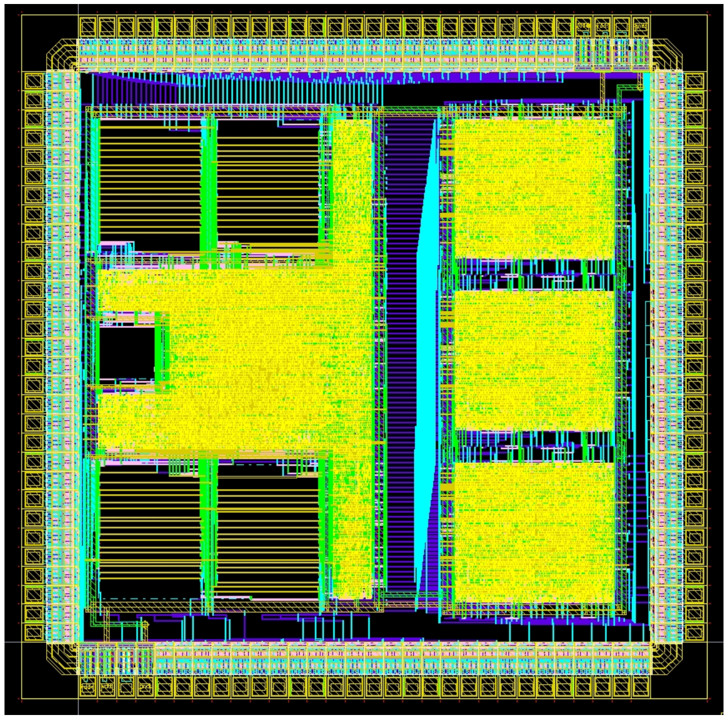
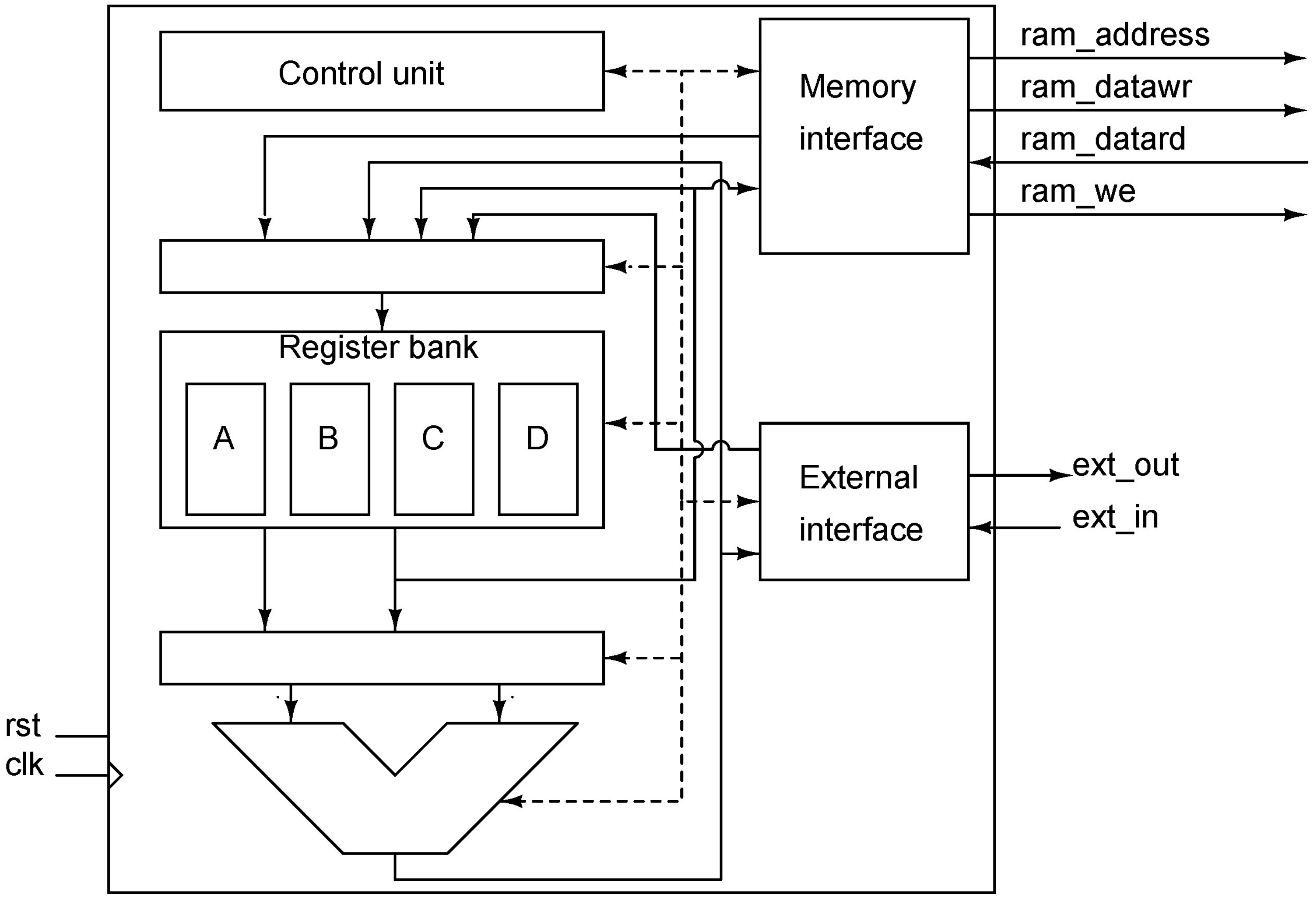
Digital Systems and Microprocessor Design (H7068)



8.2. Inside UoS Educational Processor

Daniel Roggen
d.roggen@sussex.ac.uk

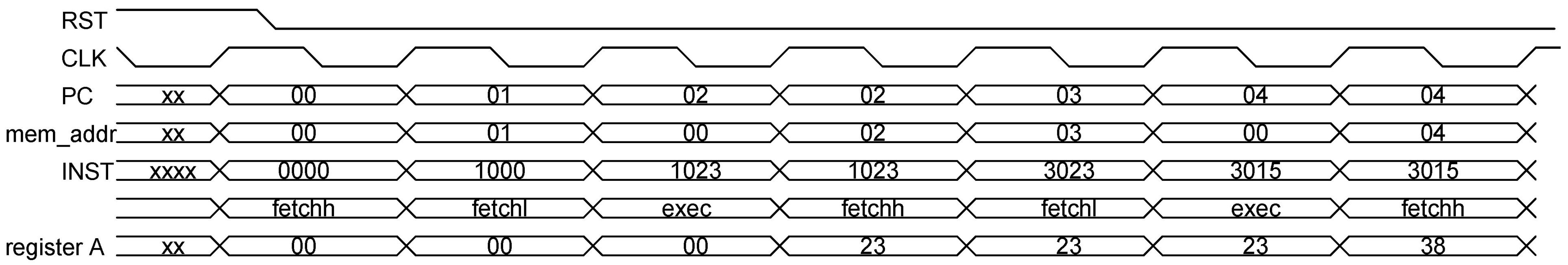
Architecture



VHDL files

• cpu.vhd:	top file for CPU
• cpusequencer.vhd:	fetch/execute sequence
• cpuregbank.vhd:	register file
• cpualu.vhd:	ALU

Fetch/Execute sequence



Memory dump:

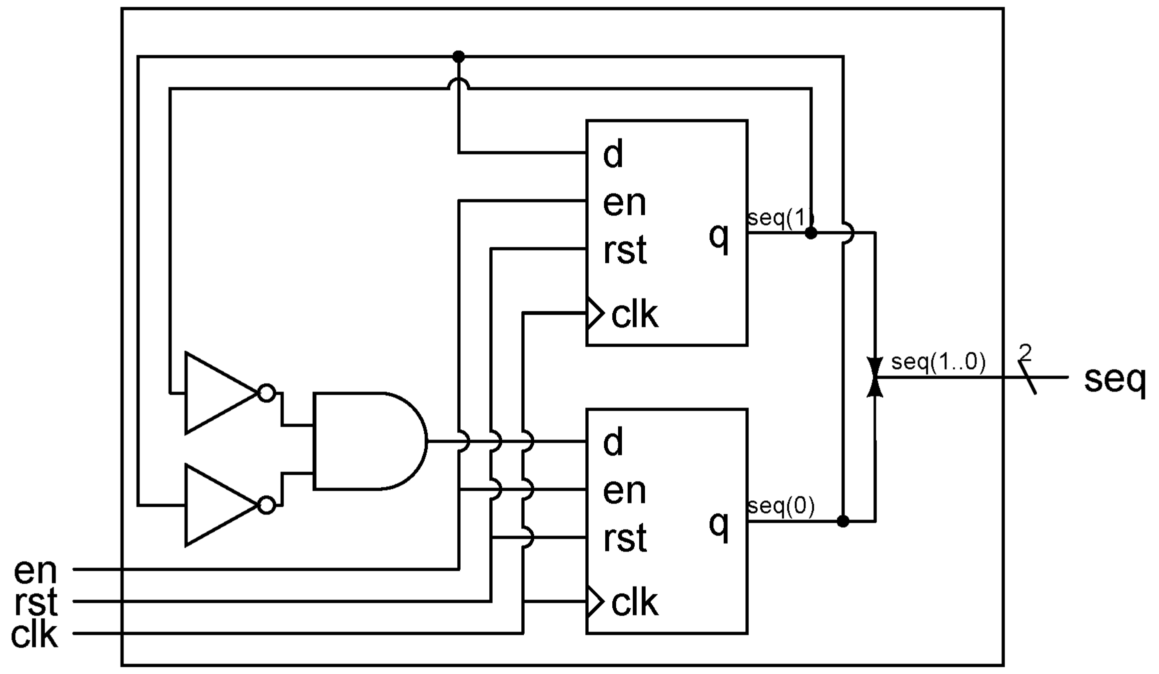
Address	Content
0:	10
1:	23
2:	30
3:	15

- 3 clock cycles per instruction
  - Fetch high
  - Fetch low
  - Execute...
- Circuit must organize this sequence

Fetch/Execute sequence

- In cpusequencer.vhd

Instruction fetch/execute state sequence



- Up counter until 2 (10b): 00=fetchh, 01=fetchl, 10=exec

Fetch/Execute sequence

```
entity cpusequencer is
   port(
        clk : in STD_LOGIC;
        rst : in STD_LOGIC;
        en : in STD_LOGIC;
        seq : out STD_LOGIC_VECTOR(1 downto 0)
        );
end cpusequencer;
```

- clk: clock
- rst: reset
- en: enable (currently not used: always wired to 1)
- seq: 2-bit code for fetchh, fetchl, exec

Fetch/Execute sequence

```
process(clk)
  begin
      if rising_edge(clk) then
          if rst='1' then
              s<="00";
          else
              if en='1' then
                  if s="10" then
                      s <= "00";
                  else
                      s <= s+1;
                  end if;
              end if;
          end if;
      end if;
  end process;

  seq <= s;
```

clock in sensitivity list

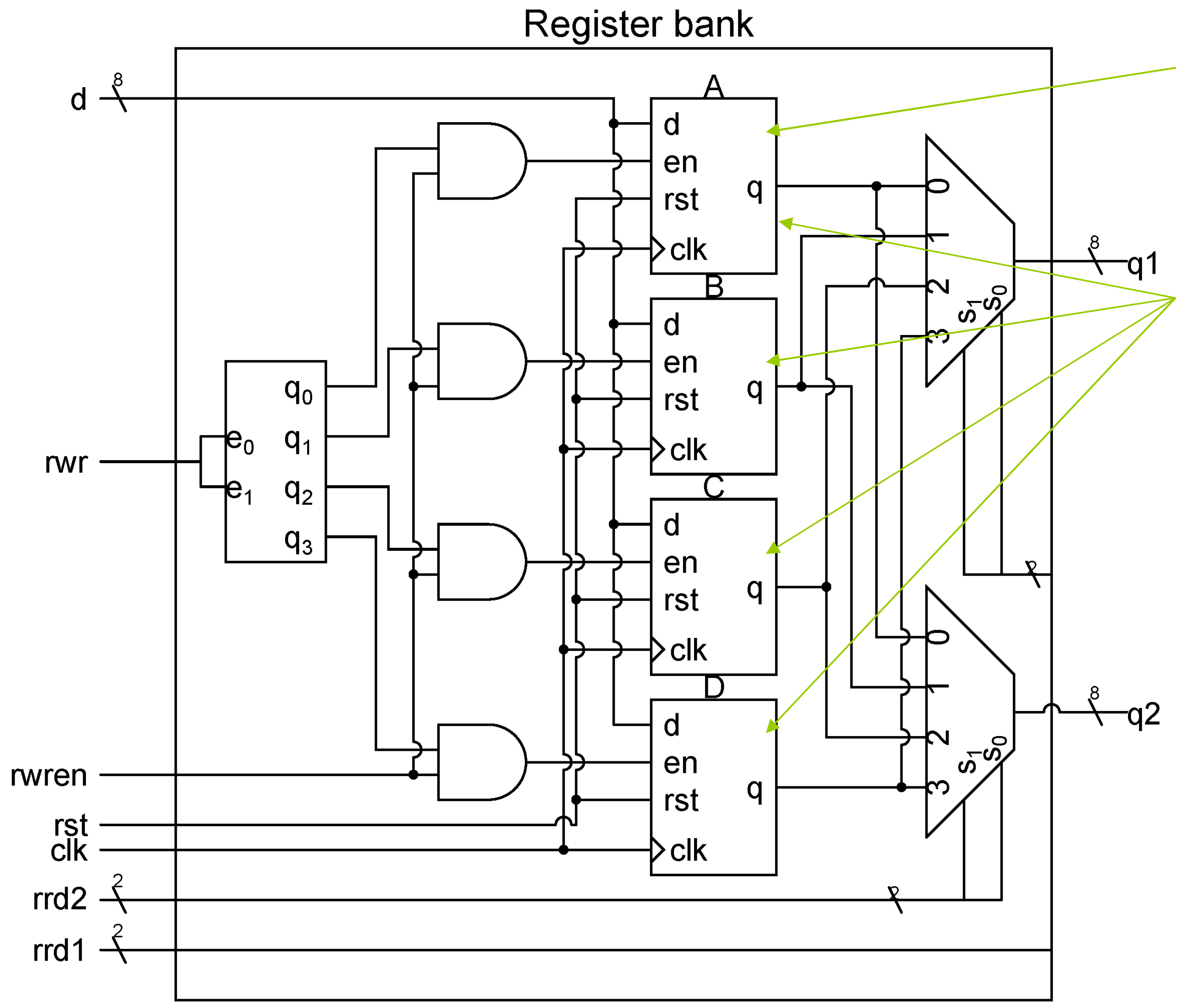
All happens on rising clock edge

Reset

Normal behavior: if enabled, increment and wrap around at 2

- VHDL does additions with '+'!
- Not taught in this module.
- Synthesizer chooses an implementation on its own
- Choice not critical here with 2 bits; with more bits specific adder architectures may be preferred

Register file (register bank)



All D FF with enable and reset (synchronous)

registers A-D

Register file (register bank)

- Reminder: move instruction
- Always operation between dst and src, with result in dst
- Let's consider what to read from our register bank:
  - The register defined by dst and src (even if we deal with an immediate-this is handled elsewhere

Instructions	instruction(15..8)								Instruction(7..0)							
Move	Opcode				dd#m	sd#m	dreg		src							
mov r, r	0	0	0	0	0	0	r	r	–	–	–	–	–	–	r	r
mov r, i	0	0	0	1	0	0	r	r	i	i	i	i	i	i	i	i
mov r, [r]	0	0	0	0	0	1	r	r	–	–	–	–	–	–	r	r
mov r, [i]	0	0	0	1	0	1	r	r	i	i	i	i	i	i	i	i
mov [r], r	0	0	0	0	1	0	r	r	–	–	–	–	–	–	r	r
mov [r], i	0	0	0	1	1	0	r	r	i	i	i	i	i	i	i	i

Register file (register bank)

```
entity cpuregbank is
   port(
        clk : in STD_LOGIC;
        rrd1 : in STD_LOGIC_VECTOR(1 downto 0);
        rrd2 : in STD_LOGIC_VECTOR(1 downto 0);
        rwr : in STD_LOGIC_VECTOR(1 downto 0);
        rwren : in STD_LOGIC;
        rst : in STD_LOGIC;
        d : in STD_LOGIC_VECTOR(7 downto 0);
        q1 : out STD_LOGIC_VECTOR(7 downto 0);
        q2 : out STD_LOGIC_VECTOR(7 downto 0);
        -- Only for debugging
        dbg_qa : out STD_LOGIC_VECTOR(7 downto 0);
        dbg_qb : out STD_LOGIC_VECTOR(7 downto 0)
        dbg_qc : out STD_LOGIC_VECTOR(7 downto 0)
        dbg_qd : out STD_LOGIC_VECTOR(7 downto 0)
        );
end cpuregbank;
```

register to read from (the 2 bits come from src or dst in the instruction!)

register to write to (always dst)

rwren: enable write
d: data to write

outputs of the register bank

This would not be here in a "production" processor: here to display the content of the register on the 7 segment display

Register file (register bank)

```
architecture Behavioral of cpuregbank is
   signal enables: STD_LOGIC_VECTOR(3 downto 0);
   signal qa,qb,qc,qd: STD_LOGIC_VECTOR(7 downto 0);
begin

   ra: entity work.dffre generic map (N=>8) port
   map(clk=>clk,en=>enables(0),rst=>rst,d=>d,q=>qa);

   rb: entity work.dffre generic map (N=>8) port
   map(clk=>clk,en=>enables(1),rst=>rst,d=>d,q=>qb);

   rc: entity work.dffre generic map (N=>8) port
   map(clk=>clk,en=>enables(2),rst=>rst,d=>d,q=>qc);

   rd: entity work.dffre generic map (N=>8) port
   map(clk=>clk,en=>enables(3),rst=>rst,d=>d,q=>qd);
```

enable each register
q: output of each register

Instantiate a register with reset and enable.

Generic map: parameterized register.

Register file (register bank)

2-4 decoder: when write is enabled one of the register is enabled for write

```
    with rwr select
         enables <="0001" and rwren&rwren&rwren&rwren when "00",
                  "0010" and rwren&rwren&rwren&rwren when "01",
                  "0100" and rwren&rwren&rwren&rwren when "10",
                  "1000" and rwren&rwren&rwren&rwren when others;
```

first output multiplexer

```
    with rrd1 select
         q1 <=    qa when "00",
                  qb when "01",
                  qc when "10",
                  qd when others;
```

second output multiplexer

```
    with rrd2 select
         q2 <=    qa when "00",
                  qb when "01",
                  qc when "10",
                  qd when others;
```

Register

```
entity dffre is
   generic (N : integer);
   port(
       clk : in STD_LOGIC;
       en : in STD_LOGIC;
       rst: in STD_LOGIC;
       d : in STD_LOGIC_VECTOR(N-1 downto 0);
       q : out STD_LOGIC_VECTOR(N-1 downto 0)
       );
end dffre;
```

generic: useful to create components that can be parameterized

Here N is the number of bits of the D flip-flop

Register

```
architecture Behavioral of dffre is
begin

   process(clk)
   begin
        if rising_edge(clk) then
                if rst='1' then
                        q<=(others=>'0');
                else
                        if en='1' then
                                q<=d;
                        end if;
                end if;
        end if;
   end process;
```

all on rising edge

synchronous reset

copy if enabled

VHDL generic

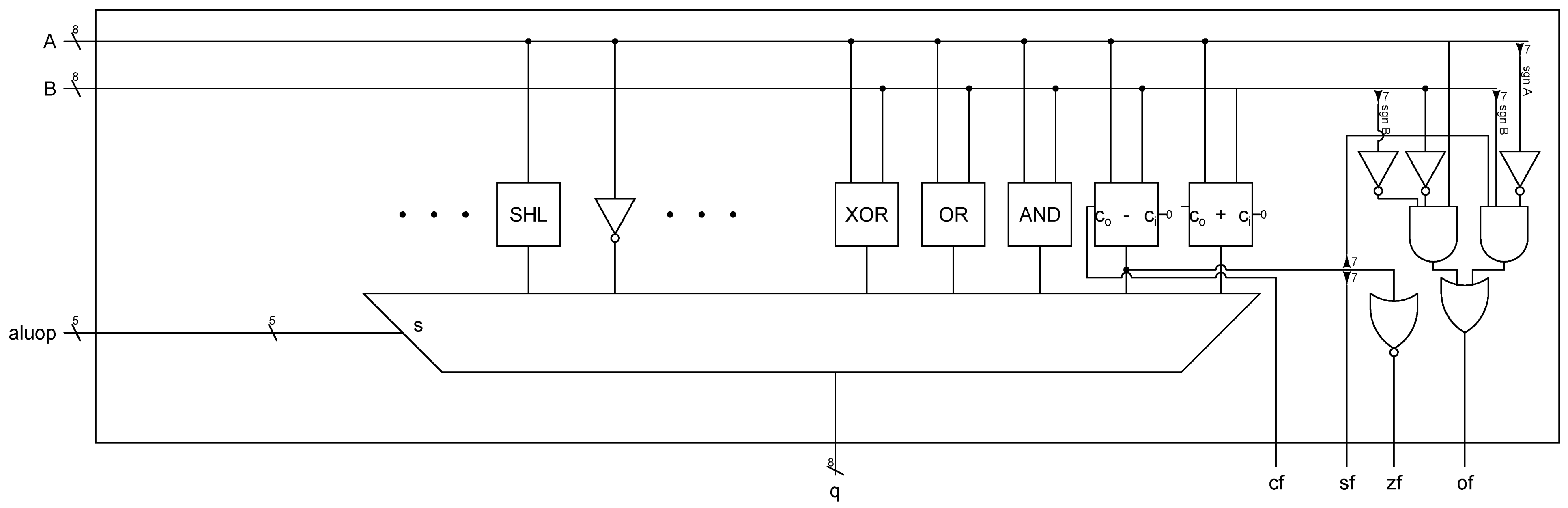
<u>In the component declaration:</u>

```
entity entity_name is
  generic (generic list);
  port (port list);
end entity_name;
```

<u>In the component instantiation:</u>

```
label: entity work.comp_name
     generic map (generic_association_list)
     port map (port_association_list);
```

ALU



- A, B: input of the ALU (8 bits)
- aluop: operation to perform (5 bits)
  - bit 14 to bit 10 of instruction to identify the type of operation
  - (part opcode and part other bits of the instruction)
- Implemented as multiplexer
- flags as discrete logic

ALU

Instructions	instruction(15..8)								Instruction(7..0)							
ALU 2 op	opcode			$\overline{R}/I$	ALU op		dreg		src							
add r, r	0	0	1	0	0	0	r	r	-	-	-	-	-	-	r	r
add r, i	0	0	1	1	0	0	r	r	i	i	i	i	i	i	i	i
xor r, r	0	1	0	0	0	0	r	r	-	-	-	-	-	-	r	r

Test	opco de				ALU op		dr eg		immedite / reg							
cmp r, r	0	1	0	0	0	1	r	r	-	-	-	-	-	-	r	r
cmp r, i	0	1	0	1	0	1	r	r	i	i	i	i	i	i	i	i

Instructions	instruction(15..8)								Instruction(7..0)							
ALU 1 op	opcode			ALU op			dreg									
not r	0	1	1	0	0	0	r	r	-	-	-	-	-	-	-	-
shr r	0	1	1	0	0	1	r	r	-	-	-	-	-	-	-	-
...																

Only the bits in red are necessary to define the ALU operation!

ALU

```
entity cpualu is
   port (
        clk : in STD_LOGIC;
        rst : in STD_LOGIC;
        op : in STD_LOGIC_VECTOR(4 downto 0);
        a : in STD_LOGIC_VECTOR(7 downto 0);
        b : in STD_LOGIC_VECTOR(7 downto 0);
        q : out STD_LOGIC_VECTOR(7 downto 0);
        f : out STD_LOGIC_VECTOR(3 downto 0)
        );
end cpualu;
```

Input A,B

Output

Flags

- Flags are always returned, but only used if the instruction is "cmp": control unit stores the flag in a flag register

ALU

```
r <=  a+b when op(4 downto 3)="01" and op(1 downto 0)="00" else
      sub(7 downto 0) when op(4 downto 3)="01" and op(1 downto 0)="01" else
      a and b when op(4 downto 3)="01" and op(1 downto 0)="10" else
      a or b when op(4 downto 3)="01" and op(1 downto 0)="11" else
      a xor b when op(4 downto 3)="10" and op(1 downto 0)="00" else
      not a when op(4 downto 0)="11000" else
      '0'&a(7 downto 1) when op(4 downto 0)="11001" else
      a(0)&a(7 downto 1) when op(4 downto 0)="11010" else
      a(7)&a(7 downto 1) when op(4 downto 0)="11011" else
      a(6 downto 0)&a(7) when op(4 downto 0)="11100" else
      "00000000";
```

- Multiplexer
- Can you recognize an instruction?
- Add is `001000rr ------rr`
- or `001100rr iiiiiiii`
- (in red op)
- Therefore we react is op(4..3)=01 and op(1..0)=00

ALU

```
sf <= sub(7);
zf <= not(sub(7) or sub(6) or sub(5) or sub(4) or sub(3) or sub(2) or sub(1) or
sub(0));
cf <= sub(8);
ovf <= (not a(7) and b(7) and sub(7)) or (a(7) and not b(7) and not sub(7));
```

```
f<=zf&ovf&cf&sf;
q<=r;
```

Combine flags in a vector

- Sign flag is bit 7 of the subtraction
- Zero flag obtained by oring
- Carry flag is bit 8 of the subtration (only 8 bits are returned but subtraction done on 9 bits to obtain the carry)
- Overflow flag: triggers when the result flips sign:
  - positive minus negative must give positive
  - negative minus positive must give negative
  - Otherwise it's an overflow

Top level CPU entity

```
entity cpu is
   generic(N : integer);
   port(
                  clk : in STD_LOGIC;
                  rst : in STD_LOGIC;
                  ext_in : in STD_LOGIC_VECTOR(7 downto 0);
                  ext_out : out STD_LOGIC_VECTOR(7 downto 0);
                  ram_we : out STD_LOGIC;
                  ram_address : out STD_LOGIC_VECTOR(N-1 downto 0);
                  ram_datawr : out STD_LOGIC_VECTOR(7 downto 0);
                  ram_datard : in STD_LOGIC_VECTOR(7 downto 0);
                  -- Only for debugging
                  dbg_qa : out STD_LOGIC_VECTOR(7 downto 0);
                  dbg_qb : out STD_LOGIC_VECTOR(7 downto 0);
                  dbg_qc : out STD_LOGIC_VECTOR(7 downto 0);
                  dbg_qd : out STD_LOGIC_VECTOR(7 downto 0);
                  dbg_instr : out STD_LOGIC_VECTOR(15 downto 0);
                  dbg_seq : out STD_LOGIC_VECTOR(1 downto 0);
                  dbg_flags : out STD_LOGIC_VECTOR(3 downto 0)
         );
end cpu;
```

External interface

Memory interface

CPU: instantiating the register bank

```
comp_regs: entity work.cpuregbank port map(
    clk=>clk,rst=>rst,
    rrd1=>instruction(9 downto 8),
    rrd2=>instruction(1 downto 0),
    rwr=>instruction(9 downto 8),
    rwren=>regwren,
    d=>wrdata,
    q1=>reg1out,q2=>reg2out);
```

1st register is "dst"
2nd register is "src"

write register is "dst "

Controls the write

1st and 2nd register output

- The instruction set is designed to help the control unit
- Source and destination always in the same location in the instruction
  - reg1out is the value in the "dst" register
  - reg2out is the value in the "src" register
- It does not hurt to "wire up" src and dst to the register bank, even if the instruction does not use src/dst (the control unit handles that)
- Write only occurs with a Move or an ALU instruction!

CPU: instantiating fetch/execute

```
comp_seq: entity work.cpusequencer port
  map(clk=>clk,rst=>rst,en=>'1',seq=>seq);
```

```
fetchh <=        '1' when seq="00" else
                      '0';
fetchl <=        '1' when seq="01" else
                      '0';
execute <=   '1' when seq="10" else
                      '0';
fetch <= fetchl or fetchh;
```

Helper signals - not mandatory but makes reading simpler

CPU: fetch instruction

```
comp_instrh: entity work.dffre generic map(N=>8)
   port map(clk=>clk,rst=>rst,en=>fetchh,d=>ram_datard,
   q=>instruction(15 downto 8));

comp_instrl: entity work.dffre generic map(N=>8) port
   map(clk=>clk,rst=>rst,en=>fetchl,d=>ram_datard,
   q=>instruction(7 downto 0));
```

- instruction is the 16-bit instruction read from memory
- It is 16-bit register realized by two 8-bit D flip-flops
- First flip-flop enabled on "fetch high"
- Second flip-flop enabled on "fetch low"

CPU: Program counter / instruction pointer

```
comp_ip: entity work.dffre generic map(N=>N) port
   map(clk=>clk,rst=>rst,en=>'1',d=>ipnext,q=>ip);

   ipnext <= ip+1 when fetch='1' else
                       ip when jump='0' else
                       jumpip;
```

- ip is the address where the instruction is read from
- N-bit D flip-flop
  - In the laboratory N is 5 bits (0...1F). It's a Design choice
- ipnext is the input of the D flip-flop

CPU: Program counter / instruction pointer

```
comp_ip: entity work.dffre generic map(N=>N) port
   map(clk=>clk,rst=>rst,en=>'1',d=>ipnext,q=>ip);

   ipnext <= ip+1 when fetch='1' else
                       ip when jump='0' else
                       jumpip;
```

- When to write: at each clock cycle!
- What to write:
  - Fetch (high or low): ipnext = ip+1
  - Exec:
    - ipnext does not change if the instruction is not a jump
    - ipnext changes if it is an absolute jump, or a conditional jump with the condition valid
      - (indicated by the signal jump)

CPU: Jump

Instructions	instruction(15..8)								Instruction(7..0)							
Jump Unsigned: JA, JAE, JB, JBE	opcode			$\overline{R} / I$	Jump type				immedite / reg							
jmp r	1	0	1	0	0	0	0	0	-	-	-	-	-	-	r	r
jmp i	1	0	1	1	0	0	0	0	i	i	i	i	i	i	i	i
je/jz r	1	0	1	0	0	0	0	1	-	-	-	-	-	-	r	r
je/jz i	1	0	1	1	0	0	0	1	i	i	i	i	i	i	i	i
jne/jnz r	1	0	1	0	1	0	0	1	-	-	-	-	-	-	r	r
jne/jnz i	1	0	1	1	1	0	0	1	i	i	i	i	i	i	i	i
ja r	1	0	1	0	0	0	1	0	-	-	-	-	-	-	r	r
ja i	1	0	1	1	0	0	1	0	i	i	i	i	i	i	i	i
jb r	1	0	1	0	1	0	1	1	-	-	-	-	-	-	r	r
jb i	1	0	1	1	1	0	1	1	i	i	i	i	i	i	i	i

- Unconditional jumps for 101x000 only
- Address is register or immediate: it is source defined before

CPU: Jump

```
jumpip <=       source(N-1 downto 0);
```

Where to jump if we were to do it

```
jump <= '1' when instruction(15 downto 13) = "101" and
   jumpconditionvalid='1' else
                         '0';
```

jump if there is a jump instruction and the jump condition is valid

```
jumpconditionvalid <=
   '1' when instruction(11 downto 8) = "0000" else
```

always valid (unconditional jump)

```
   -- je/jz
   '1' when instruction(11 downto 8) = "0001" and zf='1' else

   -- jne/jnz
   '1' when instruction(11 downto 8) = "1001" and zf='0'    else
```

Conditional jump when not zero: check the zero flag

```
   -- ja
   '1' when instruction(11 downto 8) = "0010" and zf='0' and
   cf='0' else
   -- jb
   '1' when instruction(11 downto 8) = "1011" and zf='0' and
   cf='1'
else    '0';
```

CPU: source (helper signal)

```
source <= reg2out when instruction(12)='0' else
                   instruction(7 downto 0);
```

- Many instructions use a "source": Move, ALU, Jump, out
- All these instructions use as source:
  - an immediate in the lower 8 bits of the instruction if R#/I=1
  - a register (output of the register bank) if R#/I=0
- R#/I is always instruction(12)
- source takes care of providing the right source (immediate or register) using a multiplexer
  - Either the immediate
  - or the 2nd output of register bank (src)

CPU: when to write to register

- regwren: helper signal indicating when to write to register (register file)
- Write to register only during execute cycle
- And:
  - The instruction is a Move
  - Or the instruction is ALU (but not "cmp")
  - Or the instruction is In

```
execute=1 and (
  (instr(15..13)=000 and instr(11)=0) or
  instr(15..13)=001 or
  (instr(15..13)=010 and instr(11..10) /= 01)
  instr(15..13)=011
)
```

CPU: what to write

- wrdata: helper signal containing the data to write:
  - Data for memory and register write
  - wrdata is connected to the register bank write input!
- What to write:
  - Move direct instruction: the data to write is "source" (either immediate or register)
  - Move from memory instr.: write what the memory chip provides
  - ALU instruction: write the output of the ALU
  - In instruction: write what is on the input of the external interface

```
wrdata =
  source when instr(15..10)=000X00
  ram_datard when instr(15..10)=000X01
  aluqout when instr(15..13)=001
  aluqout when instr(15..13)=010
  aluqout when instr(15..13)=011
  ext_in when instr(15..10)=110X01
```

CPU: memory interface

- What to write (put on the data bus)?
  - wrdata always
- What to put on the address bus?
- When to write?

CPU: memory interface

```
ram_address <=
   ip when fetch='1' else
   reg2out(N-1 downto 0) when instruction(15 downto 10)="000001" else
   instruction(N-1 downto 0) when instruction(15 downto 10)="000101" else
   reg1out(N-1 downto 0) when instruction(15 downto 10)="000010" else
   reg1out(N-1 downto 0) when instruction(15 downto 10)="000110" else
   (others=>'0');
```

- Address:
  - IP during the fetch cycles
  - The content of the src register or of the immediate for a move from memory
  - the content of the dst register for a move to memory
  - Otherwise zero
    - Explains why the first CPU lab showed always a zero on the address!

CPU: when to write to memory?

- Move to memory during the exec cycle:
  - mov [b],23h

```
ram_we <=
   '1' when execute='1' and instruction(15 downto 13)="000"
   and instruction(11 downto 10)="10"
   else '0';
```

CPU: External interface

- External interface output is an 8-bit D flip-flop
- When to write: during exec when instruction is "out"
- What to write: source (i.e. register or immediate)

```
comp_regextout :
   entity work.dffre generic map (N=>8)
   port map(clk=>clk,rst=>rst,en=>ext_wren,
   d=>source,q=>ext_out);

   ext_wren <=
        '1' when execute='1' and instruction(15 downto 13) =
        "110" and instruction(11 downto 10)="00"
        else '0';
```

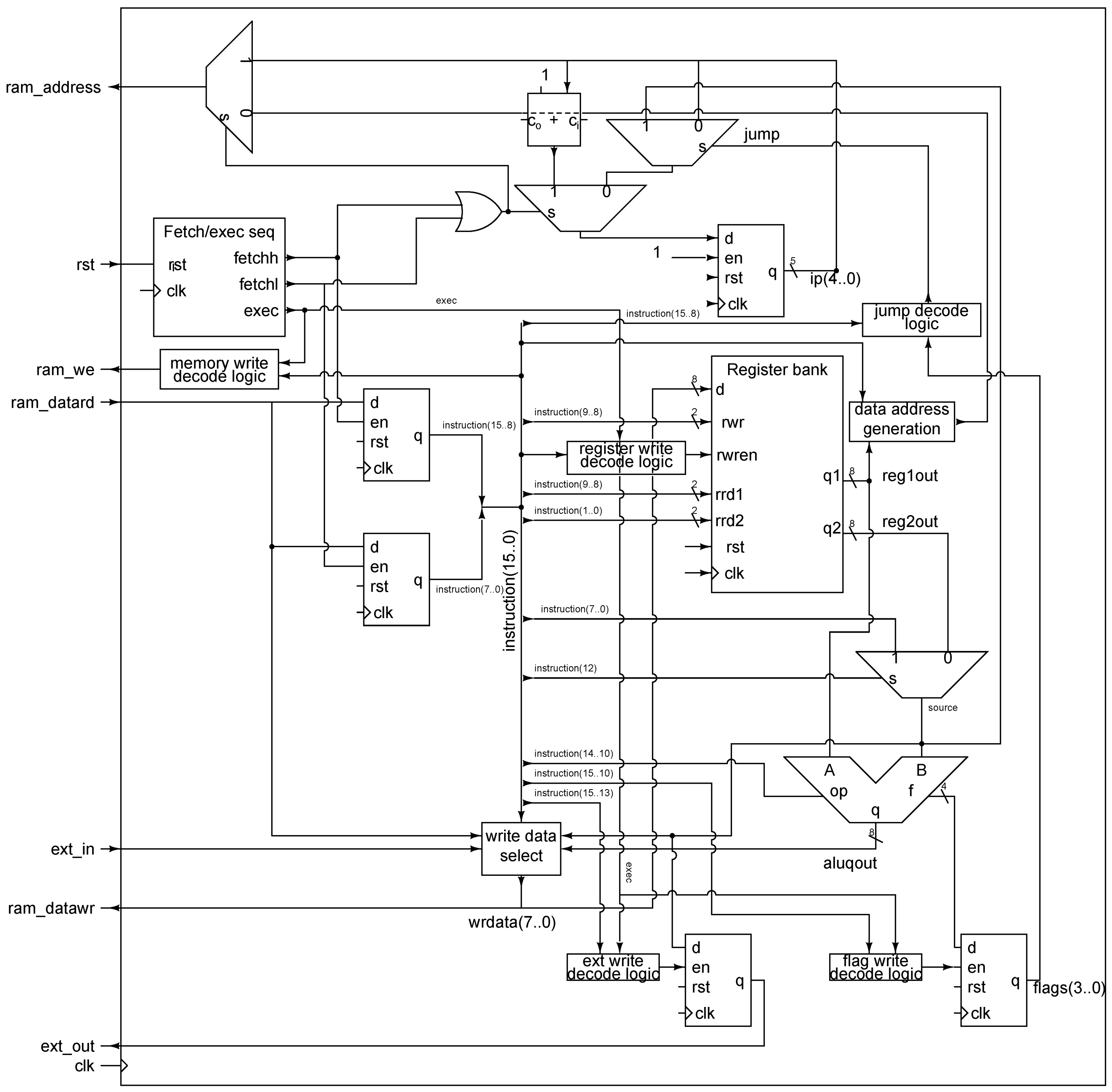
CPU: flags

- SF, CF, OF, ZF are stored in a 4-bit D flip-flop
- When to write: during exec and "cmp" instruction
- What to write: the flag output of the ALU

```
comp_flags:    entity work.dffre
   generic map(N=>4)
   port map(clk=>clk,rst=>rst,en=>flagwren,d=>alufout,q=>flags);

flagwren <= '1' when execute='1' and instruction(15 downto
   13)="010" and instruction(11 downto 10)="01"
   else '0';
```

CPU


ram_address
rst
ram_we
ram_datard
ext_in
ram_datawr
ext_out
clk
Fetch/exec seq
rst
clk
fetchh
fetchl
exec
memory write decode logic
1
c_o + c_i
jump
s
1
0
d
en
rst
q
clk
ip(4..0)
exec
instruction(15..8)
jump decode logic
Register bank
d
rwr
rwren
rrd1
rrd2
rst
clk
q1
q2
reg1out
reg2out
data address generation
register write decode logic
instruction(9..8)
instruction(1..0)
instruction(7..0)
instruction(12)
instruction(14..10)
instruction(15..10)
instruction(15..13)
instruction(15..0)
source
A
B
op
f
q
aluqout
write data select
wrdata(7..0)
ext write decode logic
flag write decode logic
flags(3..0)

Jump decode logic

```
instr(15..13)=101 and (
  instr(11..8)=0000 or
  (instr(11..8)=0001 and zf=1) or
  (instr(11..8)=1001 and zf=0)
)
```

Register write decode logic

```
execute=1 and (
  (instr(15..13)=000 and instr(11)=0) or
  instr(15..13)=001 or
  (instr(15..13)=010 and instr(11..10) /= 01)
  instr(15..13)=011
)
```

Memory write decode logic

```
execute=1 and instr(15..10)=000X10
```

External write decode logic

```
execute=1 and instr(15..13)=110
```

Flag write decode logic

```
execute=1 and instr(15..10)=010X01
```

Write data select

```
wrdata =
  source when instr(15..10)=000X00
  ram_datard when instr(15..10)=000X01
  aluqout when instr(15..13)=001
  aluqout when instr(15..13)=010
  aluqout when instr(15..13)=011
  ext_in when instr(15..10)=110X01
```

Data address generation

```
address =
  reg1out when instr(15..10)=000001
  instr(7..0) when instr(15..10)=000101
  reg1out when instr(15..10)=000010
  reg1out when instr(15..10)=000110
```

Summary

- Brief insight into the processor architecture
- Basic understanding of the control unit function:
  - deciding what and when to "store" data in register
- Sufficient knowledge to perform the coursework assignment on:
  - Modifying the instruction set (adding instruction)
  - Tracing the state of key signals for a given instruction