Design of the WashU-2 Processor – Part 2

- Processor execution cycle
- Instruction timing
- VHDL specification of processor

Jon Turner
Processing Cycle

- **Instruction fetch**
  - *PC* used to load IREG from memory
  - *PC* is incremented

- **Instruction decode**
  - Determine which instruction is to be executed, based on instruction encoding
    - e.g., if first hex digit is 5, then dStore
    - Transition to appropriate next state

- **Instruction execution**
  - Retrieve additional memory words
  - Write to memory
  - Modify *PC* or *ACC* contents
  - May take different amounts of time to complete
Instruction Fetch

1. Memory contents on Dbus

2. Memory contents on Dbus

3. Load IREG

4. Increment PC

1. PC value on Abus

2. PC value on Abus

3. PC value on Abus

state

tick

1. mem_en <= 1

3. mem_en <= 0

mem_en

mem_en

mem_en

mem_en

mem_en

Instruction Execution

- **Direct Load**
  - transfer data from memory to \( ACC \), using high 4 bits of PC and low 12 bits of instruction word as memory address
  - requires asserting of memory signals and loading \( ACC \)

- **Conditional branch**
  - determine if \( ACC=0 \) (or \( >0 \) or \( <0 \))
  - if so, add sign-extended low 8 bits of instruction to \( PC \)

- **Indirect store**
  - transfer data from memory to Indirect Address Register (IAR) using high 4 bits of PC and low 12 bits of instruction word as memory address
  - transfer data from \( ACC \) to memory, using IAR for address
  - requires placing IAR value on address bus and asserting signals to perform memory write
Signal Timing for Processor

**Fetch**
- PC value on Abus
- value returned on next tick

**Negate cLoad**

**Branch**

**dload, add, and**

- ACC loaded
- PC loaded
- IREG loaded
- PC incremented
- ACC loaded

- clk
- mem_en
- mem_rw
- abus
- dbus
- pc
- ireg
- acc
- iram
- dbus
- pc
- ireg
- acc
Signal Timing for Processor
Exercises

1. Suppose we wanted to add a *memory increment* instruction to the Washu2. Assume the instruction code is Axxx. The effect of the instruction is to add 1 to the value in M[pxxx], where p is the high 4 bits of the PC. Write out the sequence of steps that are required to implement this instruction.

2. Draw an instruction timing diagram for the memory increment instruction from problem 1.

3. Suppose we wanted to add an *indirect load with increment* instruction to the Washu2. Assume the instruction code is Bxxx. The instruction works like an indirect load, but has the additional side-effect of incrementing the pointer. Write out the sequence of steps that are required to implement this instruction.

4. Draw an instruction timing diagram for the indirect load with increment instruction from problem 3.
VHDL – interface and registers

entity cpu is port (
  clk, reset, pause: in std_logic;
  en, rw: out std_logic;
  aBus: out address; dBus: inout word;
  regSelect: in std_logic_vector(1 downto 0);
  dispReg: out word);
end cpu;
architecture cpuArch of cpu is

type state_type is (resetState, pauseState, fetch, halt, negate, branch, brZero, ..., cLoad, dLoad, iLoad, dStore, iStore, add, andd
signal state: state_type;
signal tick: std_logic_vector(3 downto 0);
signal pc: address; -- program counter
signal iReg: word; -- instruction register
signal iar: address; -- indirect address register
signal acc: word; -- accumulator
signal alu: word; -- alu output
signal this: address; -- address of instruction being executed
signal opAdr: address; -- address for direct load, store, add, andd,...
signal target: word; -- target for branch instruction
begin
  opAdr <= this(15 downto 12) & iReg(11 downto 0);
  target <= this + ((15 downto 8 => iReg(7)) & iReg(7 downto 0) with regSelect select -- connect selected register to console dispReg <= iReg when "00", this when "01",...
VHDL – instruction decoding

```vhdl
alu <= (not acc) + x"0001" when state = negate else
  acc + dbus when state = add else
  acc and dbus when state = andd else
(alu\'range => '0');
```

```vhdl
process (clk) -- main cpu process
  begin
    function decode(instr: word) return state_type is begin
      -- Instruction decoding.
      case instr(15 downto 12) is
        when x"0" =>
          case instr(11 downto 8) is
            when x"0" =>
              if instr(11 downto 0) = x"000" then return halt;
              elsif instr(11 downto 0) = x"001" then return negate;
              else return halt;
            end if;
            when x"1" => return branch;
            when others => return halt;
          end case;
          when x"1" => return cLoad;
          when others => return andd;
        when others => return halt;
      end case;
    end function decode;
  end process;
```

```
```
procedure wrapup is begin
  -- used in last tick of every instruction
  if pause = '1' then state <= pauseState;
  else state <= fetch; tick <= x"0";
  end if;
end procedure wrapup;
begin
  if rising_edge(clk) then
    if reset = '1' then
      state <= resetState; tick <= x"0";
      pc <= (others => '0'); ...
    else
      tick <= tick + 1;
      if state = resetState then
        state <= fetch; tick <= x"0";
      elsiif state = pauseState then
        if pause = '0' then
          state <= fetch; tick <= x"0";
        end if;
      elsiif state = fetch then
        if tick = x"1" then
          iReg <= dBus;
        elsif tick = x"2" then
          state <= decode(iReg);
          this <= pc; pc <= pc + 1; tick <= x"0";
        end if;
    end if;
  end if;
 VHDL – branch instructions

```vhdl
else
    case state is
        when branch =>
            pc <= target;
            wrapup;
        when brZero =>
            if acc = x"0000" then
                pc <= target;
            end if;
            wrapup;
        when brPos => ...
        when brNeg => ...
        when brInd =>
            if tick = x"1" then pc <= dBus; wrapup; end if;
    end case;
end if;
```

---

- **Conditional branches**
- **Indirect branch to address retrieved from memory**
- **Target calculation**: `target=this+ssxx` where `xx=IREG(7..0)`
VHDL - load, store, arithmetic

-- load instructions
when clload =>
  acc <= (15 downto 12 => iring(11)) & iring(11 downto 0);
  wrapup;
when dload =>
  if tick = x"1" then acc <= dBus; wrapup; end if;
when lload =>
  if tick = x"1" then iar <= dBus;
  elsif tick = x"3" then acc <= dBus; wrapup;
  end if;

-- store instructions
when dstore => wrapup;
when istore =>
  if tick = x"1" then iar <= dBus;
  elsif tick = x"2" then wrapup;
  end if;

-- arithmetic and logic instructions
when negate => acc <= alu; wrapup;
when add | andd =>
  if tick = x"1" then acc <= alu; wrapup; end if;
when others => state <= halt;
end case;
VHDL – memory control process

process (ireg, pc, dpc, iar, acc, state, tick) begin

  -- default values for memory control signals
  en <= '0'; rw <= '1';
  aBus <= (others => 'Z'); dBus <= (others => 'Z');
  case state is
  when fetch => if tick = x"0" then
    en <= '1'; aBus <= pc;
    end if;
  when dLoad | add | andd =>
    if tick = x"0" then
      en <= '1'; aBus <= opAdr;
    end if;
  when iLoad =>
    if tick = x"0" then
      en <= '1'; aBus <= opAdr;
    elsif tick = x"2" then
      en <= '1'; aBus <= iar;
    end if;
  when dStore => ...
  when iStore => ...
  when others =>
  end case;
end process; end cpuArch;
Exercises

1. Show how you would modify the VHDL code for the Washu2 to implement the memory increment instruction discussed in an earlier exercise.

2. Show how you would modify the VHDL code for the Washu2 to implement the indirect load with increment instruction discussed in an earlier exercise.