Improving Processor Performance – Part 1

- Memory latency and performance
- Expanded instruction set
- Processors with multiple registers

Jon Turner
Making Computers Faster

- Large & growing speed gap between CPU & DRAM
  - gate delays now less than 20 ps
  - can take 100 ns to retrieve a word from DRAM
  - improve performance by making fewer memory accesses

- Modern processors improve performance by
  - using more extensive instruction sets
  - providing additional addressing modes
  - using multiple registers in place of single accumulator
  - using small, fast cache memories to hold recently used instructions and data
  - overlapping the execution of several instructions (pipelining)
    - example: fetch next instruction while executing current one
  - providing multiple “cores” for parallel instruction execution
Extending Instruction Set

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

- Conditional branch instructions
  - sign of register
  - result of most recent comparison
  - state of “condition flags”

- Instruction coding must specify what registers to use for operands

- Loads and stores may use register to specify address of memory location
Exercises

1. Since the WASHU-2 has no multiply instruction, programs that require multiplication must use a subprogram that multiplies two values together. How many instructions are required to execute the multiplication subprogram described in Chapter 17?

2. Our implementation of the WASHU-2 uses the prototype board’s on-chip memory. Since the on-chip memory can do a read or write in under 20 ns, the memory does not have a big impact on the processor’s performance. Suppose that instead of using the on-chip memory, we used an off-chip DRAM, and that it took 100 ns to retrieve a word from the off-chip DRAM. For such an implementation, approximately how many clock cycles would be required for an instruction fetch? How many would be needed for the execution phase of a negate instruction? How many would be needed for the execution phase of an indirect load? How many would be needed for the execution phase of a direct store?
Instruction Set for WashU-10

- 16 general purpose registers
- Expanded instruction set. Note, some require two words.

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Description</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0tdd</td>
<td>Constant Load.</td>
<td>$R[t] = ssdd$ (sign-extended)</td>
</tr>
<tr>
<td>1txx</td>
<td>Direct Load.</td>
<td>$R[t] = M[aaaa]$</td>
</tr>
<tr>
<td>2tsx</td>
<td>Indexed Load.</td>
<td>$R[t] = M[R[s]+x]$</td>
</tr>
<tr>
<td>3tsx</td>
<td>Indexed Load with Increment.</td>
<td>$R[t] = M[R[s]+x]; R[s]++$</td>
</tr>
<tr>
<td>4tsx</td>
<td>Indexed Load with Decrement.</td>
<td>$R[s]--; R[t] = M[R[s]+x]$</td>
</tr>
<tr>
<td>5sxx</td>
<td>Direct Store.</td>
<td>$M[aaaa] = R[s]$</td>
</tr>
<tr>
<td>6tsx</td>
<td>Indexed Store.</td>
<td>$M[R[t]+x] = R[s]$</td>
</tr>
<tr>
<td>7tsx</td>
<td>Indexed Store with Increment.</td>
<td>$M[R[t]+x] = R[s]; R[t]++$</td>
</tr>
<tr>
<td>8tsx</td>
<td>Indexed Store with Decrement.</td>
<td>$R[t]--; M[R[t]+x] = R[s]$</td>
</tr>
</tbody>
</table>
90ts  Copy.  \( R[t] = R[s] \)
91ts  Add.  \( R[t] = R[t] + R[s] \)
92ts  Subtract.  \( R[t] = R[t] - R[s] \)
93ts  Negate.  \( R[t] = -R[s] \)
A0ts  And.  \( R[t] = R[t] \text{ and } R[s] \)
A1ts  Or.  \( R[t] = R[t] \text{ or } R[s] \)
A2ts  Exclusive-or.  \( R[t] = R[t] \text{ xor } R[s] \)
B000  Halt.
C0xx tttt  Branch.  \( PC = tttt \)
Cltt  Relative Branch.  \( PC = PC + sttt \) (sign-extended addition)
Dsst  Relative Branch on Zero.
    if \( R[s] = 0 \) then \( PC = PC + sstt \)
E Utt  Relative Branch on Plus.
    if \( R[s] > 0 \) then \( PC = PC + sstt \)
F Utt  Relative Branch on Minus.
    if \( R[s] < 0 \) then \( PC = PC + sstt \)
Register File for WASHU-16

- 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
entity cpu is port (
    clk, reset: in std_logic;
    en, rw: out std_logic;
    aBus: out address;
    dBus: inout word;
    ...
end cpu;
architecture a1 of cpu is

type state_type is (reset_state, fetch, dload, xload, xloadI, xloadD, dstore, xstore, xstoreI, xstoreD, copy, add, sub, neg, andd, orr, xor, halt, branch, rbranch, rbranchZ, rbranchP, rbranchN);

signal state: state_type;
signal tick: std_logic_vector(3 downto 0);

signal pc: address; -- program counter
signal iReg: word; -- instr register
signal maReg address; -- mem addr reg

type regFile is array(0 to 15) of word;
signal reg: regFile; -- register file

signal target: std_logic_vector(3 downto 0); -- target register
signal source: std_logic_vector(3 downto 0); -- source register
begin
  ...
  process (clk)
  procedure decode is begin
    -- default assignments to target and source
    target <= ired(11 downto 8); source <= ired(7 downto 4);
    -- Instruction decoding.
    case ired(15 downto 12) is
      when x"0" => state <= cload;
      when x"1" => state <= dload;
    ...
    when x"9" => case ired(11 downto 8) is
      when x"0" => state <= copy;
      when x"1" => state <= add;
      when x"2" => state <= sub;
      when x"3" => state <= neg;
      when others => state <= halt;
    end case;
    target <= ired(7 downto 4); source <= ired(3 downto 0);
    ...
    when x"f" => state <= rbranchM; source <= ired(11 downto 8);
    when others => state <= halt;
  end case;
  end procedure decode;
begin
  if rising_edge(clk) then
    if reset = '1' then
      ...
    else
      tick <= tick + 1; -- advance time by default
      case state is
      when reset_state => wrapup;
      when fetch => case tick is
        when x"1" =>
          ireg <= dBus;
        when x"2" =>
          pc <= pc + 1;
          if ireg(15 downto 12) /= x"1" and
          ireg(15 downto 12) /= x"5" and
          ireg(15 downto 8) /= x"c0"
          then
            decode; tick <= x"0";
          end if;
        when x"4" =>
          maReg <= dBus;
          decode; tick <= x"0";
          pc <= pc + 1;
        when others =>
          end case;
      end case;
  end if;
end begin
-- load instructions
when clold => reg(int(target)) <= iregSx8; wrapup;
when dload | xload =>
  if tick = x"1" then
    reg(int(target)) <= dBus; wrapup;
  end if;
when xloadI =>
  if tick = x"1" then
    reg(int(target)) <= dBus;
    reg(int(source)) <= reg(int(source)) + 1;
    wrapup;
  end if;
when xloadD =>
  if tick = x"0" then
    reg(int(source)) <= reg(int(source)) - 1;
  elsif tick = x"1" then
    reg(int(target)) <= dBus; wrapup;
  end if;

-- store instructions
when dstore | xstore => wrapup;
when xstoreI =>
  reg(int(target)) <= reg(int(target)) + 1; wrapup;
when xstoreD =>
  reg(int(target)) <= reg(int(target)) - 1; wrapup;
-- register-to-register instructions
when copy =>
  reg(int(target)) <= reg(int(source));
  wrapup;
when add =>
  reg(int(target)) <=
    reg(int(target)) + reg(int(source));
  wrapup;
...

-- branch instructions
when branch => pc <= maReg; wrapup;
when rbranch2 =>
  if reg(int(source)) = x"0000" then
    pc <= (pc - 1) + iregSx8;
  end if;
  wrapup;
...
when others => state <= halt;
  end case;
  end if;
end if;
end if;
end process;
-- combinational process controlling memory signals
process(ireg,pc,maReg,reg,state,tick) begin
    en <= '0'; rw <= '1';
    aBus <= (others => '0'); dBus <= (others => 'Z');
    case state is
    when fetch =>
        if tick = x"0" or tick = x"3" then
            en <= '1'; aBus <= pc;
        end if;
    when dload =>
        if tick = x"0" then
            en <= '1'; aBus <= maReg;
        end if;
    when xload | xloadI =>
        if tick = x"0" then
            en <= '1'; aBus <= reg(int(source)) + ireg(3 downto 0);
        end if;
    when xstore | xstoreI =>
        en <= '1'; rw <= '0';
        aBus <= reg(int(target)) + ireg(3 downto 0);
        dBus <= reg(int(source));
    end case;
end process;
end al;
<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>reset</td>
<td>0</td>
</tr>
<tr>
<td>bus signals</td>
<td>0</td>
</tr>
<tr>
<td>en</td>
<td>1</td>
</tr>
<tr>
<td>riv</td>
<td>0</td>
</tr>
<tr>
<td>abd[15:0]</td>
<td>0000</td>
</tr>
<tr>
<td>dbd[15:0]</td>
<td>2222</td>
</tr>
<tr>
<td>fp</td>
<td>0</td>
</tr>
<tr>
<td>state</td>
<td>0</td>
</tr>
<tr>
<td>coo[3:0]</td>
<td>1010</td>
</tr>
<tr>
<td>special registers</td>
<td></td>
</tr>
<tr>
<td>pc[15:0]</td>
<td>000E</td>
</tr>
<tr>
<td>reg[15:0]</td>
<td>0311</td>
</tr>
<tr>
<td>mreg[15:0]</td>
<td>0032</td>
</tr>
<tr>
<td>source[3:0]</td>
<td>1</td>
</tr>
<tr>
<td>target[3:0]</td>
<td>1</td>
</tr>
<tr>
<td>General purpose</td>
<td></td>
</tr>
<tr>
<td>reg[0:15]</td>
<td>100000</td>
</tr>
<tr>
<td>b[0]</td>
<td>0000</td>
</tr>
<tr>
<td>b[1]</td>
<td>079F</td>
</tr>
<tr>
<td>b[2]</td>
<td>001F</td>
</tr>
<tr>
<td>b[3]</td>
<td>1234</td>
</tr>
<tr>
<td>b[4]</td>
<td>0000</td>
</tr>
<tr>
<td>b[5]</td>
<td>4711</td>
</tr>
<tr>
<td>b[6]</td>
<td>5525</td>
</tr>
<tr>
<td>enex</td>
<td>1254</td>
</tr>
</tbody>
</table>

### Logic Diagrams

- **xloadi**: [Diagram]
- **copy**: [Diagram]
- **subtract**: [Diagram]
- **xorrr**: [Diagram]
Exercises

1. Rewrite the multiply subroutine from Chapter 17 using the WASHU-16’s instruction set. Use the same method for passing arguments and returning results. Use registers to hold all the program variables. How many clock ticks are required to execute one iteration of the main loop? How many clock ticks are used by the WASHU-2 version?

2. Show how you would modify the VHDL for the WASHU-16 to implement an *add-constant* instruction with the following instruction summary.

   93tc add constant to register; R[t]=R[t] + sssc

   The last hex digit is added to the register, after sign extension. This allows us to add small constants to a register, with the most common use being to increment or decrement the value in a register.

3. Suppose we wanted to implement the WASHU-16 on the prototype board. How would you re-work the LCD display so that you could see the values of all the registers? What changes are required to the interface between the processor and the console?