CMPT 250: Weeks 8-9 (Oct 24 to Nov 5)

1. INSTRUCTION SET ARCHITECTURES (Continued)

1.1. PROBLEM SPECIFICATION AND ANALYSIS (Continued)

Last week's notes described how an instruction set formed the basis of the functional specification of a CPU. Different possible designs for that CPU can be obtained by examining the instruction set from different viewpoints. This analysis is continued here with a summary describing how instructions may be examined:

1. By grouping instructions according to function (see last week's notes).
2. By fixing the number of explicit operands (see last week's notes). One-operand and register-memory machines suggest the importance of having many work registers in the CPU. There are two ways to achieve this, and the corresponding instruction styles are as follows:

- **0-operand architecture**

```
PUSH  A          S[TOP] <- M[A], TOP <- TOP-1
PUSH  B          S[TOP] <- M[B], TOP <- TOP-1
ADD              TOP <- TOP+1;
                 S[TOP+1]<-S[TOP+1]+S[TOP]
PUSH  C          S[TOP] <- M[C], TOP <- TOP-1
PUSH  D          S[TOP] <- M[D], TOP <- TOP-1
SUB              TOP <- TOP+1;
                 S[TOP+1]<-S[TOP+1]-S[TOP]
MPY              TOP <- TOP+1;
          S[TOP],S[TOP+1]<-S[TOP+1]+S[TOP]

POP   F          TOP <- TOP+1;
                 M[F] <- S[TOP]
POP   F+4        TOP <- TOP+1;
                 M[F+4] <- S[TOP]
```

In this case the registers are provided by an internal stack memory; that is, a memory implemented as an internal, hardware push-down-store.

- **register-register architecture**

```
LW  s2, A        s2 <- M[A]
LW  s3, B        s3 <- M[B]
LW  s4, C        s4 <- M[C]
LW  s5, D        s5 <- M[D]
ADD s0, s2, s3   s0 <- s2 + s3
SUB s1, s4, s5   s1 <- s4 - s5
MPY s6, s0, s1   s6,s7 <- s0*s1
SW  s6, F        M[F] <- s6
SW  s7, F+4      M[F+4} <- s7
```

  Here the set of work registers is provided by a register-file; that is, a memory whose architecture permits simultaneous retrieval and storage from its internal locations.

3. By classifying instructions on the basis of the way the operands are specified within the instruction format; that is, by judicious use of different addressing modes:
   a. **Implied**: The operands are not part of the instruction format.
   b. **Immediate**: The value of the operand is embedded in the instruction.
   c. **Register**: The operands are provided in registers that are specified in the address field of the instruction.
   d. **Direct**: The location of the operand is embedded in the instruction.
   e. **Indirect**: The location of the address of the operand is embedded in the instruction.
   f. **Indexed**: The location of the operand is computed from values provided within the instruction and in an *index* register. The effective address is obtained by adding a displacement to a base address.
   g. **Relative**: The PC provides the base address, with the displacement embedded in the instruction.

2. REGISTER-REGISTER CPU DESIGN

2.1. GENERAL DESIGN STRATEGY

1. For each instruction specified:
    - Define a unique opcode
    - Define an instruction format
    - Define an ASM to define the steps to be performed when the instruction is executed, called the *Execute Cycle* for the instruction. The execute cycle may consist of two phases: *execute* and *write back*.
2. Define an ASM to control the activities performed during instruction retrieval, called the *Fetch Cycle*. The fetch cycle may consist of two phases: *instruction fetch* and *operand fetch*.
3. Develop a structural model for the "datapath" component of the sequential circuit that constitutes the CPU using the algorithmic specifications given by the set of ASMs developed in the previous steps. Note that the controller (sequencer and control point selector) will be represented by a "black box" in this diagram.
4. Resolve the details of each component using the standard tools and techniques.

2.2. REGISTER FILES

A register file is a set of registers connected in such a way that one or more registers can be enabled for retrieval and possibly one or more registers enabled for storing values, all simultaneously. For each register that must be accessed, there must be a corresponding input address that selects the desired register, and a control point that enables either retrieval or storage to the address selected. Typical register files may have one or two read-address inputs and usually a single write-address output, and are referred to as *one-read-one-write* or *two-read-one-write* register files.

The behavioral model for a two-read-one-write register file is:

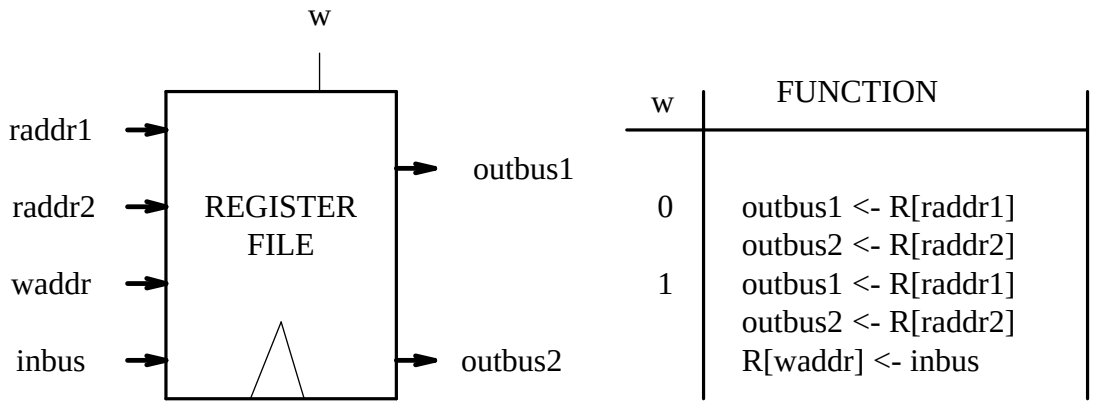


Figure 2-1: Behavioral specification of a register file

3. A DESIGN EXAMPLE

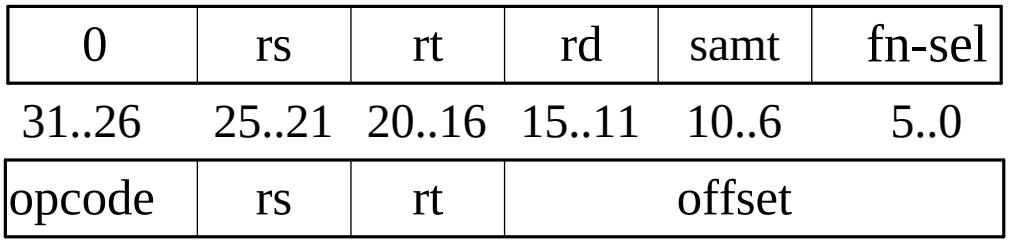
3.1. INSTRUCTION SET SPECIFICATION

The MIPS machine is an example of a register-register machine. It is described in some detail in the book "Computer Organization and Design", by Hennessy and Paterson (see related readings). The following example illustrates how the technique for sequential machine design described in class can be used to develop the architecture discussed in that text.

As with all designs we begin with a proposed instruction set. In this case it is a small subset of the MIPS assembly language; specifically, the instructions described on the next page.

The instructions are divided into two formats:

1. **R-format** is used for register-register instructions. All such instructions have the same value in the opcode field: 0. Each such instruction is distinguished from another by the value in the **`fn-sel`** field of the instruction.
2. **I-format** is used for register-memory instructions. These instructions provide two types of addressing mode: direct and indexed.



add	32	R[rd] <- R[rs] + R[rt]
sub	34	R[rd] <- R[rs] - R[rd]
and	36	R[rd] <- R[rs] and R[rt]
or	37	R[rd] <- R[rs] or R[st]
slt	42	IF R[rs]<R[rt] then R[rd] <- 1 ELSE R[rd] <- 0
lw	35	R[rt] <- M[offset + R[rs]]
sw	43	M[offset + R[rs]] <- R[rt]
beq	4	IF R[rs]= R[rt] then PC <- PC + offset

Figure 3-1: MIPS instruction subset and format

Note the location of the fields used to specify each register operand. Since 5 bits have been allocated for each register, 32 registers can be addressed. These registers are represented symbolically in MIPS assembly language as follows:

```
REGISTER          SYMBOLIC NAME
--------          ----------------
0:                $zero
8..15:            $t0, $t1,..., $t7
16..23:           $s0, $s1,..., $s7
24,25:            $t8,$t9
<13 more registers>
```

To determine the machine instruction, given its symbolic representation, use the information provided by the instruction set format definition. For example:

```
EXAMPLE 1:
MIPS assembly instruction: sub $t2, $t2, $s2

IR(31..26) = 000000  (R-type instruction)
IR(25..21) = 01010   (rs = $t2 = reg 10)
IR(20..16) = 10010   (rt = $s2 = reg 18)
IR(15..11) = 01010   (rd = $t2 = reg 10)
IR(10..6)  = 00000   (no shift required)
IR(5..0)   = 100010  (fn-sel "sub" = 34)

EXAMPLE 2:
MIPS assembly instruction: sw $t0, 0($s4)

IR(31..26) = 101011  (I-type instruction)
IR(25..21) = 01000   (rs = $s4 = reg 20)
IR(20..16) = 10100   (rt = $t0 = reg 8)
IR(15..0)  = 0000 0000 0000 0000 (offset = 0)
```

3.2. FORMAL SPECIFICATION OF THE CPU

ASSUMPTION: The processor design ia based on the Von Neumann Architecture. The **von Neumann Model** (also called the Princeton Model) is the most common computer model in existence and more than 90% of exsting computers are based on this model. With this model, instructions and data share the same memory, data paths and processing components.

The adoption of this model has the advantage that it simplifies the hardware design. However, because programs and data share the same components, it will be necessary for the processor to keep track of whether it is fetching an instruction or fetching data when it performs a memory retrieval. As a consequence, the task of executing an instruction is broken into two steps - a *fetch cycle* and an *execute cycle*.

The Fetch Cycle: The purpose of the fetch cycle is to retrieve an instruction and place it in a suitable location within the processor for decoding. This location is

called the *Instruction Register* or IR. This cycle must therefore keep track of where the next instruction is found, and this is achieved using a register called a *program counter* or PC. Finally, some decoding of the instruction may also take place, usually to identify the operands of the instruction (operand fetch).

The ASM diagram for the the fetch cycle of the μMIPS machine whose instruction set was provided previously is given by ASM blocks F1 (instruction fetch) and F2 (operand fetch) on the following page.

The Execute Cycle: Once an instruction is in the IR, it can be interpreted by the controller. Specifically this interpretation involves:

1. Identifying the instruction to be executed.
2. Retrieving from memory any operands required for execution. This is distinct from the oerand fetch phase in that an effective address must be calculated (execute phase) and the data retrieved
3. Performing a sequence of register transfer expressions, which collectively define what it means to execute the instruction (execute phase).
4. If a result has been computed it needs to be stored in the register file (write-back phase).
5. Recognizing completion of an instruction, and initiating another fetch cycle.

To perform an execute cycle for a given instruction requires that the processor datapath provide the necessary data flow paths, registers, and combinational logic as defined in the set of register transfer statements which characterize the execution of the instruction. The ASM diagrams for the fetch and execute cycles for the μMIPS machine are given on the next two pages.

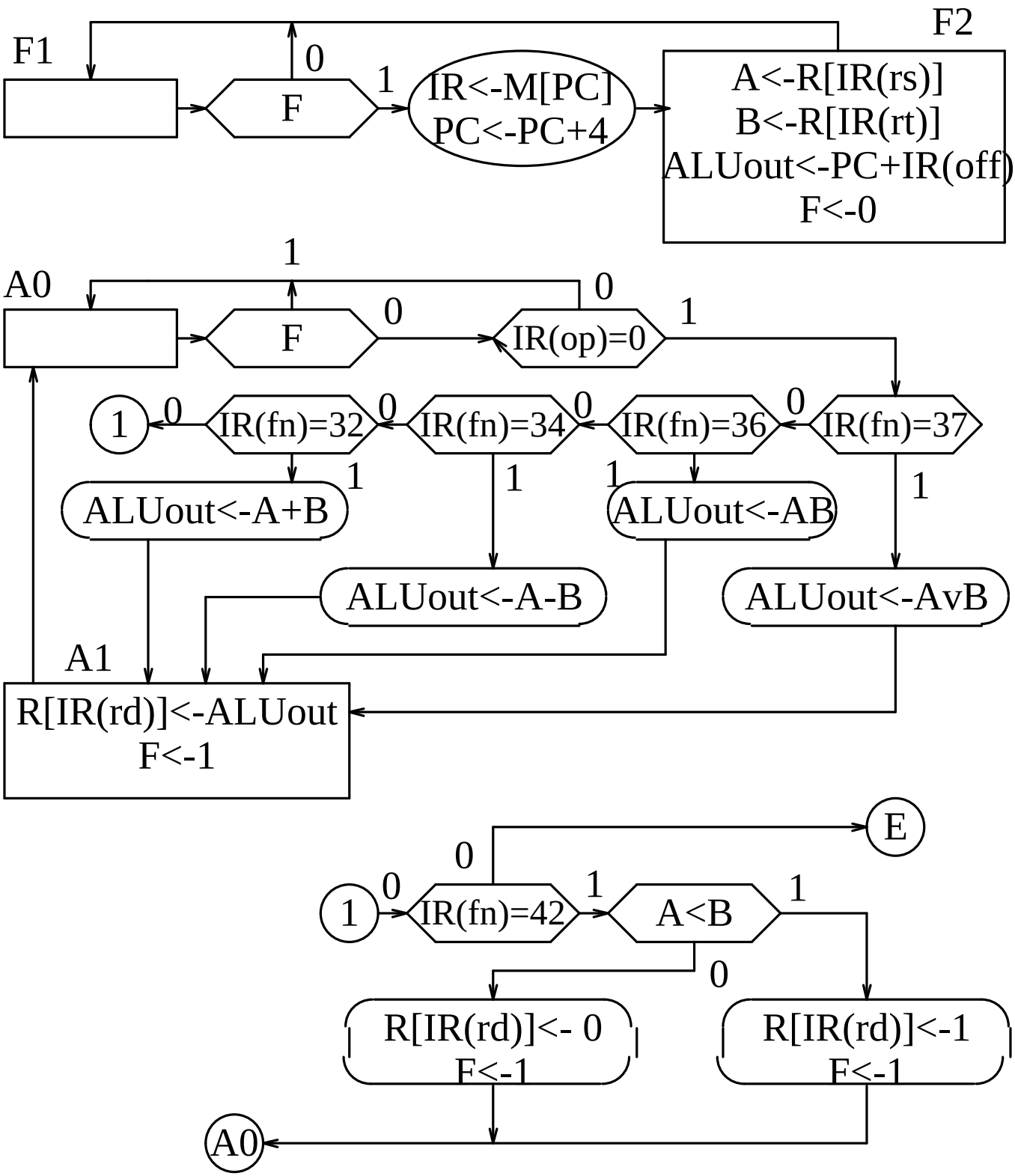


Figure 3-2: Fetch and Execute ASM Diagrams Part I

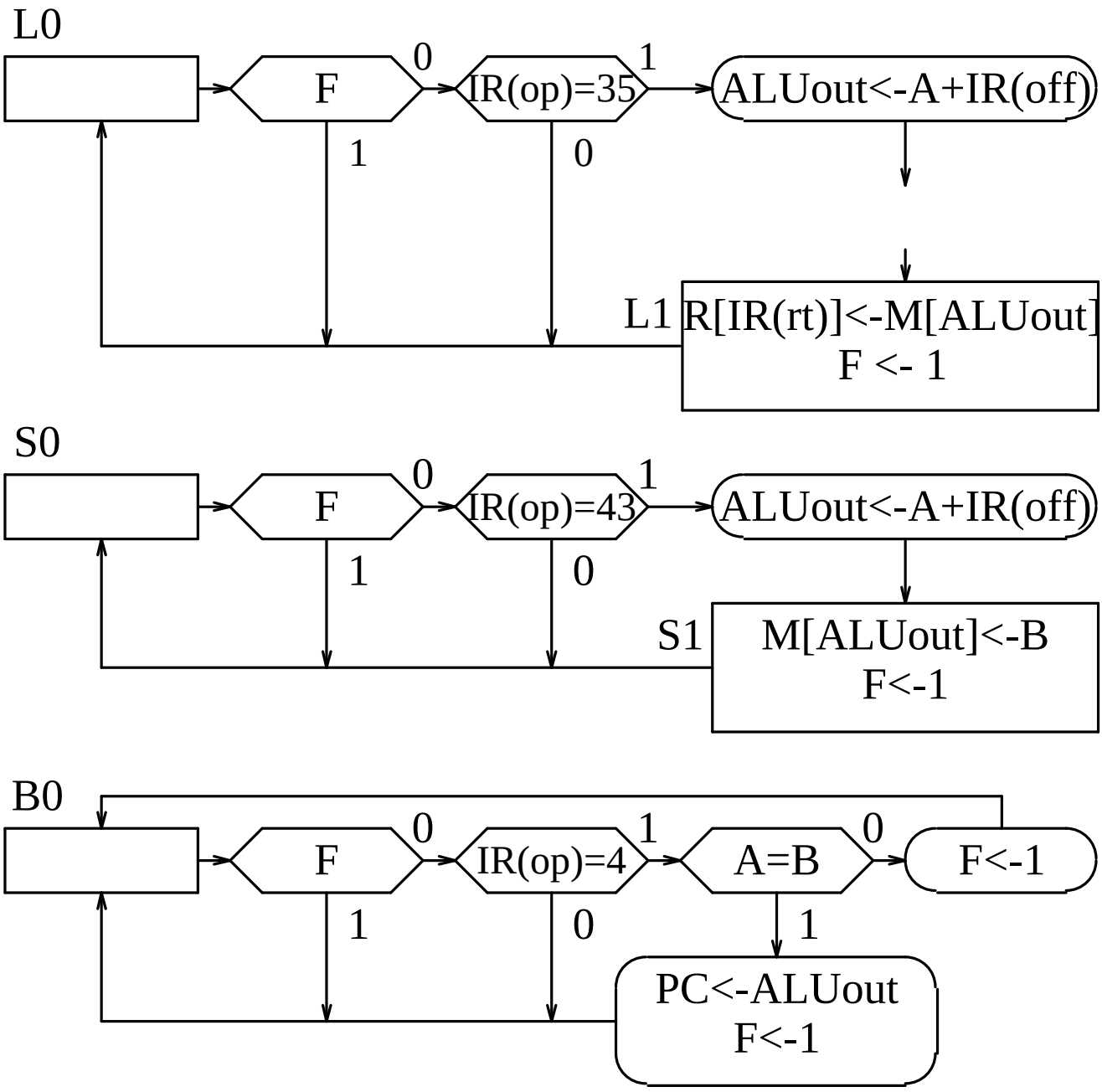


Figure 3-3: Execute ASM Diagrams Part II

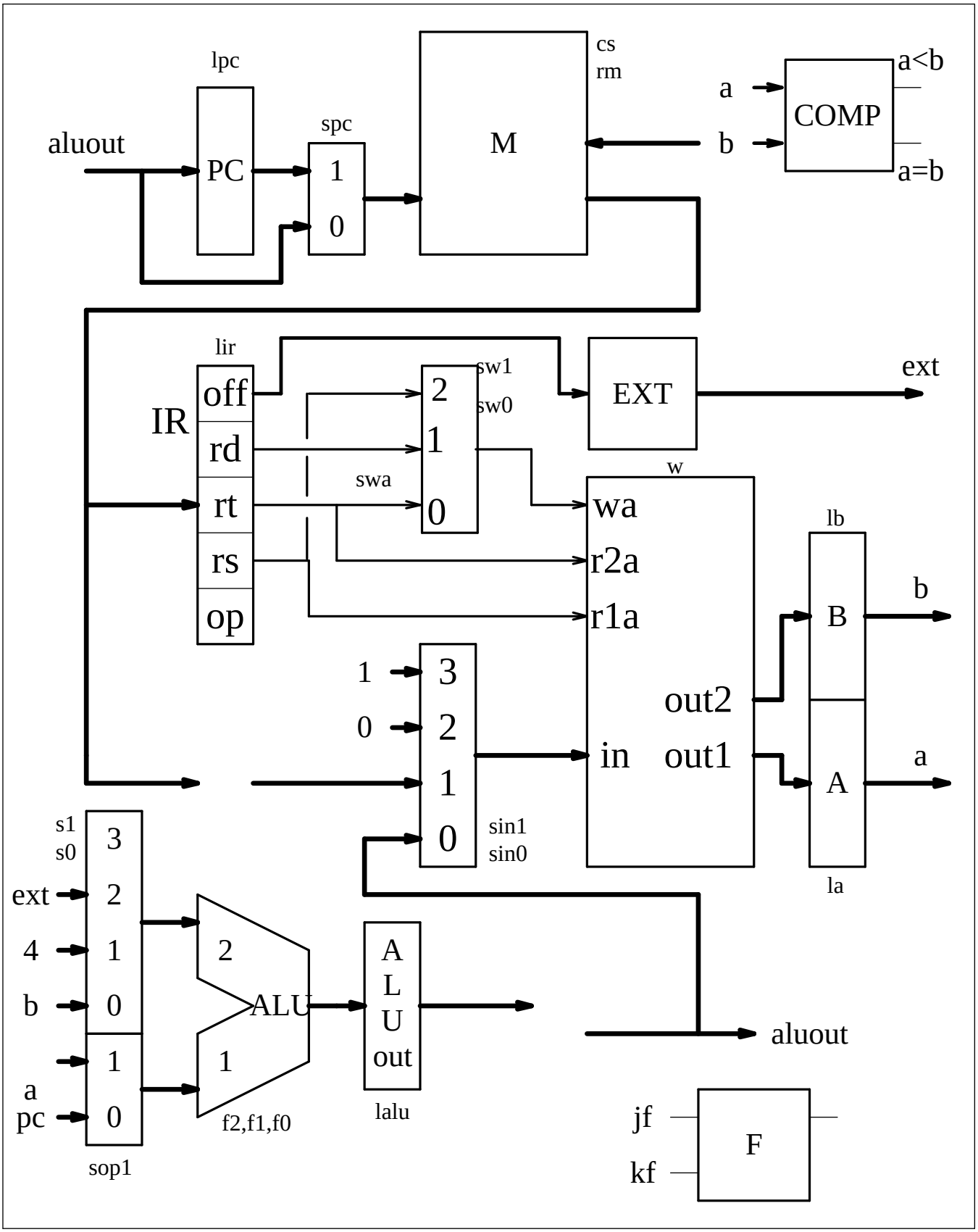
By defining the instruction set and specifying the ASM for each instruction, we have sufficient information to define the internal organization of the processor. Although it is possible to describe the machine behaviour with a single large ASM diagram, it is convenient to repesent each the fetch cycle and each instruction's execute cycle by a separate ASM and view the collection as describing a set of machines that are running independently, sharing common components to communicate when one machine has completed a task and when another can start.

From the ASM Diagrams the register requirements and the inter-register data transfer requirements can be defined. The register transfer statements are grouped by destination:

```
PC:     PC <- ALUOut
        PC <- PC + 4

IR:     IR <- M[PC]

A:      A <- R[IR(rs)]

B:      B <- R[IR(rt)]

ALUOut: ALUOut <- PC + IR(offset)
        ALUOut <- A + IR(offset)
        ALUOut <- A op B
                op = +, - , and, or

R[..]:  (storage)
        R[IR(rt)] <- M[ALUout]
        R[IR(rd)] <- ALUOut
        R[IR(rd)] <- 0
        R[IR(rd)] <- 1

        (retrieval)
        A <- R[IR(rs)]
        b <- R[IR(rt)]

M[..]:  (storage)
        M[ALUOut] <- B

        (retrieval)
        IR <- M[PC]
        R[IR(rt)] <- M[ALUOut]
```

The register file required must therefore be able to retrieve two values and store one value simultaneously; that is, a 2-read, 1-write register file is required.

From the register transfer statement analysis it is evident that the **PC**, **ALUOut**, and both address inputs of the register file have multiple sources. The resulting architecture for the processing component of the CPU is given on the next page.


lpc
aluout
PC
spc
1
0
M
cs
rm
a
b
COMP
a<b
a=b
lir
IR
off
rd
rt
rs
op
2
1
0
sw1
sw0
swa
EXT
ext
w
wa
r2a
r1a
lb
B
b
1
0
3
2
1
0
sin1
sin0
in
out2
out1
A
a
la
s1
s0
ext
4
b
a
pc
sop1
ALU
f2,f1,f0
A
L
U
out
lalu
aluout
jf
kf
F