Improving Processor Performance

- Memory latency and performance
- Expanded register sets
- More complex instructions
- Caches
- Pipelining

Read MK 376-382, 467-490, 613-631
Making Computers Faster

- Large and growing speed gap between CPU & DRAM.
  - gate delays now less than a few hundred ps
  - can take 100 ns to retrieve a word from DRAM
  - get better performance by making fewer memory accesses

- Modern processors improve performance by
  - using multiple registers in place of single accumulator
  - using more extensive instruction sets
  - providing additional addressing modes
  - using small, fast cache memories to hold recently used instructions and data
  - overlapping the execution of several instructions (pipelining)
    - simple example, fetch next instruction while executing current one
  - providing multiple ALUs for parallel instruction execution
Logically, a random access memory contains an array of numbered storage locations, called words.

- When read/write is high, data_out is equal to the value stored in word specified by address inputs.
- When read/write is low, the value on data_in replaces the value in word specified by address outputs.
- Separate enable signal also usually provided.

Simplest RAMs are asynchronous — no clock input.

- Synchronous circuits using RAMs must ensure that RAM timing requirements are satisfied to ensure correct operation.
Timing of RAM Operations

- **Read cycle**
  - access time: time from “last” address change until output data is valid

- **Write cycle**
  - $t_1$ is min time from address stable and enable asserted until $r' w'$ is lowered
  - $t_2$ is min time that input data must remain stable before $r' w'$ can be raised
  - $t_3$ is min time that address stays valid after $r' w'$ is raised
  - cycle time is $t_1 + t_2 + t_3$

- Circuits using RAM must ensure timing conditions are met.
Implementing a RAM

4 word RAM with 4 bits per word

data_in
r/w

address
bits select row

address

row decoder

D

D

D

D

0

data_out
Computers and other digital systems generally use large amounts of memory.

Specialized memory cells have been developed to pack more memory in given amount of space.

Typical static RAM uses 6 transistor cell using pair of inverters and pair of pass transistors.

Bit line asserted to read or write the cell.

Complementary data lines used for input and output.

Column drivers enabled when writing.
  - large current capacity allows them to force cell to desired state

\[
\text{data_in} \quad \begin{array}{c}
\text{column drivers} \\
\text{SRAM cell}
\end{array} \quad \text{bit line} \\
\quad \begin{array}{c}
\text{sense amplifier} \\
\text{data_out}
\end{array}
\]
Tri-State Buffers

A tri-state buffer has a data input and a control input.
- when control input is asserted, output equals input
- when control input is not asserted, output is disconnected from input - called high impedance state

Tri-state buffers can be used to build “distributed” multiplexors.
Shared outputs are called buses.
Also allows single wire to be used as data input and output.
Static RAM Array (4x4)
Static RAM Array (8x2)

data_in
address
r/ w

row decoder

column decoder/ demux

column decoder/ mux

data_out
Building Larger RAMs

- Systems often require larger RAMs than can be constructed using a single SRAM component.
- The use of an external decoder and the enable input allows larger RAMs to be constructed.

Alternative design uses 64Kx4 RAM chips.
  - no external decoder needed in this case
Dynamic RAMs

- Dynamic RAMs use a simpler memory cell to enable more bits to be stored in a single chip (4-8x).
  - each storage cell consists of a pass transistor and a capacitor
  - reading contents, destroys value
    - need to write back after reading
  - stored charge leaks from capacitor after 10-100 ms
    - requires periodic refresh of memory contents

- DRAM cells are organized in 2D arrays, much like those for SRAM.
  - single data line rather than pair
  - use sense amplifiers to detect stored charge
  - takes more time (10x) to read values than with SRAM.
Dynamic RAMs

- Large memory chips require lots of address pins.
- Many DRAM chips reduce number of address pins by dividing address into two parts.
  - Row address determines which row in 2D array is selected
  - Column address selects one or more bits in the row
- Column address can be provided after row address without slowing down memory access.
  - So, same address pins can be used to supply both row and column addresses
- Row Address Strobe (RAS), Column Address Strobe (CAS) used to load row and column addresses into on-chip registers
- Refresh circuitry periodically reads each row in memory array and writes it back - often built into chip.
Processor with Multiple Registers

- Use of multiple registers can reduce number of memory accesses.
  - modern processors have at least 32 general purpose registers
- Requires Register File and more general set of control signals.

ALU operations:
- \( R \leq A, A-B, A+B, \ldots \)
Extending Instruction Set

- Arithmetic and logic instructions
  - integer add, subtract, multiply, divide
  - word-wise AND, OR, NOT, EXOR, shift, rotate
  - compare values (\(<\), \(\leq\), \(\geq\))
  - floating point add, subtract, multiply, divide, compare

- Conditional branch instructions
  - sign of register
  - result of comparison
  - occurrence of arithmetic error

- Instruction coding must specify what registers to use for operands.

- Loads and stores may use register to specify address of memory location.
Instruction Set for a 4 Register CPU

16 bit binary instruction formats.

00aa rrdd dddd dddd  Load the register specified by rr. The aa bits specify the addressing mode: 00 for immediate, 01 for direct and 10 for indirect. For immediate addressing, the register is loaded with the value specified by the d bits. For direct addressing, it is loaded from the location specified by the d bits. For indirect addressing, it is loaded from the address stored in the location specified by the d bits.

01aa rrdd dddd dddd  Store the value in the register specified by rr. The aa bits specify the addressing mode: 01 for direct, 10 for indirect.

1000 0000 rraa xxxx  Copy the value in register aa to register rr.

1000 0001 rraa bbxx  Add the values in registers aa and bb and put the result in register rr.
1000 0010   rraa bbxx Subtract the value in register bb from the value in register aa and put the result in rr.
1000 0011   rraa xxxx Negate the value in register aa and put the result in rr.
1000 0100   rraa bbxx Shift the value in register aa to the left by the number of positions specified by register bb and put the result in rr.
1000 0101   rraa bbxx Shift the value in register aa to the right by the number of positions specified by register aa and put the result in rr.
1000 1111   xxxx xxxx Halt
1100 aatt  tttt  tttt Branch if the value in register aa is zero to the location specified by the t bits.
1101 aatt  tttt  tttt Branch if the value in register aa is less than zero to the location specified by the t bits.
1110 aatt  tttt  tttt Branch if the value in register aa is greater than zero to the location specified by the t bits.
1111 xxtt  tttt  tttt Branch to the location specified by the t bits.
entity instruction_register is
    port(
        clk, en_A, en_D, ld, reset:
            in STD_LOGIC;
        aBus:
            out STD_LOGIC_VECTOR(15 downto 0);
        dBus:
            inout STD_LOGIC_VECTOR(15 downto 0);
        load, store, copy, add, sub, negate:
            out STD_LOGIC;
        shiftL, shiftR, halt:
            out STD_LOGIC;
        brZero, brLess, brGtr, branch:
            out STD_LOGIC;
        treg, areg, breg, amode:
            out STD_LOGIC_VECTOR(1 downto 0);
        irRegX:
            out STD_LOGIC_VECTOR(15 downto 0)
    );
end instruction_register;
architecture irArch of instruction_register is
signal irReg: STD_LOGIC_VECTOR(15 downto 0);
begin
    process(clk) begin
        if clk'event and clk = '0' then
            if reset = '1' then
                irReg <= x"0000";
            elsif ld = '1' then
                irReg <= dBus;
            end if;
        end if;
    end process;
aBus <= "000000" & irReg(9 downto 0) when en_A='1' else "ZZZZZZZZZZZZZZZZZ";
dBus <= "000000" & irReg(9 downto 0) when en_D='1' else "ZZZZZZZZZZZZZZZZZZZ";
load <= '1' when irReg(15 downto 14) = "00" else '0';
store <= '1' when irReg(15 downto 14) = "01" else '0';
copy  <= '1' when irReg(15 downto 8)  =x"80" else '0';
add   <= '1' when irReg(15 downto 8)  =x"81" else '0';

branch<= '1' when irReg(15 downto 12)=x"f"  else '0';
treg  <= irReg(11 downto 10) when irReg(15)= '0' else
  irReg( 7 downto  6)
    when irReg(15 downto 12)=x"8" else "00";
areg  <= irReg( 5 downto  4)
    when irReg(15 downto 14)="10" else
  irReg(11 downto 10)
    when irReg(15 downto 14)="11" else "00";
breg <= irReg( 3 downto  2)
    when irReg(15 downto 14)="10" else "00";
amode <= irReg(13 downto 12);
end irArch;
Register File for 4 Register CPU

entity regFile is
  port (clk, en_D, ld, selALU, reset:
in STD_LOGIC;
  target, srcA, srcB, srcD:
in STD_LOGIC_VECTOR(1 downto 0);
  ALUbusR:
in STD_LOGIC_VECTOR(15 downto 0);
  dBus:  inout STD_LOGIC_VECTOR(15 downto 0);
  ALUbusA, ALUbusB: out STD_LOGIC_VECTOR(15 downto 0);
  r0, r1, r2, r3: out STD_LOGIC_VECTOR(15 downto 0)
);
end regFile;

architecture regFileArch of regFile is
type regFile_typ is array(0 to 3) of
  STD_LOGIC_VECTOR(15 downto 0);
signal reg: regFile_typ;
begin

process(clk) begin
    if clk'event and clk = '1' then
        if reset = '1' then
            for i in 0 to 3 loop
                reg(i) <= x"0000";
            end loop;
        elsif ld = '1' and selALU = '1' then
            reg(conv_integer(unsigned(target))) <= ALUbusR;
        elsif ld = '1' and selALU = '0' then
            reg(conv_integer(unsigned(target))) <= dBus;
        end if;
    end if;
end process;

dBus <= reg(conv_integer(unsigned(srcD))) when en_D = '1'
    else "Z:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\\";
    ALUbusA <= reg(conv_integer(unsigned(srcA)));
    ALUbusB <= reg(conv_integer(unsigned(srcB)));
    r0 <= reg(0); r1 <= reg(1); r2 <= reg(2); r3 <= reg(3);
end regFileArch;
A cache is a small memory that contains recently used words of memory.

Conceptually simplest cache is the fully associative cache which stores (key, data) pairs.
- associative lookup using key to find data
- implementation involves parallel comparison of stored keys with “query key”

In cache, main memory address used as key.
- before retrieving word from main memory, first check for it in cache
- retrieved words stored in cache
Direct-Mapped Caches

- Fully associative caches are too expensive to be cost-effective in most computers.
- A direct mapped cache is a less expensive alternative that uses SRAM and performs well in common cases.
  - words stored at cache location specified by lower DRAM address bits
  - higher DRAM address bits stored with data
  - to see if DRAM word stored in cache, lookup using low bits and check tag against high bits
  - works well for sequentially accessed DRAM locations
Set Associative Caches

- Set associative caches are an intermediate alternative between fully associative and direct-mapped caches.
  - In a 2-way s.a. cache, there are 2 SRAM banks and any given memory word can be stored in either one.
  - Tags compared on lookup to see if either stored word matches address.
- Better performance than direct-mapped.
- Less expensive than fully associative cache.
- Can be generalized to N-way (typically 4, 5).
More About Cache Operation

- Whenever a word is needed from memory, first check the cache and use the stored copy if possible.
- If word not in cache, fetch the cache line containing the required word and put it into the cache (note delay).
  - retrieved cache line replaces one of the stored cache lines; replacement policy determines which cache line is replaced
  - replaced lines written back to memory if modified (dirty bit)
  - for sequentially accessed data, fetching whole cache line speeds up later accesses
- Many processors have multiple caches.
  - first-level cache usually on-chip, separate instruction cache
  - second-level cache typically off-chip and much larger
- Cache consistency issue in multiple processor systems.
Pipelining

- Most modern processors use pipelining to improve performance.
- Simplest form overlaps instruction fetch and execution.
  - if instructions are in instruction cache and data in data cache or registers, can nearly double effective processor speed
- By splitting instructions into several steps, parts of several instructions can be executed at the same time.
  - most processors have at least 4 pipeline stages
- Conditional branches hurt pipeline efficiency.
  - branch prediction hardware attempts to guess which way branch will go, in order to keep pipeline busy
  - quite effective for conditional branches in loops, which are very “predictable”