Support for Branch

The processor we have designed so far should be able to do simple sequential calculations. It will be a slow one though because everything is done in one cycle so the critical path will be quite long. But before making it fast, let's eliminate some important weaknesses. Real programs support constructs like **IF-ELSE, SWITCH-CASE** and **LOOPS.** All of these constructs check some condition and instead of executing next instruction jump to some other instruction.

So we introduce some new instructions which should enable us to do that. We introduce three instructions using register 0 as our condition register.

- BRZ: Branch if register0 is zero
- BRN: Branch if register0 is not zero
- BRP: Branch if register0 is positive or zero

If the condition is true Branch will take us to an offset relative to **pc.** This is called **PC-relative offset.** To support we define instruction format for branch and then add hardware support for that.

OPCODE [4:0]	**Offset [26:0]**

Figure 1: Instruction format Branch instructions

This will affect the instruction fetch part of processor that deals with **PC.** So instead of just incrementing we can use a mux to select whether we need to increment by an offset or 1. The select line of the mux is asserted only if the instruction is branch instruction and branch condition is true.

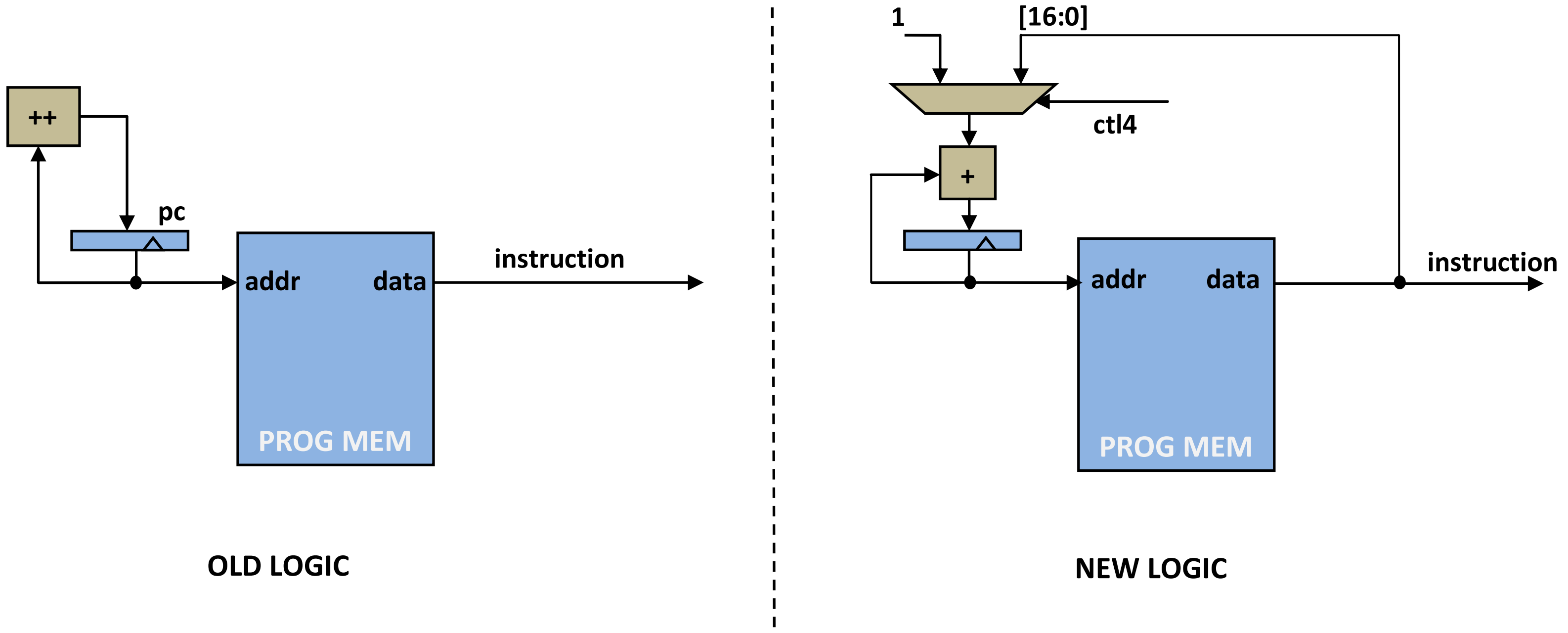

Figure 2: Hardware support for Branch instructions

The **LUI** and **LLI** load the upper and lower 16 bits of a register. The instruction format can be as follows.

Opcode [4:0]	Dest [4:0]	Unused [4:0]	Immediate value [15:0]

Figure 3: LUI and LLI instruction format

As these instructions modify half of register we need to incorporate this functionality in reg file. Each register should support loading of upper or lower half. This can be done by supporting two enables as shown in figure below. Normally both can be driven by same decoded **wr_en[i]** but in case the instruction is **LUI** or **LLI** we can assert control signals **LUI_en** or **LLI_en** respectively in addition to **wr_en[i]**. Note that even in case of **LUI** and **LLI,** we need to assert **wr_en[i]** which is controlled by **ctl1**. The upper half is enabled if there is **wr_en[i]** for **i**$^{\text{th}}$ register is asserted and the instruction is not **LLI**, since if the instruction is LLI that means lower half is to be written. The design of write enable for lower half is similar.

In addition to that we need to add a mux at reg file **data_in** (in write results stage) to select immediate value instead of **ALU** or **Data memory output** as shown in red below. The immediate value is 16 bits wide, so we replicate it twice to make 32 bits. The enable signals will ensure that only 16 bits are written on specified register. The control signal **ctl7** will be asserted if the instruction is either **LUI** or **LLI.**

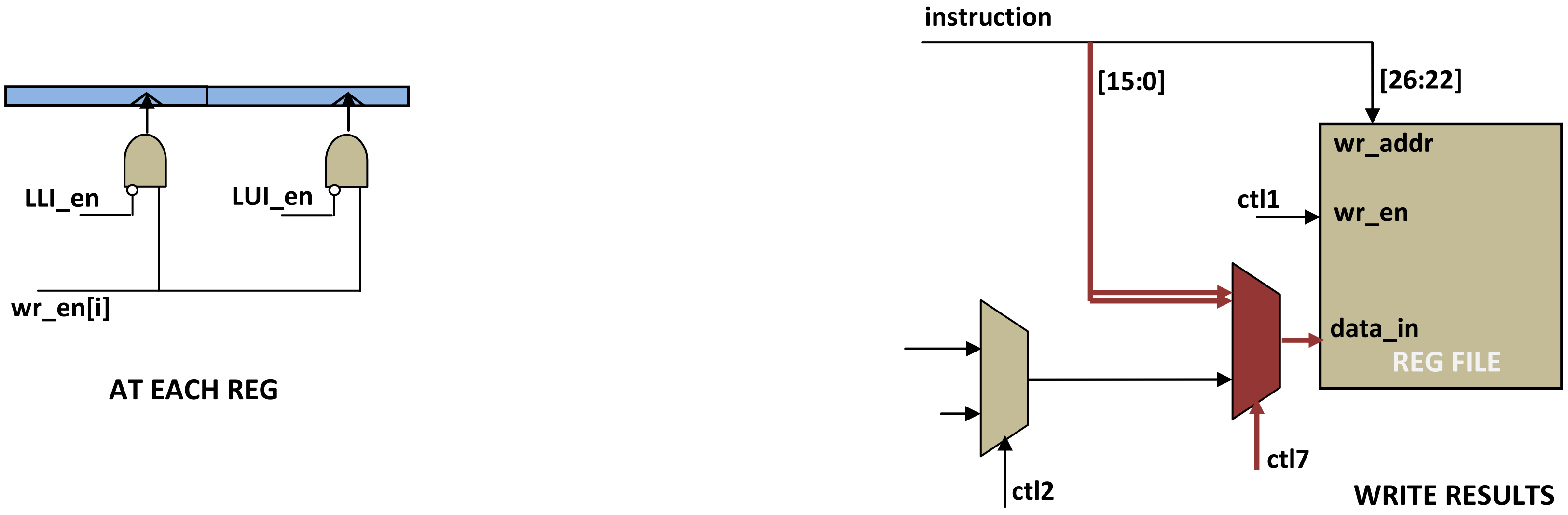

Figure 4: LUI and LLI hardware support

Next instruction to be supported is **ADDI.** Unlike normal addition, this instruction takes one operand from register and the other operand is immediate value in the instruction.

Opcode [4:0]	Dest [4:0]	Source [4:0]	Immediate Value [16:0]

Figure 5: Add Immediate instruction format

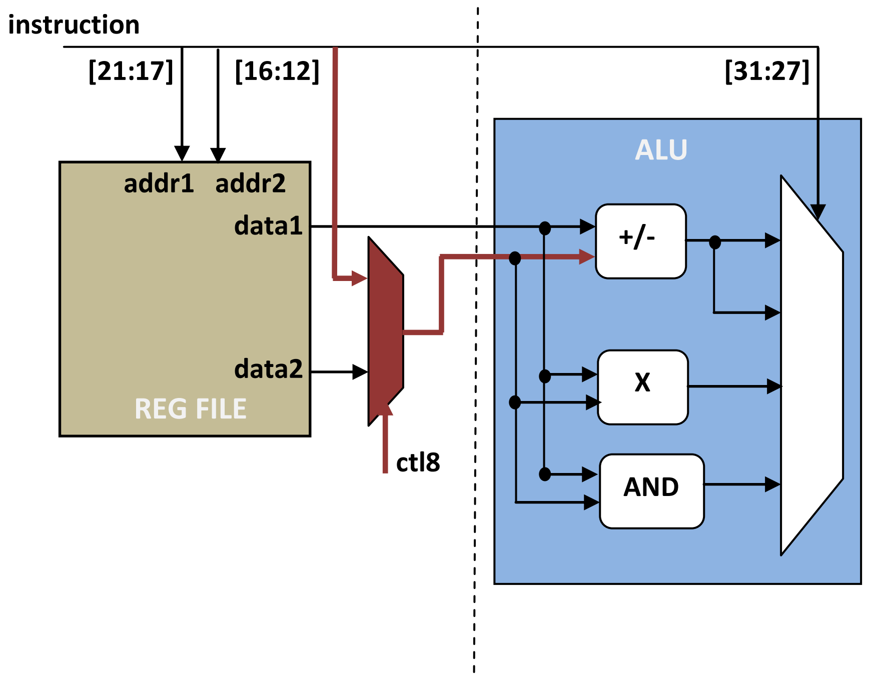

Figure 6: Immediate Operand support

To support second operand from within instruction instead of reg file we need to add a mux at second input operand of **ALU** as shown in red**.** This mux is sufficient to add support for not only **ADDI** but also for **SUBI**, **MULTI** and **ANDI.** The control signal **ctl8** needs to be controlled accordingly. Since the output of ALU units is selected using a mux, the **opcode** for immediate instructions should be chosen carefully such that **instruction[31:27]** values of immediate and respective normal instructions are same.

Subroutine call and return support

Subroutines or functions are integral part of any programming language making it possible to re-use same part of code time and again without the need of re-writing it. We covered a few examples of these subroutines in the class and it became very evident that if we incorporate the instructions **JAL** and **JR** we can support subroutine call and stack. In addition to that we also noticed that to support the conditional constructs it is required to have a **J (JUMP)** instruction. Quite often in loops and otherwise we require immediate values to be stored in a register or to take one of the operands as immediate value instead of taking from register.

Instruction	Description
J	**JUMP** to an offset relative to **PC**
JR	**JUMP** to an address in a register
JAL	**JUMP and Link:** Jump to an offset relative to **PC** and save **PC** to return address register (**RA**)
LUI	**Load upper immediate:** Load 16 bits into MSBs of a register
LLI	**Load lower immediate:** Load 16 bits into LSBs of a register
ADDI	**Add immediate** value in a register value. Similar immediate instructions for other arithmetic and logic operations.
LW with register value as memory address	**LOAD** from data memory. The data memory address is value in a "Source Register". The data from that memory address is written in a "Destination register".
SW with register value as memory address	**STORE** into data memory. The data from a "Source Register" is written at Data memory address equal to a "Destination Register"

Table 1: Instructions to support subroutine call and return

Now let's add hardware support for these instructions one by one. Again we will first define the Instruction format and then add hardware support. The **J** instruction is very similar to branch except that it does not check any condition. Below is the instruction format

OPCODE [4:0]	**Offset [26:0]**

Figure 7: Jump instruction format

Looking back at Figure 2 we can see that hardware already supports jumps to offsets relative to **PC.** We just need to modify **ctl4** logic (which we intentionally left in that figure) to ensure that in case of **J** instruction offset is added to **PC.**

To support **JR** we need to read a value from a register and copy that value into the **PC.**

OPCODE [4:0]	Unused [4:0]	Source [4:0]	Unused [16:0]

Figure 8: JR instruction format

Instead of using field adjacent to Opcode we have preferred skipping five bits. This makes **Source** field aligned to **Source1** field in ALU instructions that we covered earlier. As a result we can use data from register file directly as jump value. But we still need to add support to copy that value to **PC.** We can add a mux to that as shown in red below. The control signal **ctl5** is asserted when instruction is **JR.**

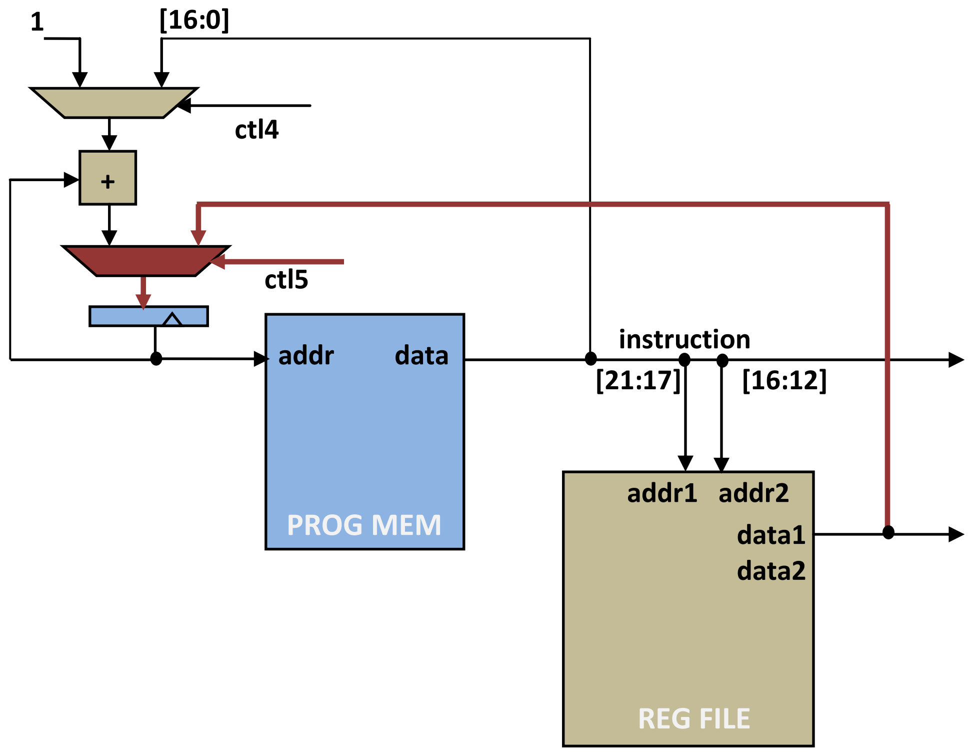

Figure 9: Hardware support for JR

The next instruction is **JAL** which is very similar to **J** except that it ensures that **PC** is stored in a return address register. We shall call this register **RA.** We can dedicate a register in our register file for that purpose. All we need to do is to change the **Write Logic** for that specific register. Instead of being written by normal instructions that register is only written when **JAL** is executed.

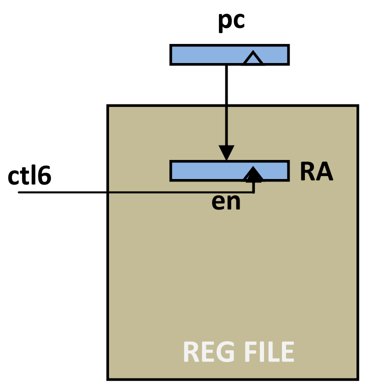

Figure 10 : JAL support in Reg file

The instruction format of **JAL** is same as that of **J** instruction.

To support **LW with register value as memory address** two registers are used. **Source** register specifies the value of data memory address. The data read from that address will be saved in the **Dest** register.

Opcode [4:0]	Dest [4:0]	Unused [4:0]	Source [4:0]	Unused [11:0]

Figure 11: LW with register address instruction format

The **Source** field is already tied to register file's read port for arithmetic operations. The respective value coming out of reg file i.e. **data2** needs to be tied to Data memory address. Data memory address was earlier connected to **instruction [16:0]** to support LW with immediate address. So we add a mux to select between the two sources as shown in red in the figure. A control signal **ctl9** will ensure based on opcode which data to select.

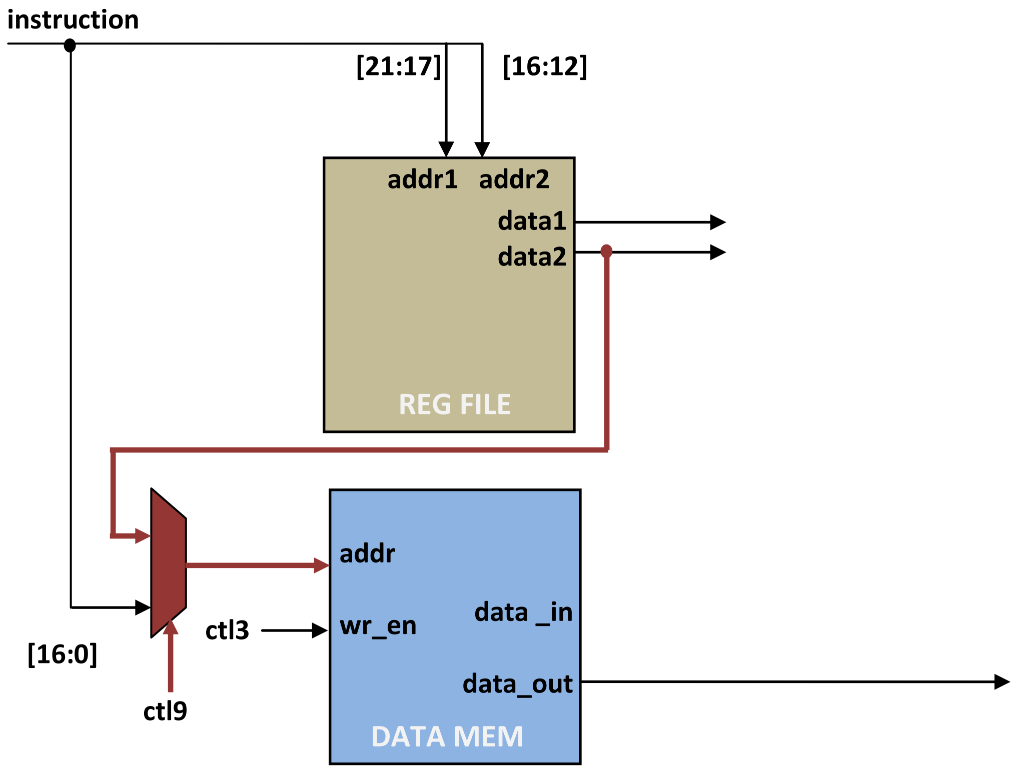

Figure 12: LW with registe as address support

Since the **Dest** field is already tied to write port of the register file so we just need to ensure that ctl1 is asserted on this type of instruction. We intentionally left four bits unused as by using **data2** instead of **data1** as source address we can use common logic to support LW and SW instructions (As further elaborated in next paragraph).

For incorporating **SW with register value as memory address** we need two registers as well. The value of register **Source1** – just like in normal SW instruction – is used as data to be written on Data memory. The **Source2** register provides Data memory address just like the case of **LW with register value as address**. So same hardware can be used, we just need to make sure that **ctl9** selects address from register in this type of instruction.

Opcode [4:0]	**Unused [4:0]**	**Source1 [4:0]**	**Source2 [4:0]**	**Unused [11:0]**

Figure 13: SW with register address instruction format

The integrated diagram of EvoproK is shown on next page.

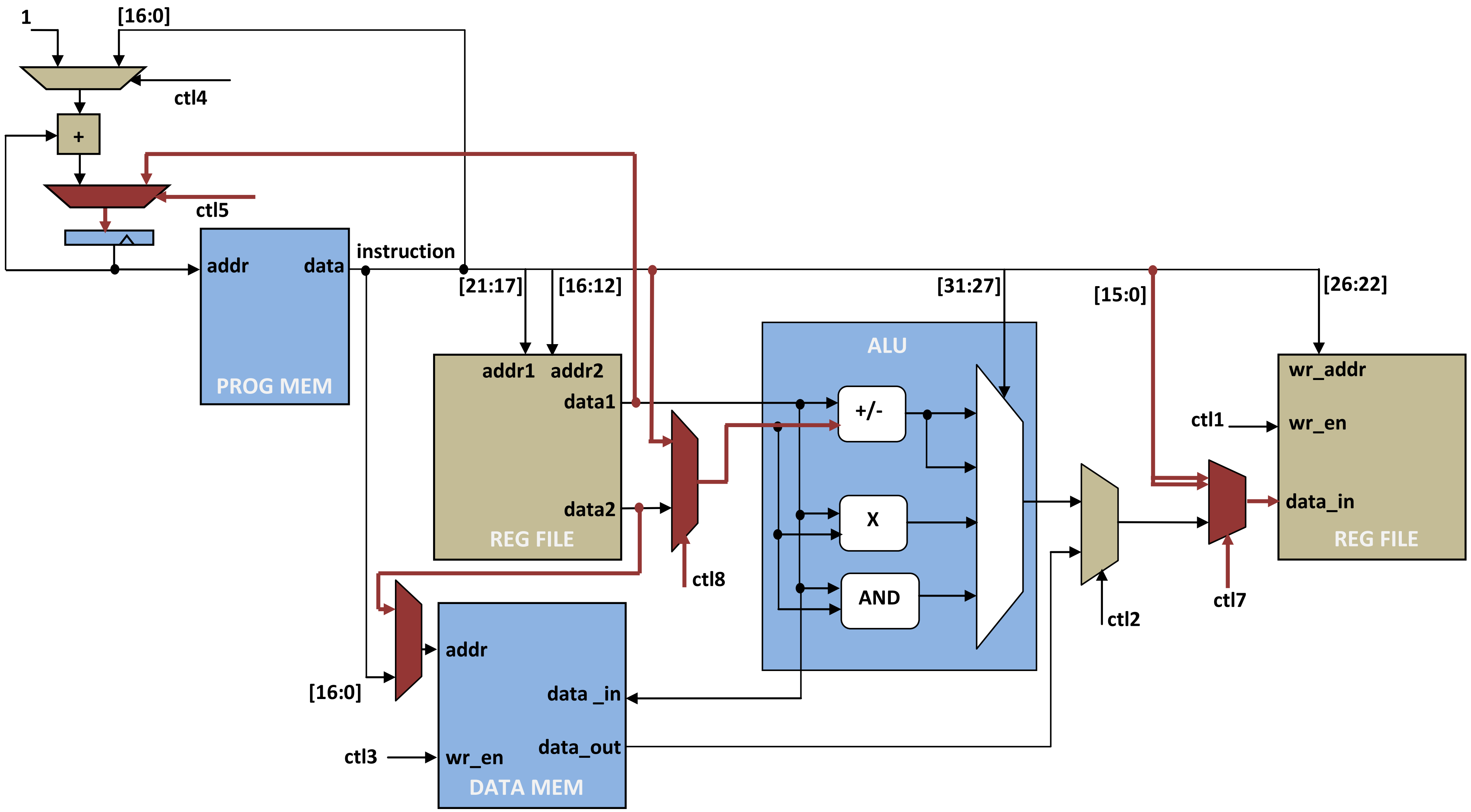

Figure 14: Integrated Diagram