	0.8	0.8	1.35	1.35	0.8	0.8
V_{OH} (V)	2.4	2.7	2.7	2.5	2.5	2.5	3.84	3.8	3.84	3.8
V_{OL} (V)	0.4	0.5	0.5	0.5	0.5	0.5	0.33	0.44	0.33	0.44
I_{OH} (mA)	0.4	1	0.4	2	0.4	1	4	8	4	8
I_{OL} (mA)	16	20	8	20	8	20	4	8	4	8
I_{IL} (mA)	1.6	2	0.4	0.5	0.1	0.6	0.001	0.001	0.001	0.001
I_{IH} (mA)	0.04	0.05	0.02	0.02	0.02	0.02	0.001	0.001	0.001	0.001
I_{DD} (mA)	33	54	6.6	26	4.2	15	0.02	0.02	0.02	0.02
C_{pd} (pF)			n/a				20	12	20	14
cost* (US $)	obsolete	0.57	0.29	0.53	0.33	0.20	0.15	0.15	0.15	0.15

* Per unit in quantities of 1000 for the 7408 from Texas Instruments in 2006

As CMOS circuits matured in the 1980s and 1990s, they became popular because they draw very little power supply or input current. The *High Speed CMOS* (*HC*) and *Advanced High Speed CMOS* (*AHC*) families draw almost no static power. They also deliver the same current for HIGH and LOW outputs. They conform to the "CMOS" logic levels: $V_{IH} = 3.15$ V, $V_{IL} = 1.35$ V, $V_{OH} > 3.8$ V, and $V_{OL} < 0.44$ V. Unfortunately, these levels are incompatible with TTL circuits, because a TTL HIGH output of 2.4 V may not be recognized as a legal CMOS HIGH input. This motivates the use of *High Speed TTL-compatible CMOS* (*HCT*) and *Advanced High Speed TTL-compatible CMOS* (*AHCT*), which accept TTL input logic levels and generate valid CMOS output logic levels. These families are slightly slower than their pure CMOS counterparts. All CMOS chips are sensitive to *electrostatic discharge* (*ESD*) caused by static electricity. Ground yourself by touching a large metal object before handling CMOS chips, lest you zap them.

The 74xx-series logic is inexpensive. The newer logic families are often cheaper than the obsolete ones. The LS family is widely available and robust and is a popular choice for laboratory or hobby projects that have no special performance requirements.

The 5-V standard collapsed in the mid-1990s, when transistors became too small to withstand the voltage. Moreover, lower voltage offers lower power consumption. Now 3.3, 2.5, 1.8, 1.2, and even lower voltages are commonly used. The plethora of voltages raises challenges in communicating between chips with different power supplies. Table A.3 lists some of the low-voltage logic families. Not all 74xx parts are available in all of these logic families.

All of the low-voltage logic families use CMOS transistors, the workhorse of modern integrated circuits. They operate over a wide range of V_{DD}, but the speed degrades at lower voltage. *Low-Voltage CMOS* (*LVC*) logic and *Advanced Low-Voltage CMOS* (*ALVC*) logic are commonly used at 3.3, 2.5, or 1.8 V. LVC withstands inputs up to 5.5 V, so it can receive inputs from 5-V CMOS or TTL circuits. *Advanced Ultra-Low-Voltage CMOS (AUC)* is commonly used at 2.5, 1.8, or 1.2 V and is exceptionally fast. Both ALVC and AUC withstand inputs up to 3.6 V, so they can receive inputs from 3.3-V circuits.

FPGAs often offer separate voltage supplies for the internal logic, called the *core*, and for the input/output (I/O) pins. As FPGAs have advanced, the core voltage has dropped from 5 to 3.3, 2.5, 1.8, and 1.2 V to save power and avoid damaging the very small transistors. FPGAs have configurable I/Os that can operate at many different voltages, so as to be compatible with the rest of the system.

Table A.3 Typical specifications for low-voltage logic families

	LVC			ALVC			AUC		
V_{dd} (V)	3.3	2.5	1.8	3.3	2.5	1.8	2.5	1.8	1.2
t_{pd} (ns)	4.1	6.9	9.8	2.8	3	?[1]	1.8	2.3	3.4
V_{IH} (V)	2	1.7	1.17	2	1.7	1.17	1.7	1.17	0.78
V_{IL} (V)	0.8	0.7	0.63	0.8	0.7	0.63	0.7	0.63	0.42
V_{OH} (V)	2.2	1.7	1.2	2	1.7	1.2	1.8	1.2	0.8
V_{OL} (V)	0.55	0.7	0.45	0.55	0.7	0.45	0.6	0.45	0.3
I_O (mA)	24	8	4	24	12	12	9	8	3
I_I (mA)		0.02			0.005			0.005	
I_{DD} (mA)		0.01			0.01			0.01	
C_{pd} (pF)	10	9.8	7	27.5	23	?*	17	14	14
cost (US $)		0.17			0.20			not available	

* Delay and capacitance not available at the time of writing

A.7 PACKAGING AND ASSEMBLY

Integrated circuits are typically placed in *packages* made of plastic or ceramic. The packages serve a number of functions, including connecting the tiny metal I/O pads of the chip to larger pins in the package for ease of connection, protecting the chip from physical damage, and spreading the heat generated by the chip over a larger area to help with cooling. The packages are placed on a breadboard or printed circuit board and wired together to assemble the system.

Packages

Figure A.10 shows a variety of integrated circuit packages. Packages can be generally categorized as *through-hole* or *surface mount* (*SMT*). Through-hole packages, as their name implies, have pins that can be inserted through holes in a printed circuit board or into a socket. *Dual inline packages* (*DIPs*) have two rows of pins with 0.1-inch spacing between pins. *Pin grid arrays* (*PGAs*) support more pins in a smaller package by placing the pins under the package. SMT packages are soldered directly to the surface of a printed circuit board without using holes. Pins on SMT parts are called *leads*. The *thin small outline package* (*TSOP*) has two rows of closely spaced leads (typically 0.02-inch spacing). *Plastic leaded chip carriers* (*PLCCs*) have J-shaped leads on all four

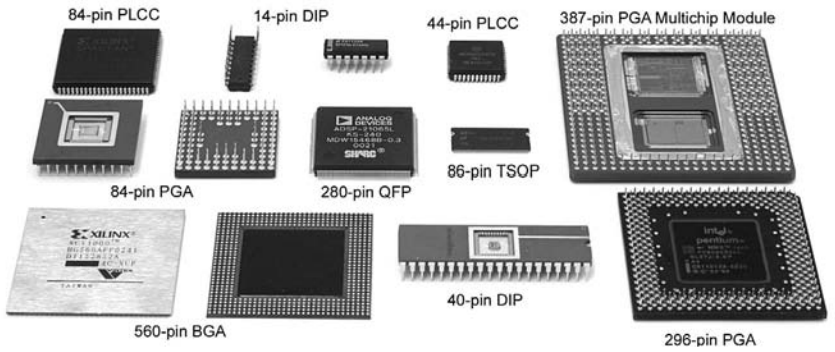


Figure A.10 Integrated circuit packages

sides, with 0.05-inch spacing. They can be soldered directly to a board or placed in special sockets. *Quad flat packs* (*QFPs*) accommodate a large number of pins using closely spaced legs on all four sides. *Ball grid arrays* (*BGAs*) eliminate the legs altogether. Instead, they have hundreds of tiny solder balls on the underside of the package. They are carefully placed over matching pads on a printed circuit board, then heated so that the solder melts and joins the package to the underlying board.

Breadboards

DIPs are easy to use for prototyping, because they can be placed in a *breadboard*. A breadboard is a plastic board containing rows of sockets, as shown in Figure A.11. All five holes in a row are connected together. Each pin of the package is placed in a hole in a separate row. Wires can be placed in adjacent holes in the same row to make connections to the pin. Breadboards often provide separate columns of connected holes running the height of the board to distribute power and ground.

Figure A.11 shows a breadboard containing a majority gate built with a 74LS08 AND chip and a 74LS32 OR chip. The schematic of the circuit is shown in Figure A.12. Each gate in the schematic is labeled with the chip (08 or 32) and the pin numbers of the inputs and outputs (see Figure A.1). Observe that the same connections are made on the breadboard. The inputs are connected to pins 1, 2, and 5 of the 08 chip, and the output is measured at pin 6 of the 32 chip. Power and ground are connected to pins 14 and 7, respectively, of each chip, from the vertical power and ground columns that are attached to the banana plug receptacles, Vb and Va. Labeling the schematic in this way and checking off connections as they are made is a good way to reduce the number of mistakes made during breadboarding.

Unfortunately, it is easy to accidentally plug a wire in the wrong hole or have a wire fall out, so breadboarding requires a great deal of care (and usually some debugging in the laboratory). Breadboards are suited only to prototyping, not production.

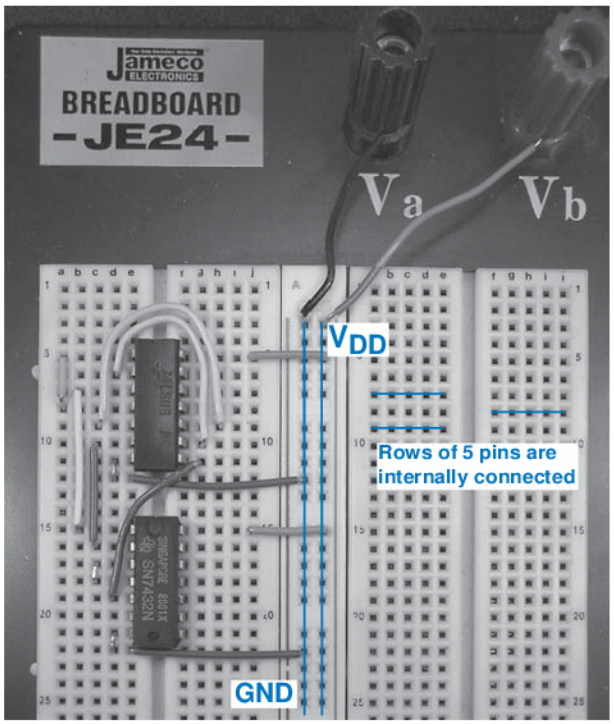


Figure A.11 **Majority circuit on breadboard**

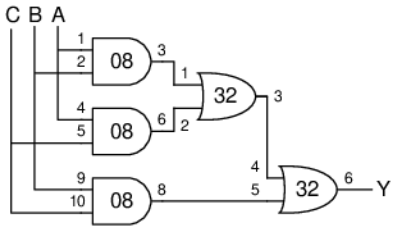


Figure A.12 **Majority gate schematic with chips and pins identified**

Printed Circuit Boards

Instead of breadboarding, chip packages may be soldered to a *printed circuit board* (*PCB*). The PCB is formed of alternating layers of conducting copper and insulating epoxy. The copper is etched to form wires called *traces*. Holes called *vias* are drilled through the board and plated with metal to connect between layers. PCBs are usually designed with *computer-aided design* (*CAD*) tools. You can etch and drill your own simple boards in the laboratory, or you can send the board design to a specialized factory for inexpensive mass production. Factories have turnaround times of days (or weeks, for cheap mass production runs) and typically charge a few hundred dollars in setup fees and a few dollars per board for moderately complex boards built in large quantities.

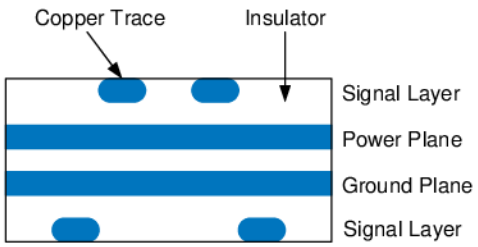


Figure A.13 Printed circuit board cross-section

PCB traces are normally made of copper because of its low resistance. The traces are embedded in an insulating material, usually a green, fire-resistant plastic called FR4. A PCB also typically has copper power and ground layers, called *planes*, between signal layers. Figure A.13 shows a cross-section of a PCB. The signal layers are on the top and bottom, and the power and ground planes are embedded in the center of the board. The power and ground planes have low resistance, so they distribute stable power to components on the board. They also make the capacitance and inductance of the traces uniform and predictable.

Figure A.14 shows a PCB for a 1970s vintage Apple II+ computer. At the top is a Motorola 6502 microprocessor. Beneath are six 16-Kb ROM chips forming 12 KB of ROM containing the operating system. Three rows of eight 16-Kb DRAM chips provide 48 KB of RAM. On the right are several rows of 74xx-series logic for memory address decoding and other functions. The lines between chips are traces that wire the chips together. The dots at the ends of some of the traces are vias filled with metal.

Putting It All Together

Most modern chips with large numbers of inputs and outputs use SMT packages, especially QFPs and BGAs. These packages require a printed circuit board rather than a breadboard. Working with BGAs is especially challenging because they require specialized assembly equipment. Moreover, the balls cannot be probed with a voltmeter or oscilloscope during debugging in the laboratory, because they are hidden under the package.

In summary, the designer needs to consider packaging early on to determine whether a breadboard can be used during prototyping and whether BGA parts will be required. Professional engineers rarely use breadboards when they are confident of connecting chips together correctly without experimentation.

A.8 TRANSMISSION LINES

We have assumed so far that wires are *equipotential* connections that have a single voltage along their entire length. Signals actually propagate along wires at the speed of light in the form of electromagnetic waves. If the wires are short enough or the signals change slowly, the equipotential assumption

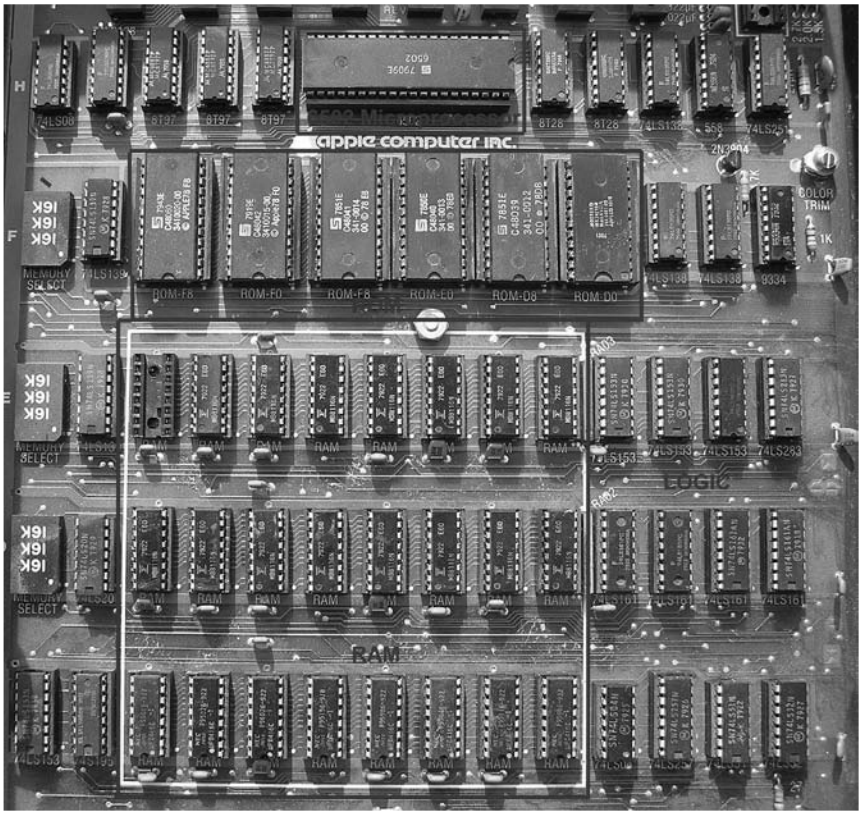


Figure A.14 **Apple II+ circuit board**

is good enough. When the wire is long or the signal is very fast, the *transmission time* along the wire becomes important to accurately determine the circuit delay. We must model such wires as *transmission lines*, in which a wave of voltage and current propagates at the speed of light. When the wave reaches the end of the line, it may reflect back along the line. The reflection may cause noise and odd behaviors unless steps are taken to limit it. Hence, the digital designer must consider transmission line behavior to accurately account for the delay and noise effects in long wires.

Electromagnetic waves travel at the speed of light in a given medium, which is fast but not instantaneous. The speed of light depends on the *permittivity*, ε, and *permeability*, μ, of the medium[2]: $v = \frac{1}{\sqrt{\mu\varepsilon}} = \frac{1}{\sqrt{LC}}$.

[2] The capacitance, C, and inductance, L, of a wire are related to the permittivity and permeability of the physical medium in which the wire is located.

The speed of light in free space is $v = c = 3 \times 10^8$ m/s. Signals in a PCB travel at about half this speed, because the FR4 insulator has four times the permittivity of air. Thus, PCB signals travel at about 1.5×10^8 m/s, or 15 cm/ns. The time delay for a signal to travel along a transmission line of length l is

$$t_d = l/v \tag{A.4}$$

The *characteristic impedance* of a transmission line, Z_0 (pronounced "Z-naught"), is the ratio of voltage to current in a wave traveling along the line: $Z_0 = V/I$. It is *not* the resistance of the wire (a good transmission line in a digital system typically has negligible resistance). Z_0 depends on the inductance and capacitance of the line (see the derivation in Section A.8.7) and typically has a value of 50 to 75 Ω.

$$Z_0 = \sqrt{\frac{L}{C}} \tag{A.5}$$

Figure A.15 shows the symbol for a transmission line. The symbol resembles a *coaxial cable* with an inner signal conductor and an outer grounded conductor like that used in television cable wiring.

The key to understanding the behavior of transmission lines is to visualize the wave of voltage propagating along the line at the speed of light. When the wave reaches the end of the line, it may be absorbed or reflected, depending on the termination or load at the end. Reflections travel back along the line, adding to the voltage already on the line. Terminations are classified as matched, open, short, or mismatched. The following subsections explore how a wave propagates along the line and what happens to the wave when it reaches the termination.

A.8.1 Matched Termination

Figure A.16 shows a transmission line of length l with a *matched termination*, which means that the load impedance, Z_L, is equal to the characteristic impedance, Z_0. The transmission line has a characteristic impedance of 50 Ω. One end of the line is connected to a voltage

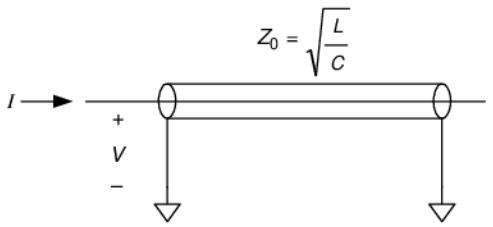


Figure A.15 Transmission line symbol

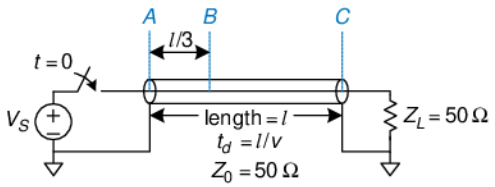


Figure A.16 Transmission line with matched termination

source through a switch that closes at time $t = 0$. The other end is connected to the 50 Ω matched load. This section analyzes the voltages and currents at points A, B, and C—at the beginning of the line, one-third of the length along the line, and at the end of the line, respectively.

Figure A.17 shows the voltages at points A, B, and C over time. Initially, there is no voltage or current flowing in the transmission line, because the switch is open. At time $t = 0$, the switch closes, and the voltage source launches a wave with voltage $V = V_S$ along the line. Because the characteristic impedance is Z_0, the wave has current $I = V_S/Z_0$. The voltage reaches the beginning of the line (point A) immediately, as shown in Figure A.17(a). The wave propagates along the line at the speed of light. At time $t_d/3$, the wave reaches point B. The voltage at this point abruptly rises from 0 to V_S, as shown in Figure A.17(b). At time t_d, the *incident wave* reaches point C at the end of the line, and the voltage rises there too. All of the current, I, flows into the resistor, Z_L, producing a voltage across the resistor of $Z_L I = Z_L(V_S/Z_0) = V_S$ because $Z_L = Z_0$. This voltage is consistent with the wave flowing along the transmission line. Thus, the wave is *absorbed* by the load impedance, and the transmission line reaches its *steady state*.

In steady state, the transmission line behaves like an ideal equipotential wire because it is, after all, just a wire. The voltage at all points along the line must be identical. Figure A.18 shows the steady-state equivalent model of the circuit in Figure A.16. The voltage is V_S everywhere along the wire.

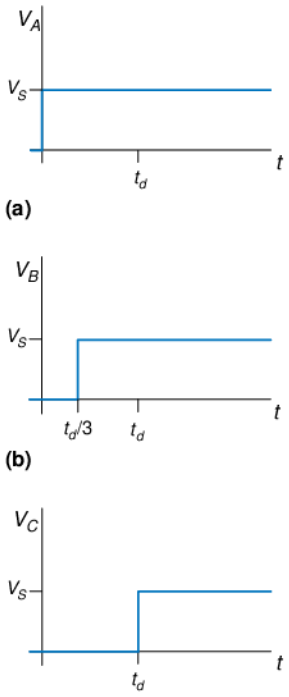


Figure A.17 Voltage waveforms for Figure A.16 at points *A*, *B*, and *C*

Example A.2 TRANSMISSION LINE WITH MATCHED SOURCE AND LOAD TERMINATIONS

Figure A.19 shows a transmission line with matched source and load impedances Z_S and Z_L. Plot the voltage at nodes A, B, and C versus time. When does the system reach steady-state, and what is the equivalent circuit at steady-state?

SOLUTION: When the voltage source has a source impedance, Z_S, in series with the transmission line, part of the voltage drops across Z_S, and the remainder

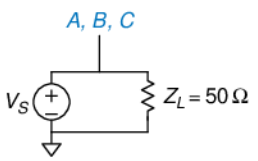


Figure A.18 Equivalent circuit of Figure A.16 at steady state

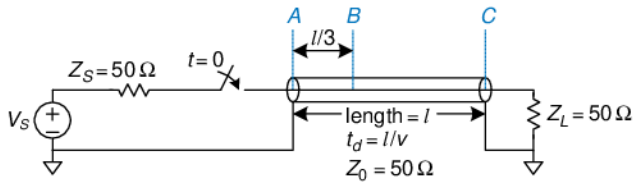


Figure A.19 Transmission line with matched source and load impedances

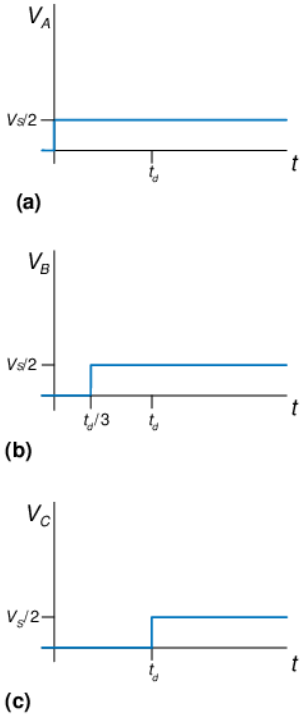


Figure A.20 Voltage waveforms for Figure A.19 at points *A*, *B*, and *C*

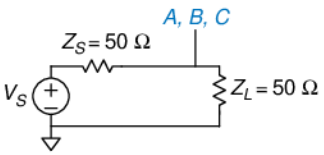


Figure A.21 Equivalent circuit of Figure A.19 at steady state

propagates down the transmission line. At first, the transmission line behaves as an impedance Z_0, because the load at the end of the line cannot possibly influence the behavior of the line until a speed of light delay has elapsed. Hence, by the *voltage divider equation*, the incident voltage flowing down the line is

$$V = V_S\left(\frac{Z_0}{Z_0 + Z_S}\right) = \frac{V_S}{2} \tag{A.6}$$

Thus, at $t = 0$, a wave of voltage, $V = \frac{V_S}{2}$, is sent down the line from point *A*. Again, the signal reaches point *B* at time $t_d/3$ and point *C* at t_d, as shown in Figure A.20. All of the current is absorbed by the load impedance Z_L, so the circuit enters steady-state at $t = t_d$. In steady-state, the entire line is at $V_S/2$, just as the steady-state equivalent circuit in Figure A.21 would predict.

A.8.2 Open Termination

When the load impedance is not equal to Z_0, the termination cannot absorb all of the current, and some of the wave must be reflected. Figure A.22 shows a transmission line with an open load termination. No current can flow through an open termination, so the current at point C must always be 0.

The voltage on the line is initially zero. At $t = 0$, the switch closes and a wave of voltage, $V = V_S\frac{Z_0}{Z_0 + Z_S} = \frac{V_S}{2}$, begins propagating down the line. Notice that this initial wave is the same as that of Example A.2 and is independent of the termination, because the load at the end of the line cannot influence the behavior at the beginning until at least t_d has elapsed. This wave reaches point *B* at $t_d/3$ and point C at t_d as shown in Figure A.23.

When the incident wave reaches point C, it cannot continue forward because the wire is open. It must instead reflect back toward the source. The reflected wave also has voltage $V = \frac{V_S}{2}$, because the open termination reflects the entire wave.

The voltage at any point is the sum of the incident and reflected waves. At time $t = t_d$, the voltage at point C is $V = \frac{V_S}{2} + \frac{V_S}{2} = V_S$. The reflected wave reaches point *B* at $5t_d/3$ and point *A* at $2t_d$. When it reaches point *A*, the wave is absorbed by the source termination

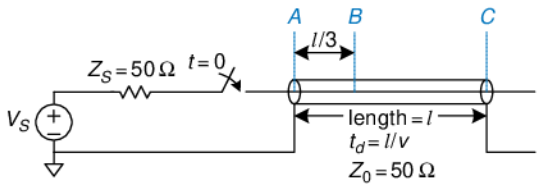


Figure A.22 Transmission line with open load termination

impedance that matches the characteristic impedance of the line. Thus, the system reaches steady state at time $t = 2t_d$, and the transmission line becomes equivalent to an equipotential wire with voltage V_S and current $I = 0$.

A.8.3 Short Termination

Figure A.24 shows a transmission line terminated with a short circuit to ground. Thus, the voltage at point C must always be 0.

As in the previous examples, the voltages on the line are initially 0. When the switch closes, a wave of voltage, $V = \frac{V_S}{2}$, begins propagating down the line (Figure A.25). When it reaches the end of the line, it must reflect with opposite polarity. The reflected wave, with voltage $V = \frac{-V_S}{2}$, adds to the incident wave, ensuring that the voltage at point C remains 0. The reflected wave reaches the source at time $t = 2t_d$ and is absorbed by the source impedance. At this point, the system reaches steady state, and the transmission line is equivalent to an equipotential wire with voltage $V = 0$.

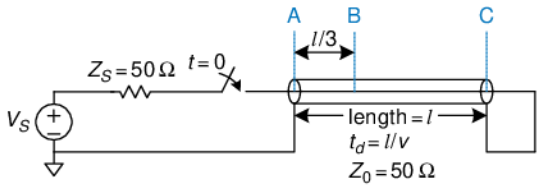
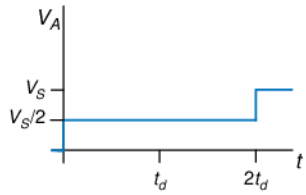
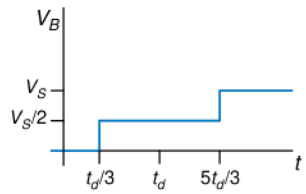


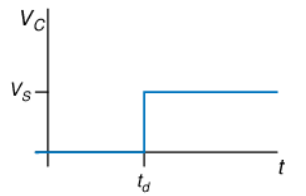
Figure A.24 Transmission line with short termination



(a)



(b)



(c)

Figure A.23 Voltage waveforms for Figure A.22 at points *A*, *B*, and *C*

A.8.4 Mismatched Termination

The termination impedance is said to be *mismatched* when it does not equal the characteristic impedance of the line. In general, when an incident wave reaches a mismatched termination, part of the wave is absorbed and part is reflected. The reflection coefficient, k_r, indicates the fraction of the incident wave (V_i) that is reflected: $V_r = k_r V_i$.

Section A.8.8 derives the reflection coefficient using conservation of current arguments. It shows that, when an incident wave flowing along a

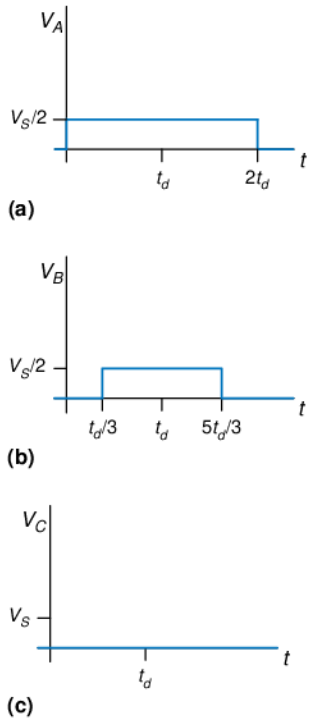


Figure A.25 Voltage waveforms for Figure A.24 at points *A*, *B*, and *C*

transmission line of characteristic impedance Z_0 reaches a termination impedance, Z_T, at the end of the line, the reflection coefficient is

$$k_r = \frac{Z_T - Z_0}{Z_T + Z_0} \tag{A.7}$$

Note a few special cases. If the termination is an open circuit ($Z_T = \infty$), $k_r = 1$, because the incident wave is entirely reflected (so the current out the end of the line remains zero). If the termination is a short circuit ($Z_T = 0$), $k_r = -1$, because the incident wave is reflected with negative polarity (so the voltage at the end of the line remains zero). If the termination is a matched load ($Z_T = Z_0$), $k_r = 0$, because the incident wave is absorbed.

Figure A.26 illustrates reflections in a transmission line with a *mismatched load termination* of 75 Ω. $Z_T = Z_L = 75\ \Omega$, and $Z_0 = 50\ \Omega$, so $k_r = 1/5$. As in previous examples, the voltage on the line is initially 0. When the switch closes, a wave of voltage, $V = \frac{V_S}{2}$, propagates down the line, reaching the end at $t = t_d$. When the incident wave reaches the termination at the end of the line, one fifth of the wave is reflected, and the remaining four fifths flows into the load impedance. Thus, the reflected wave has a voltage $V = \frac{V_S}{2} \times \frac{1}{5} = \frac{V_S}{10}$. The total voltage at point C is the sum of the incoming and reflected voltages, $V_C = \frac{V_S}{2} + \frac{V_S}{10} = \frac{3V_S}{5}$. At $t = 2t_d$, the reflected wave reaches point A, where it is absorbed by the matched 50 Ω termination, Z_S. Figure A.27 plots the voltages and currents along the line. Again, note that, in steady state (in this case at time $t > 2t_d$), the transmission line is equivalent to an equipotential wire, as shown in Figure A.28. At steady-state, the system acts like a voltage divider, so

$$V_A = V_B = V_C = V_S\left(\frac{Z_L}{Z_L + Z_S}\right) = V_S\left(\frac{75\ \Omega}{75\ \Omega + 50\ \Omega}\right) = \frac{3V_S}{5}$$

Reflections can occur at both ends of the transmission line. Figure A.29 shows a transmission line with a source impedance, Z_S, of 450 Ω and an open termination at the load. The reflection coefficients at the load and source, k_{rL} and k_{rS}, are 4/5 and 1, respectively. In this case, waves reflect off both ends of the transmission line until a steady state is reached.

The *bounce diagram*, shown in Figure A.30, helps visualize reflections off both ends of the transmission line. The horizontal axis represents

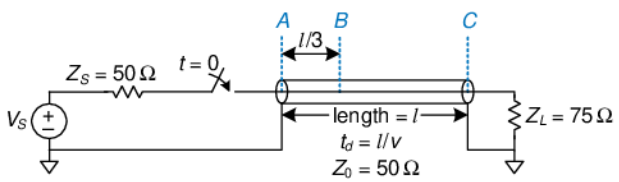


Figure A.26 Transmission line with mismatched termination

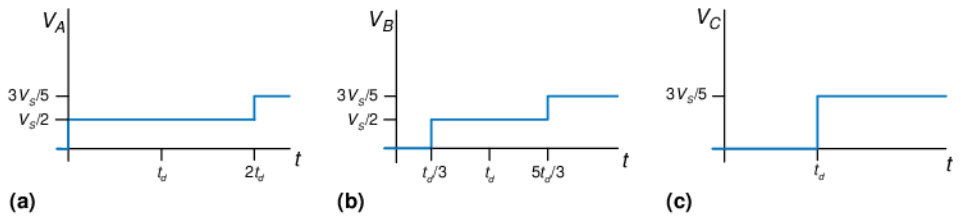


Figure A.27 Voltage waveforms for Figure A.26 at points *A*, *B*, and *C*

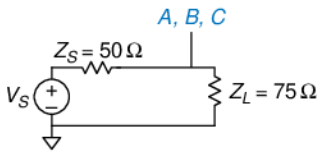


Figure A.28 Equivalent circuit of Figure A.26 at steady-state

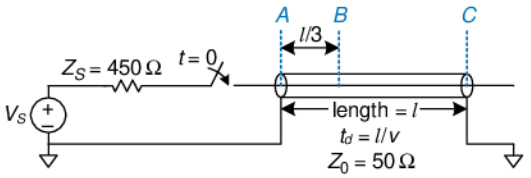


Figure A.29 Transmission line with mismatched source and load terminations

distance along the transmission line, and the vertical axis represents time, increasing downward. The two sides of the bounce diagram represent the source and load ends of the transmission line, points *A* and C. The incoming and reflected signal waves are drawn as diagonal lines between points *A* and C. At time $t = 0$, the source impedance and transmission line behave as a voltage divider, launching a voltage wave of $\frac{V_S}{10}$ from point *A* toward point C. At time $t = t_d$, the signal reaches point C and is completely reflected ($k_{rL} = 1$). At time $t = 2t_d$, the reflected wave of $\frac{V_S}{10}$ reaches point *A* and is reflected with a reflection coefficient, $k_{rS} = 4/5$, to produce a wave of $\frac{2V_S}{25}$ traveling toward point C, and so forth.

The voltage at a given time at any point on the transmission line is the sum of all the incident and reflected waves. Thus, at time $t = 1.1t_d$, the voltage at point C is $\frac{V_S}{10} + \frac{V_S}{10} = \frac{V_S}{5}$. At time $t = 3.1t_d$, the voltage at point C is $\frac{V_S}{10} + \frac{V_S}{10} + \frac{2V_S}{25} + \frac{2V_S}{25} = \frac{9V_S}{25}$, and so forth. Figure A.31 plots the voltages against time. As t approaches infinity, the voltages approach steady-state with $V_A = V_B = V_C = V_S$.

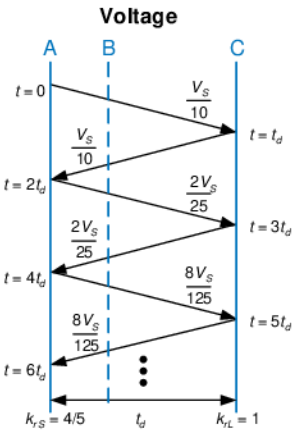


Figure A.30 Bounce diagram for Figure A.29

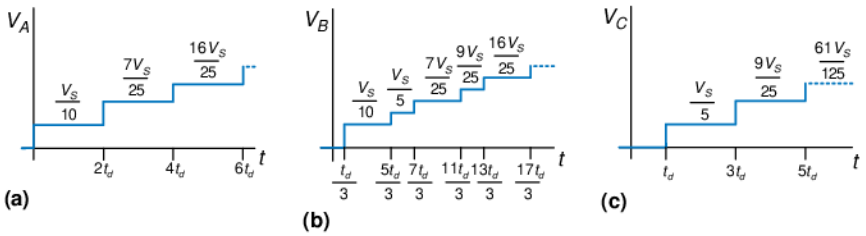


Figure A.31 Voltage and current waveforms for Figure A.29

A.8.5 When to Use Transmission Line Models

Transmission line models for wires are needed whenever the wire delay, t_d, is longer than a fraction (e.g., 20%) of the edge rates (rise or fall times) of a signal. If the wire delay is shorter, it has an insignificant effect on the propagation delay of the signal, and the reflections dissipate while the signal is transitioning. If the wire delay is longer, it must be considered in order to accurately predict the propagation delay and waveform of the signal. In particular, reflections may distort the digital characteristic of a waveform, resulting in incorrect logic operations.

Recall that signals travel on a PCB at about 15 cm/ns. For TTL logic, with edge rates of 10 ns, wires must be modeled as transmission lines only if they are longer than 30 cm (10 ns × 15 cm/ns × 20%). PCB traces are usually less than 30 cm, so most traces can be modeled as ideal equipotential wires. In contrast, many modern chips have edge rates of 2 ns or less, so traces longer than about 6 cm (about 2.5 inches) must be modeled as transmission lines. Clearly, use of edge rates that are crisper than necessary just causes difficulties for the designer.

Breadboards lack a ground plane, so the electromagnetic fields of each signal are nonuniform and difficult to model. Moreover, the fields interact with other signals. This can cause strange reflections and crosstalk between signals. Thus, breadboards are unreliable above a few megahertz.

In contrast, PCBs have good transmission lines with consistent characteristic impedance and velocity along the entire line. As long as they are terminated with a source or load impedance that is matched to the impedance of the line, PCB traces do not suffer from reflections.

A.8.6 Proper Transmission Line Terminations

There are two common ways to properly terminate a transmission line, shown in Figure A.32. In *parallel termination* (Figure A.32(a)), the driver has a low impedance ($Z_S \approx 0$). A load resistor (Z_L) with impedance Z_0 is placed in parallel with the load (between the input of the receiver gate and

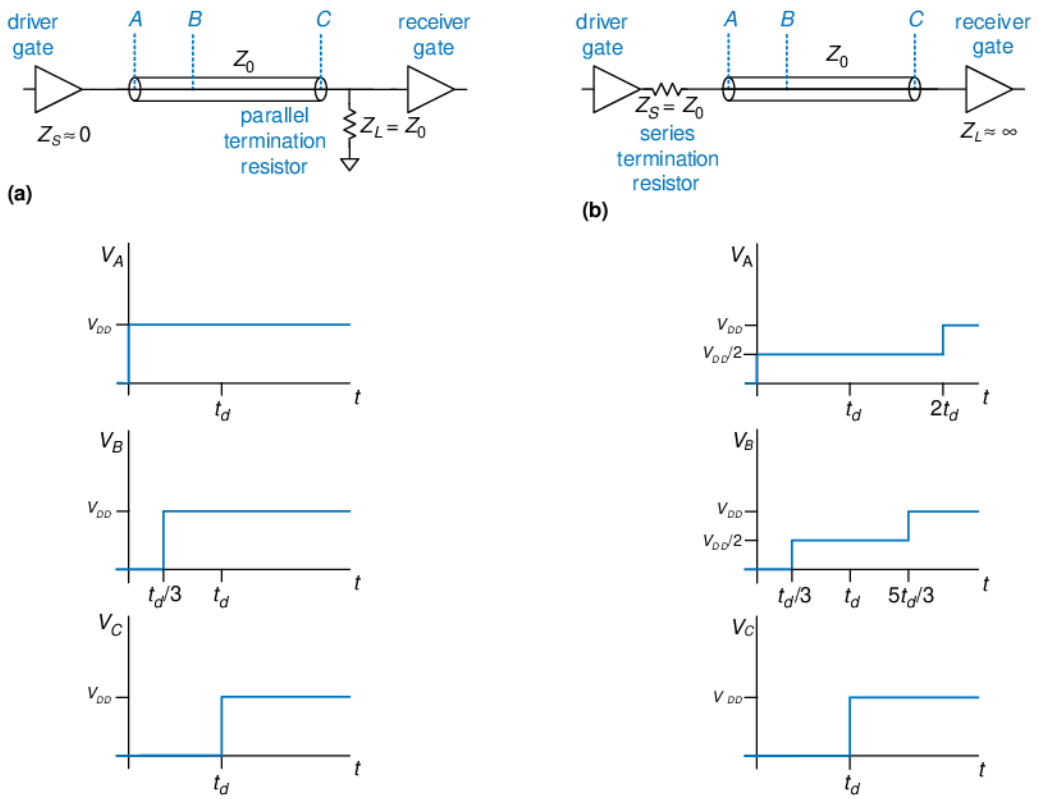


Figure A.32 Termination schemes: (a) parallel, (b) series

ground). When the driver switches from 0 to V_{DD}, it sends a wave with voltage V_{DD} down the line. The wave is absorbed by the matched load termination, and no reflections take place. In *series termination* (Figure A.32(b)), a source resistor (Z_S) is placed in series with the driver to raise the source impedance to Z_0. The load has a high impedance ($Z_L \approx \infty$). When the driver switches, it sends a wave with voltage $V_{DD}/2$ down the line. The wave reflects at the open circuit load and returns, bringing the voltage on the line up to V_{DD}. The wave is absorbed at the source termination. Both schemes are similar in that the voltage at the receiver transitions from 0 to V_{DD} at $t = t_d$, just as one would desire. They differ in power consumption and in the waveforms that appear elsewhere along the line. Parallel termination dissipates power continuously through the load resistor when the line is at a high voltage. Series termination dissipates no DC power, because the load is an open circuit. However, in series terminated lines, points near the middle of the transmission line initially see a voltage of $V_{DD}/2$, until the reflection returns. If other gates are attached to the

middle of the line, they will momentarily see an illegal logic level. Therefore, series termination works best for *point-to-point* communication with a single driver and a single receiver. Parallel termination is better for a *bus* with multiple receivers, because receivers at the middle of the line never see an illegal logic level.

A.8.7 Derivation of Z_0*

Z_0 is the ratio of voltage to current in a wave propagating along a transmission line. This section derives Z_0; it assumes some previous knowledge of resistor-inductor-capacitor (RLC) circuit analysis.

Imagine applying a step voltage to the input of a semi-infinite transmission line (so that there are no reflections). Figure A.33 shows the semi-infinite line and a model of a segment of the line of length *dx*. *R*, *L*, and *C*, are the values of resistance, inductance, and capacitance per unit length. Figure A.33(b) shows the transmission line model with a resistive component, *R*. This is called a *lossy* transmission line model, because energy is dissipated, or lost, in the resistance of the wire. However, this loss is often negligible, and we can simplify analysis by ignoring the resistive component and treating the transmission line as an *ideal* transmission line, as shown in Figure A.33(c).

Voltage and current are functions of time and space throughout the transmission line, as given by Equations A.8 and A.9.

$$\frac{\partial}{\partial x}V(x,t) = L\frac{\partial}{\partial t}I(x,t) \tag{A.8}$$

$$\frac{\partial}{\partial x}I(x,t) = C\frac{\partial}{\partial t}V(x,t) \tag{A.9}$$

Taking the space derivative of Equation A.8 and the time derivative of Equation A.9 and substituting gives Equation A.10, the *wave equation*.

$$\frac{\partial^2}{\partial x^2}V(x,t) = LC\frac{\partial^2}{\partial t^2}I(x,t) \tag{A.10}$$

Z_0 is the ratio of voltage to current in the transmission line, as illustrated in Figure A.34(a). Z_0 must be independent of the length of the line, because the behavior of the wave cannot depend on things at a distance. Because it is independent of length, the impedance must still equal Z_0 after the addition of a small amount of transmission line, *dx*, as shown in Figure A.34(b).

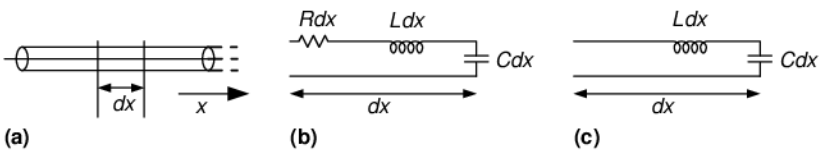


Figure A.33 Transmission line models: (a) semi-infinite cable, (b) lossy, (c) ideal

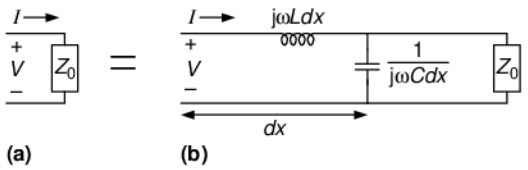


Figure A.34 Transmission line model: (a) for entire line and (b) with additional length, *dx*

Using the impedances of an inductor and a capacitor, we rewrite the relationship of Figure A.34 in equation form:

$$Z_0 = j\omega L dx + [Z_0 \| (1/(j\omega C dx))] \tag{A.11}$$

Rearranging, we get

$$Z_0^2(j\omega C) - j\omega L + \omega^2 Z_0 L C dx = 0 \tag{A.12}$$

Taking the limit as dx approaches 0, the last term vanishes and we find that

$$Z_0 = \sqrt{\frac{L}{C}} \tag{A.13}$$

A.8.8 Derivation of the Reflection Coefficient*

The reflection coefficient, k_r, is derived using conservation of current. Figure A.35 shows a transmission line with characteristic impedance, Z_0, and load impedance, Z_L. Imagine an incident wave of voltage V_i and current I_i. When the wave reaches the termination, some current, I_L, flows through the load impedance, causing a voltage drop, V_L. The remainder of the current reflects back down the line in a wave of voltage, V_r, and current, I_r. Z_0 is the ratio of voltage to current in waves propagating along the line, so $\frac{V_i}{I_i} = \frac{V_r}{I_r} = Z_0$.

The voltage on the line is the sum of the voltages of the incident and reflected waves. The current flowing in the positive direction on the line is the difference between the currents of the incident and reflected waves.

$$V_L = V_i + V_r \tag{A.14}$$

$$I_L = I_i - I_r \tag{A.15}$$

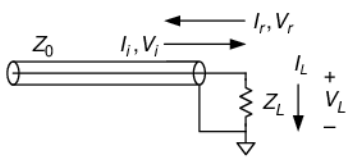


Figure A.35 Transmission line showing incoming, reflected, and load voltages and currents

Using Ohm's law and substituting for I_L, I_i, and I_r in Equation A.15, we get

$$\frac{V_i + V_r}{Z_L} = \frac{V_i}{Z_0} - \frac{V_r}{Z_0} \tag{A.16}$$

Rearranging, we solve for the reflection coefficient, k_r:

$$\frac{V_r}{V_i} = \frac{Z_L - Z_0}{Z_L + Z_0} = k_r \tag{A.17}$$

A.8.9 Putting It All Together

Transmission lines model the fact that signals take time to propagate down long wires because the speed of light is finite. An ideal transmission line has uniform inductance, L, and capacitance, C, per unit length and zero resistance. The transmission line is characterized by its characteristic impedance, Z_0, and delay, t_d, which can be derived from the inductance, capacitance, and wire length. The transmission line has significant delay and noise effects on signals whose rise/fall times are less than about $5t_d$. This means that, for systems with 2 ns rise/fall times, PCB traces longer than about 6 cm must be analyzed as transmission lines to accurately understand their behavior.

A digital system consisting of a gate driving a long wire attached to the input of a second gate can be modeled with a transmission line as shown in Figure A.36. The voltage source, source impedance (Z_S), and switch model the first gate switching from 0 to 1 at time 0. The driver gate cannot supply infinite current; this is modeled by Z_S. Z_S is usually small for a logic gate, but a designer may choose to add a resistor in series with the gate to raise Z_S and match the impedance of the line. The input to the second gate is modeled as Z_L. CMOS circuits usually have little input current, so Z_L may be close to infinity. The designer may also choose to add a resistor in parallel with the second gate, between the gate input and ground, so that Z_L matches the impedance of the line.

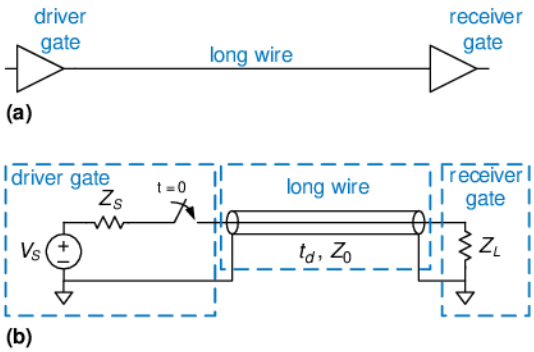


Figure A.36 Digital system modeled with transmission line

When the first gate switches, a wave of voltage is driven onto the transmission line. The source impedance and transmission line form a voltage divider, so the voltage of the incident wave is

$$V_i = V_S \frac{Z_0}{Z_0 + Z_S} \tag{A.18}$$

At time t_d, the wave reaches the end of the line. Part is absorbed by the load impedance, and part is reflected. The reflection coefficient, k_r, indicates the portion that is reflected: $k_r = V_r/V_i$, where V_r is the voltage of the reflected wave and V_i is the voltage of the incident wave.

$$k_r = \frac{Z_L - Z_0}{Z_L + Z_0} \tag{A.19}$$

The reflected wave adds to the voltage already on the line. It reaches the source at time $2t_d$, where part is absorbed and part is again reflected. The reflections continue back and forth, and the voltage on the line eventually approaches the value that would be expected if the line were a simple equipotential wire.

A.9 ECONOMICS

Although digital design is so much fun that some of us would do it for free, most designers and companies intend to make money. Therefore, economic considerations are a major factor in design decisions.

The cost of a digital system can be divided into *nonrecurring engineering costs* (*NRE*), and *recurring costs*. NRE accounts for the cost of designing the system. It includes the salaries of the design team, computer and software costs, and the costs of producing the first working unit. The fully loaded cost of a designer in the United States in 2006 (including salary, health insurance, retirement plan, and a computer with design tools) is roughly $200,000 per year, so design costs can be significant. Recurring costs are the cost of each additional unit; this includes components, manufacturing, marketing, technical support, and shipping.

The sales price must cover not only the cost of the system but also other costs such as office rental, taxes, and salaries of staff who do not directly contribute to the design (such as the janitor and the CEO). After all of these expenses, the company should still make a profit.

Example A.3 BEN TRIES TO MAKE SOME MONEY

Ben Bitdiddle has designed a crafty circuit for counting raindrops. He decides to sell the device and try to make some money, but he needs help deciding what implementation to use. He decides to use either an FPGA or an ASIC. The

development kit to design and test the FPGA costs $1500. Each FPGA costs $17. The ASIC costs $600,000 for a mask set and $4 per chip.

Regardless of what chip implementation he chooses, Ben needs to mount the packaged chip on a printed circuit board (PCB), which will cost him $1.50 per board. He thinks he can sell 1000 devices per month. Ben has coerced a team of bright undergraduates into designing the chip for their senior project, so it doesn't cost him anything to design.

If the sales price has to be twice the cost (100% profit margin), and the product life is 2 years, which implementation is the better choice?

SOLUTION: Ben figures out the total cost for each implementation over 2 years, as shown in Table A.4. Over 2 years, Ben plans on selling 24,000 devices, and the total cost is given in Table A.4 for each option. If the product life is only two years, the FPGA option is clearly superior. The per-unit cost is $445,500/24,000 = $18.56, and the sales price is $37.13 per unit to give a 100% profit margin. The ASIC option would have cost $732,000/24,000 = $30.50 and would have sold for $61 per unit.

Table A.4 ASIC vs FPGA costs

Cost	ASIC	FPGA
NRE	$600,000	$1500
chip	$4	$17
PCB	$1.50	$1.50
TOTAL	$600,000 + (24,000 × $5.50) = **$732,000**	$1500 + (24,000 × $18.50) = **$445,500**
per unit	$30.50	$18.56

Example A.4 BEN GETS GREEDY

After seeing the marketing ads for his product, Ben thinks he can sell even more chips per month than originally expected. If he were to choose the ASIC option, how many devices per month would he have to sell to make the ASIC option more profitable than the FPGA option?

SOLUTION: Ben solves for the minimum number of units, N, that he would need to sell in 2 years:

$$\$600{,}000 + (N \times \$5.50) = \$1500 + (N \times \$18.50)$$

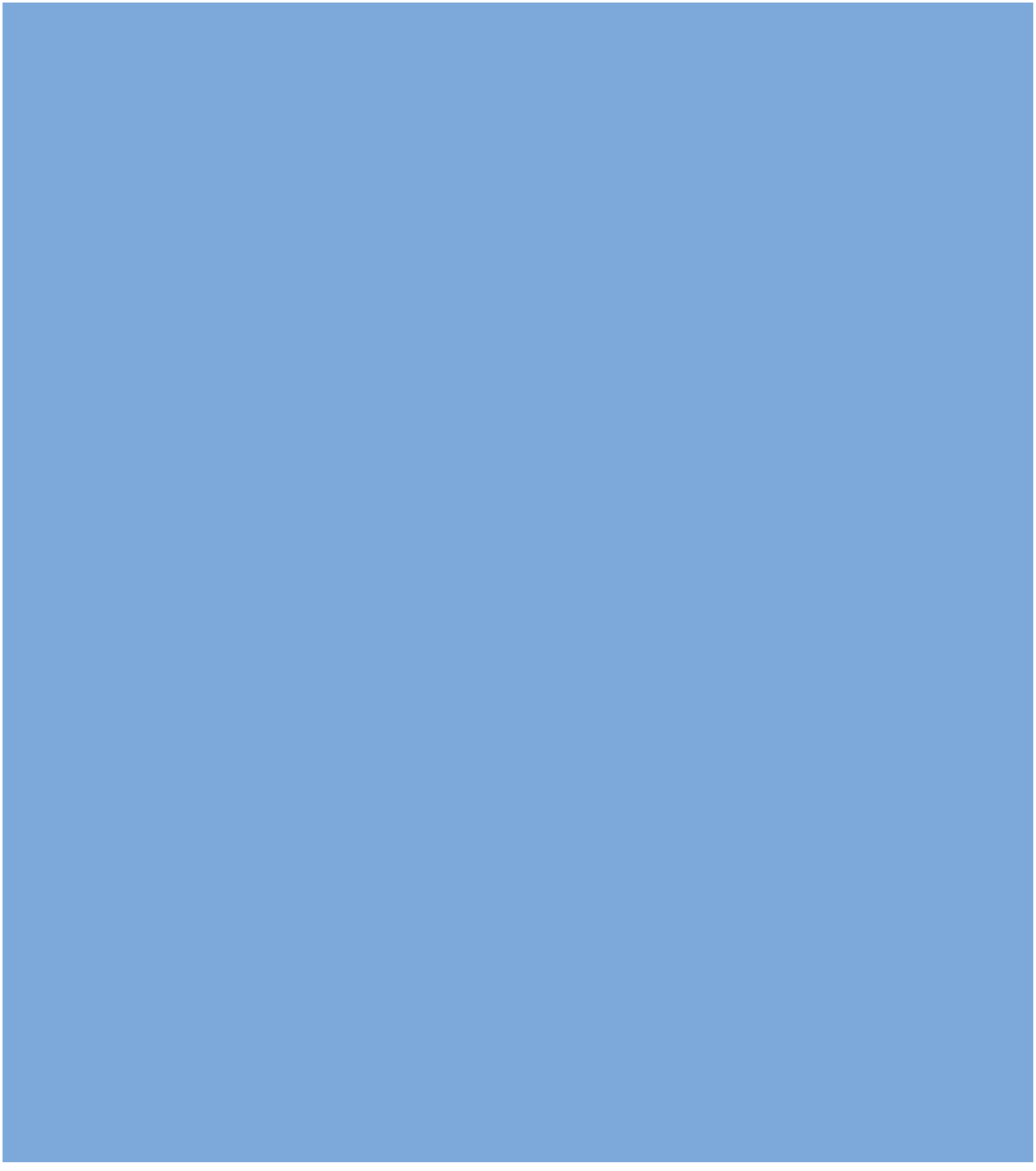
Solving the equation gives $N = 46{,}039$ units, or 1919 units per month. He would need to almost double his monthly sales to benefit from the ASIC solution.

Example A.5 BEN GETS LESS GREEDY

Ben realizes that his eyes have gotten too big for his stomach, and he doesn't think he can sell more than 1000 devices per month. But he does think the product life can be longer than 2 years. At a sales volume of 1000 devices per month, how long would the product life have to be to make the ASIC option worthwhile?

SOLUTION: If Ben sells more than 46,039 units in total, the ASIC option is the best choice. So, Ben would need to sell at a volume of 1000 per month for at least 47 months (rounding up), which is almost 4 years. By then, his product is likely to be obsolete.

Chips are usually purchased from a distributor rather than directly from the manufacturer (unless you are ordering tens of thousands of units). Digikey (`www.digikey.com`) is a leading distributor that sells a wide variety of electronics. Jameco (`www.jameco.com`) and All Electronics (`www.allelectronics.com`) have eclectic catalogs that are competitively priced and well suited to hobbyists.



MIPS Instructions

This appendix summarizes MIPS instructions used in this book. Tables B.1–B.3 define the `opcode` and `funct` fields for each instruction, along with a short description of what the instruction does. The following notations are used:

- `[reg]`: contents of the register
- `imm`: 16-bit immediate field of the I-type instruction
- `addr`: 26-bit address field of the J-type instruction
- `SignImm`: sign-extended immediate
  `= {{16{imm[15]}}, imm}`
- `ZeroImm`: zero-extended immediate
  `= {16'b0, imm}`
- `Address`: `[rs] + SignImm`
- `[Address]`: contents of memory location `Address`
- `BTA`: branch target address[1]
  `= PC + 4 + (SignImm << 2)`
- `JTA`: jump target address
  `= {(PC + 4)[31:28], addr, 2'b0}`

[1] The SPIM simulator has no branch delay slot, so BTA is PC + (SignImm << 2). Thus, if you use the SPIM assembler to create machine code for a real MIPS processor, you must decrement the immediate field by 1 to compensate.

Table B.1 Instructions, sorted by opcode

Opcode	Name	Description	Operation
000000 (0)	R-type	all R-type instructions	see Table B.2
000001 (1) (rt = 0/1)	bltz/bgez	branch less than zero/ branch greater than or equal to zero	if ([rs] < 0) PC = BTA/ if ([rs] ≥ 0) PC = BTA
000010 (2)	j	jump	PC = JTA
000011 (3)	jal	jump and link	$ra = PC+4, PC = JTA
000100 (4)	beq	branch if equal	if ([rs]==[rt]) PC = BTA
000101 (5)	bne	branch if not equal	if ([rs]!=[rt]) PC = BTA
000110 (6)	blez	branch if less than or equal to zero	if ([rs] ≤ 0) PC = BTA
000111 (7)	bgtz	branch if greater than zero	if ([rs] > 0) PC = BTA
001000 (8)	addi	add immediate	[rt] = [rs] + SignImm
001001 (9)	addiu	add immediate unsigned	[rt] = [rs] + SignImm
001010 (10)	slti	set less than immediate	[rs] < SignImm ? [rt]=1 : [rt]=0
001011 (11)	sltiu	set less than immediate unsigned	[rs] < SignImm ? [rt]=1 : [rt]=0
001100 (12)	andi	and immediate	[rt] = [rs] & ZeroImm
001101 (13)	ori	or immediate	[rt] = [rs] | ZeroImm
001110 (14)	xori	xor immediate	[rt] = [rs] ^ ZeroImm
001111 (15)	lui	load upper immediate	[rt] = {Imm, 16'b0}
010000 (16) (rs = 0/4)	mfc0, mtc0	move from/to coprocessor 0	[rt] = [rd]/[rd] = [rt] (rd is in coprocessor 0)
010001 (17)	F-type	fop = 16/17: F-type instructions	see Table B.3
010001 (17) (rt = 0/1)	bc1f/bc1t	fop = 8: branch if fpcond is FALSE/TRUE	if (fpcond == 0) PC = BTA/ if (fpcond == 1) PC = BTA
100000 (32)	lb	load byte	[rt] = SignExt ([Address]$_{7:0}$)
100001 (33)	lh	load halfword	[rt] = SignExt ([Address]$_{15:0}$)
100011 (35)	lw	load word	[rt] = [Address]
100100 (36)	lbu	load byte unsigned	[rt] = ZeroExt ([Address]$_{7:0}$)
100101 (37)	lhu	load halfword unsigned	[rt] = ZeroExt ([Address]$_{15:0}$)
101000 (40)	sb	store byte	[Address]$_{7:0}$ = [rt]$_{7:0}$

(*continued*)

Table B.1 Instructions, sorted by opcode—Cont'd

Opcode	Name	Description	Operation
101001 (41)	sh	store halfword	$[Address]_{15:0} = [rt]_{15:0}$
101011 (43)	sw	store word	[Address] = [rt]
110001 (49)	lwc1	load word to FP coprocessor 1	[ft] = [Address]
111001 (56)	swc1	store word to FP coprocessor 1	[Address] = [ft]

Table B.2 R-type instructions, sorted by funct field

Funct	Name	Description	Operation
000000 (0)	sll	shift left logical	[rd] = [rt] << shamt
000010 (2)	srl	shift right logical	[rd] = [rt] >> shamt
000011 (3)	sra	shift right arithmetic	[rd] = [rt] >>> shamt
000100 (4)	sllv	shift left logical variable	[rd] = [rt] << $[rs]_{4:0}$ assembly: sllv rd, rt, rs
000110 (6)	srlv	shift right logical variable	[rd] = [rt] >> $[rs]_{4:0}$ assembly: srlv rd, rt, rs
000111 (7)	srav	shift right arithmetic variable	[rd] = [rt] >>> $[rs]_{4:0}$ assembly: srav rd, rt, rs
001000 (8)	jr	jump register	PC = [rs]
001001 (9)	jalr	jump and link register	$ra = PC + 4, PC = [rs]
001100 (12)	syscall	system call	system call exception
001101 (13)	break	break	break exception
010000 (16)	mfhi	move from hi	[rd] = [hi]
010001 (17)	mthi	move to hi	[hi] = [rs]
010010 (18)	mflo	move from lo	[rd] = [lo]
010011 (19)	mtlo	move to lo	[lo] = [rs]
011000 (24)	mult	multiply	{[hi], [lo]} = [rs] × [rt]
011001 (25)	multu	multiply unsigned	{[hi], [lo]} = [rs] × [rt]
011010 (26)	div	divide	[lo] = [rs]/[rt], [hi] = [rs]%[rt]

(continued)

Table B.2 R-type instructions, sorted by funct field—Cont'd

Funct	Name	Description	Operation
011011 (27)	divu	divide unsigned	[lo] = [rs]/[rt], [hi] = [rs]%[rt]
100000 (32)	add	add	[rd] = [rs] + [rt]
100001 (33)	addu	add unsigned	[rd] = [rs] + [rt]
100010 (34)	sub	subtract	[rd] = [rs] − [rt]
100011 (35)	subu	subtract unsigned	[rd] = [rs] − [rt]
100100 (36)	and	and	[rd] = [rs] & [rt]
100101 (37)	or	or	[rd] = [rs] | [rt]
100110 (38)	xor	xor	[rd] = [rs] ^ [rt]
100111 (39)	nor	nor	[rd] = ~([rs] | [rt])
101010 (42)	slt	set less than	[rs] < [rt] ? [rd] = 1 : [rd] = 0
101011 (43)	sltu	set less than unsigned	[rs] < [rt] ? [rd] = 1 : [rd] = 0

Table B.3 F-type instructions (fop = 16/17)

Funct	Name	Description	Operation
000000 (0)	add.s/add.d	FP add	[fd] = [fs] + [ft]
000001 (1)	sub.s/sub.d	FP subtract	[fd] = [fs] − [ft]
000010 (2)	mul.s/mul.d	FP multiply	[fd] = [fs] * [ft]
000011 (3)	div.s/div.d	FP divide	[fd] = [fs]/[ft]
000101 (5)	abs.s/abs.d	FP absolute value	[fd] = ([fs] < 0) ? [−fs] : [fs]
000111 (7)	neg.s/neg.d	FP negation	[fd] = [−fs]
111010 (58)	c.seq.s/c.seq.d	FP equality comparison	fpcond = ([fs] == [ft])
111100 (60)	c.lt.s/c.lt.d	FP less than comparison	fpcond = ([fs] < [ft])
111110 (62)	c.le.s/c.le.d	FP less than or equal comparison	fpcond = ([fs] ≤ [ft])

Further Reading

Berlin L., *The Man Behind the Microchip: Robert Noyce and the Invention of Silicon Valley*, Oxford University Press, 2005.

The fascinating biography of Robert Noyce, an inventor of the microchip and founder of Fairchild and Intel. For anyone thinking of working in Silicon Valley, this book gives insights into the culture of the region, a culture influenced more heavily by Noyce than any other individual.

Colwell R., *The Pentium Chronicles: The People, Passion, and Politics Behind Intel's Landmark Chips*, Wiley, 2005.

An insider's tale of the development of several generations of Intel's Pentium chips, told by one of the leaders of the project. For those considering a career in the field, this book offers views into the managment of huge design projects and a behind-the-scenes look at one of the most significant commercial microprocessor lines.

Ercegovac M., and Lang T., *Digital Arithmetic*, Morgan Kaufmann, 2003.

The most complete text on computer arithmetic systems. An excellent resource for building high-quality arithmetic units for computers.

Hennessy J., and Patterson D., *Computer Architecture: A Quantitative Approach, 4th ed.*, Morgan Kaufmann, 2006.

The authoritative text on advanced computer architecture. If you are intrigued about the inner workings of cutting-edge microprocessors, this is the book for you.

Kidder T., *The Soul of a New Machine*, Back Bay Books, 1981.

A classic story of the design of a computer system. Three decades later, the story is still a page-turner and the insights on project managment and technology still ring true.

Pedroni V., *Circuit Design with VHDL*, MIT Press, 2004.

A reference showing how to design circuits with VHDL.

Thomas D., and Moorby P., *The Verilog Hardware Description Language, 5th ed.*, Kluwer Academic Publishers, 2002.

Ciletti M., *Advanced Digital Design with the Verilog HDL*, Prentice Hall, 2003.

Both excellent references covering Verilog in more detail.

Verilog IEEE Standard (IEEE STD 1364).
The IEEE standard for the Verilog Hardware Description Language; last updated in 2001. Available at *ieeexplore.ieee.org*.

VHDL IEEE Standard (IEEE STD 1076).
The IEEE standard for VHDL; last updated in 2004. Available from IEEE. Available at *ieeexplore.ieee.org*.

Wakerly J., *Digital Design: Principles and Practices, 4th ed.*, Prentice Hall, 2006.
A comprehensive and readable text on digital design, and an excellent reference book.

Weste N., and Harris D., *CMOS VLSI Design, 3rd ed.*, Addison-Wesley, 2005.
Very Large Scale Integration (VLSI) Design is the art and science of building chips containing oodles of transistors. This book, coauthored by one of our favorite writers, spans the field from the beginning through the most advanced techniques used in commercial products.

Index

B

C

D

E

F

G

H

I

J

K

L

M

Q

R

S

T

U

V

W

X

Z