EvoproK:

EvoproK is an evolving processor whose first level of evolution is this course. Later on it might evolve further in future courses into different species depending on applications. Though the option of studying architecture of MIPS (or any Opensource processor) architecture is there, but we have chosen the path of re-inventing the wheel (in fact the processor). The main reason behind that is the change of perspective "when we try to understand things" and "when we try to make things". The designer's mindset is more natural way of understanding things in my view. We shall implement this processor at various stages of its evolutionary path and be witness of how things vary by taking different trade-offs. We might eventually end up with something close to MIPS as MIPS is the design discussed in the sources we are taking guidance from.

Let's start with a very basic subset of instructions that we shall support. We want to incorporate **ADD**, **SUB**, **MULT** and **AND** instructions that can do these operations by taking two operands from a register file and storing the results in the same register file. Let's assume that we have 32 registers each 32 bit wide. Let's keep the size of each instruction to be 32-bits. So we can support these instructions in the following format.

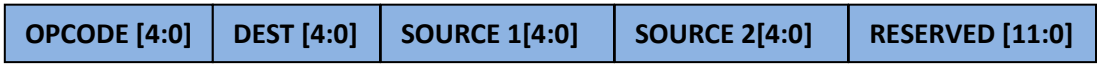

Figure 1: Arithmetic instructions format

The hardware for these instructions can be quite straightforward. An overview of architecture is shown in Figure 2. We will discuss it left to right. We assume that we have separate program and data memories.

First of all we need instructions in a sequence. We can store the instructions in program memory and a program counter (**pc**) that increments every cycle can be used to address program memory. The program memory will in return present us with corresponding sequence of instructions.

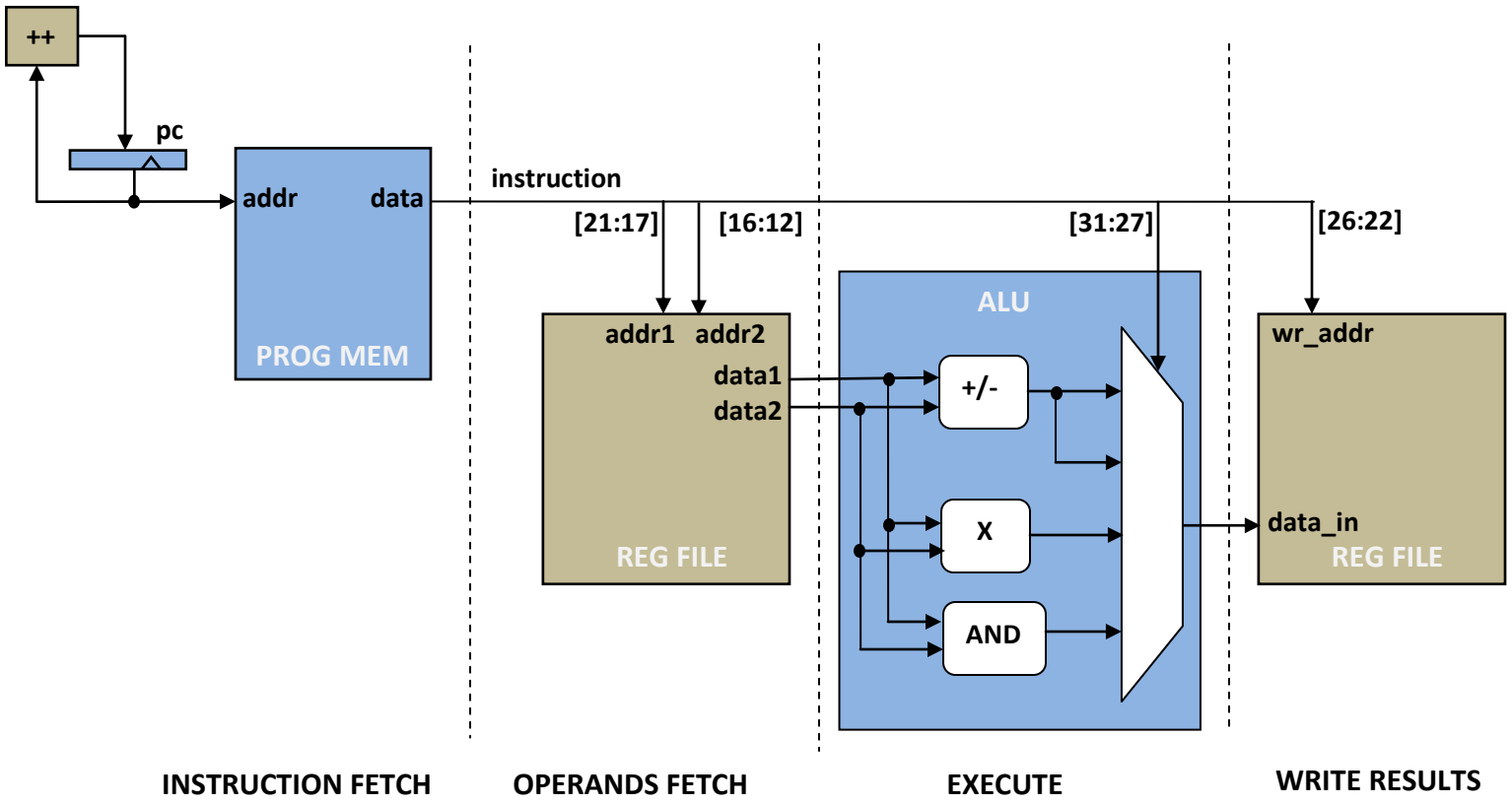

Figure 2: Overview of architecture

The register file has two read ports and one write port. The source addresses in the instruction can be fed to read address ports of register file to get the operands. The operands can be broadcasted to Arithmetic and Logic unit (**ALU**) and **Opcode** in the instruction can be used to select among the outputs of functional units. The output can then be forwarded to the write port of register file and **Dest** in the instruction can be used as write address of reg file. We have shown REG FILE twice in the diagram as this diagram is intended to show the sequence of events from left to right. It should be kept in mind that in fact it's one register file. We will continue with this convention in future as well.

So we have incorporated support for doing some basic operations on registers. It will be useful only if we add support for transferring data from data memory to registers and vice versa. We call these instructions **LOAD** words and **STORE** word represented by **LW** and **SW**. We stick to RISC architecture i.e. use only LOAD/STORE for memory transactions.

Both load and store need a register address and a memory address. We have already allocated 5 bits for Opcode which means that we can support 32 different instructions. So far arithmetic and memory instructions have consumed just 6 of those. Below is instruction format for Load and Store.

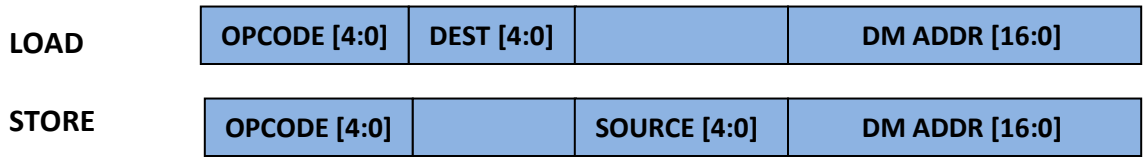

Figure 3: Load/Store instruction Format

The source register address in Store instruction is aligned with **SOURCE1** field in Arithmetic instructions. This way the **dataout1** from register file is our required data and we do not need to add any extra hardware.

Hardware Architecture for supporting Load and Store instructions is shown in Figure 4 on next page. The 16 LSBs of instruction are used as Data memory address. In case of Load the **data_out** from memory is written in Reg file. A mux selects at the **data_in** port of Reg file to select between data from ALU (in case of Arithmetic instructions) and Data Memory (in case of Load instruction). The **write_en** of Reg file is enabled if the instruction is not a store instruction as all other instructions so far write into Reg file. The **dataout1** from Regs file is tied to **data_in** port of data memory. Data memory is written when **wr_en** is asserted i.e. only when the instruction is Store.

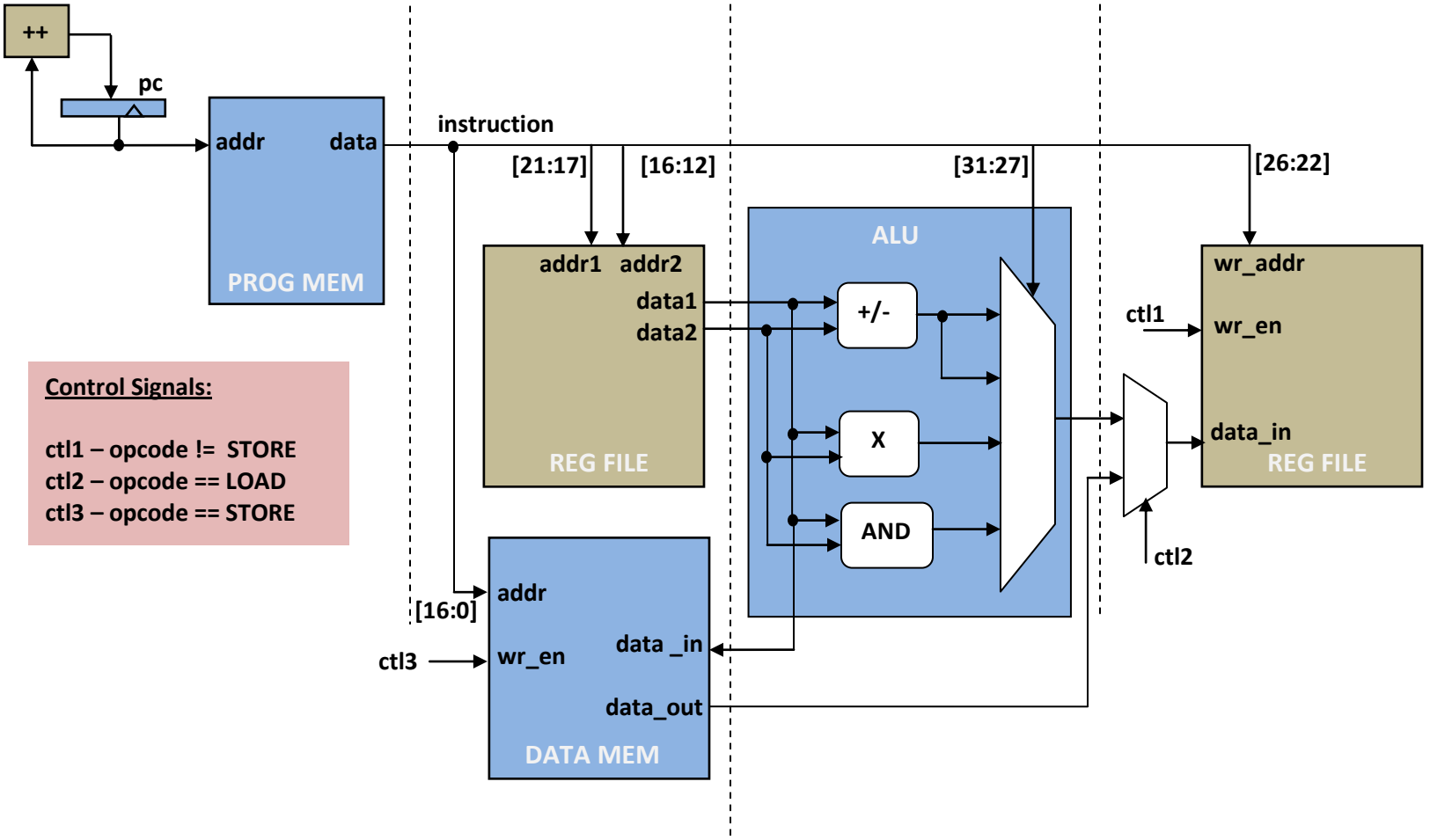

Figure 4: Architecture with Load/Store support