**CPU Components**

**Arithmetic and Logic Unit (ALU)**

The ALU was designed to perform each of the arithmetic operations defined by the instruction set. These operations are bitwise AND, bitwise OR, bitwise NOT, bitwise XOR, addition(ADD), addition with carry(ADC), subtraction(SUB), subtraction with borrow(SBB), negation(NEG) and compare(CMP). A number of these operations are similar enough that they can share a large amount of circuitry, reducing hardware overhead. In particular, the add instruction is the same as addition with carry, only omitting the input carry and not setting the carry bit. The subtraction with borrow hardware can also be used to perform a SUB operation and the CMP operation.

The ALU module takes 4 inputs: 2 register buses of 8 bits each, an ALU select line which determines the operation to be performed by the ALU and a carry in line of 1 bit. It has an out bus of 8 bits carrying the result and 3 bits which maintain the carry out, N and Z flags.

If the current opcode is encoding for an ALU operation the output of the ALU is always the result of the 2 currently active registers, i.e it is constant rather than event triggered. If the opcode does not match an ALU instruction, the ALU ouptus “00000000”. This value is never written to a register however as the ALU is only ever set to write to a register if the current instruction dictates it.

**Registers**

The CPU maintains 8 Genreral Purpose Registers each 8 bits long, 3 address registers of 16 bits in length, a program counter and a status register, each of 16 bits length.

The standard registers are controlled by a demultiplexer on the input side and 2 multiplexers on the output side (from the perspective of the registers). The demultiplexer controls which register is written to based on a select line which is determined by the opcode for any given cycle. There are two multiplexers, which are wired to the ALU inputs. The output from these multiplexers is also used to pass data to address registers or other registers depending on the current instruction. Each of the two outputs is controlled by a select line, with one output always showing the register which is being written to.

**Control Unit**

**Decoder**

The leading 6 bits (MSBs) of any specific opcode encode the instruction to be performed. 6 bits of information gives 64 possible states and thus 64 possible operations; however the instruction set implemented only contains 24 entries, thus many codes go unused. The decoder determines what each instruction means based on the first 6 bits and sets the register select and ALU lines accordingly. This happens as soon as the opcode changes.

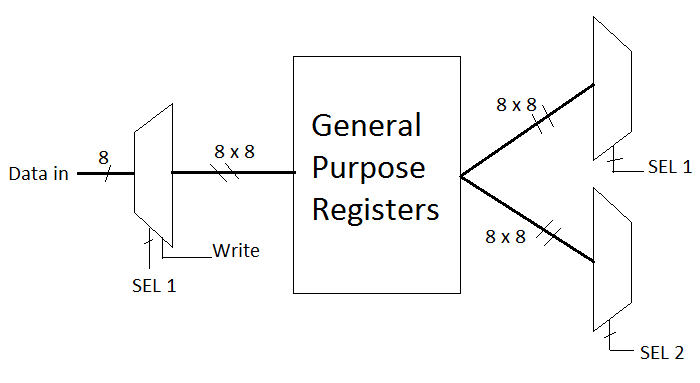


Figure 3.1: Structure of the General Purpose Registers

**Register Control**

The control unit contains decoder that control the select lines for the register multiplexers. Each opcode is interpreted and the register select lines are then assigned. For example an AND statement has the opcode 0 0 0 0 1 0 0 0 y2 y1 y0 0 0 x2 x1 x0 where y0-y2 and x0-x2 determine the specific registers which will be ANDed together. Also y0-y2 is the register which the result will be written back into. Therefore the same code can be used to identify one of the output registers and the input register. This code is given the designation SEL1 while the code for the remaining register is called SEL2. The opcodes are written such that for most instructions the registers are defined in the same place in the instruction word; this made the process of defining the register select codes very easy. Exceptions were coded for in this module (Cu.vhd, line 150). As there are 8 general register each SEL code is 3 bits long to provide the 8 states necessary.

A similar process was adopted for the address register controls. As the address registers are 16 bit in length compared to the 8 bit registers an additional bit is required to indicate which half of the address register is to be used in the instruction.

**Write Control**

A write event occurs every clock cycle. At each event each register type (REG, PC, ADR, SR) is written to regardless of whether that register has changed value this clock cycle or not. Each register has a feedback loop that will write the current value of the register to the register if it hasn’t changed value (determined by the control unit and passed to the write control). For example the value that is written to a standard register can be either an immediate value, the value of another register (standard or address), its current value added with another number, or its current value. The hardware that controls this is a multiplexer whose select signal is changed depending on which instruction is being performed.

**Special Registers**

There are two special registers, the Program Counter (PC) and the Status Register (SR). The program counter is a 16 bit register which stores the address of the current instruction while the status register, which is also 16 bits, contains the N, Z and Carry flags.

The Program Counter increments by one every cycle, unless a branch instruction is performed. A separate control module for the Program Counter performs load, increment and branch operations on the program counter. Its output is fed directly to the MMU, as it contains the memory address of the next instruction to be executed.

**Description of Operation**

In its current form the CPU performs operations in the following manner:

1. Gets instruction from testbench. In a final implementation the CPU would get its instruction from the address in memory specified by the program counter. This is timed in the testbench to occur on a falling clock edge. \*
2. Decode the instruction
3. Set all control lines/buses to correct values for given instruction. Includes setting write line to true
4. Calculate result and wait for write cycle
5. Write result or data from another register to register specified by instruction. This is initiated in the testbench to occur on the rising clock edge.
6. Set write back to false.
7. Repeat for next instruction

\*In conjunction with an actual MMU, the incoming opcode would change when the MMU responds to a change in the output value of the Program Counter register. This would be well before the clock falling edge (assuming a slow enough clock).

Below is a basic timing diagram for a few instructions

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Clock Edge | Instruction (paraphrased) | Register0 | Register1 | Register2 | Address\_Reg1 |
| Down | load Reg0 | UUUUUUUU | UUUUUUUU | UUUUUUUU | UUUUUUUUUUUUUUUU |
| Up |  | 11110101 | ~ | ~ | ~ |
| Down | load Reg1 | ~ | ~ | ~ | ~ |
| Up |  | ~ | 10101010 | ~ | ~ |
| Down | Reg0 XOR Reg1 | ~ | ~ | ~ | ~ |
| Up |  | 0101111 | ~ | ~ | ~ |
| Down | load Reg1 | ~ | ~ | ~ | ~ |
| Up |  | ~ | 10111000 | ~ | ~ |
| Down | R0 AND R1 | ~ | ~ | ~ | ~ |
| Up |  | 00011000 | ~ | ~ | ~ |
| Down | ADR\_Reg1(HIGH) <- Reg1 | ~ | ~ | ~ | ~ |
| Up |  | ~ | ~ | ~ | 10111000UUUUUUUU |
| Down | ADR\_Reg1(LOW) <- Reg0 | ~ | ~ | ~ | ~ |
| Up |  | ~ | ~ | ~ | 1011100000011000 |
| Down | Reg2 <- ADR\_Reg1(LOW) | ~ | ~ | ~ | ~ |
| Up |  | ~ | ~ | 00011000 | ~ |

Table 3.1: Timing and Register outputs.

This above timing diagram shows the state of each register at each clock edge as this is what currently drives the CPU write cycle, where U indicates the bit is undefined and “~” indicates that the register has the same value as previously. The diagram clearly shows that on each rising edge the register updates its value correctly. This data is from a testbench run on the final design, the original timing diagram is in the appendices [REFERENCE] but presented here in table form for clarity.

**Design Process**

The design concept for the CPU was to break the project down into smaller tasks that would be easier to complete. This is a valid design strategy for a lot of projects but given our lack of experience coding in VHDL and the reasonable scope of the project it seemed especially appropriate.

As such the Initial focus was on getting a working ALU. The bitwise AND, OR, NOT and XOR commands were easily implemented but some trouble was encountered in realising more complex operations such as ADD and SBB. This was because these operations involve a carry which we initially tried to implement concurrently. This proved to be the biggest hurdle in the design on the CPU as the concept of concurrency is not one which had been encountered in other programming languages and took a while to start thinking about what the code was doing. The output of each ALU function was verified to be correct by using a testbench with predefined inputs to the ALU and reading out from the output. All functions were correctly implemented. This can be verified using the ALU testbench file (etalu.vhd).

The next stage of the design was the registers. This was broken down further by defining the standard registers, address registers and special purpose registers separately. The registers proved easy to implement as their size was the only thing needing to be declared.

It was decided that the easiest way to implement writing and reading functionality in the registers would be to design “sub” control units that would handle the reading and writing for each type of register. This approach was chosen as it would allow functionality testing of a much smaller entity of the full control unity and was in line with the design concept. Once designed the sub units could be easily folded into the higher control unit architecture. The register control units were tested using the LOAD IMMEDIATE and STORE IMMEDIATE instructions as test cases. The address control unit and program counter unit were developed using the code from the register control as the requirements of each module are essentially the same.

Once the registers were receiving and outputting values when requested the final component to implement was the control unit. The control unit was implemented a level above the register sub controls, passing values to the sub control units after interpreting the opcode. The control unit is also responsible for maintaining the timing of the CPU and initiates each write cycle.

**Integration into the Final Design**

The CPU interacts solely with the Memory Management Unit (MMU). This interaction consists primarily of receiving an instruction as a response to a fetch request, performing the operation and if required sending a value back to the MMU to be stored at a particular memory location. The location in memory of the next instruction is determined by the program counter. The program counter increments by one after each successful instruction thus the instruction will be contiguous to the last. The exception to this is when an instruction specifically alters the instruction address such as a JUMP TO (JMP) statement. While we believe that we have implemented the required hardware to interface with the MMU we were unable to test it due to some persistent errors in the MMU implementation and a lack of time.

**Discussion**

**Things we did well**

The modular approach adopted for this project worked extremely well. By developing modules that handle discrete tasks from the beginning we were essentially creating “black boxes” that took some inputs and gave some outputs. Because of this it made creating the control unit which is the most complex part of the CPU relatively straight forward as the task was reduced to interpreting each instruction and then setting the inputs to the system according to that particular instruction.

**What we learned and what we could do better next time**

As this was our first hands on experience coding in VHDL we learned a lot about basic VHDL coding. The concept of concurrency took a bit of time to fully appreciate and is very different compared to software languages such as C and requires thinking about a problem in a slightly different way.

We also could have named signals and variables in a more coordinated way. When in the rhythm of programming it is not too hard to keep track of everything but can take a few minutes to re-familiarise oneself with the inner workings after being away from it for a day or two.

We didn’t make any effort to code in a similar style to the other members of the group which would lead to potential difficulties in understanding the code if it eventually became integrated with the other components of a microprocessor.

It would have been beneficial to start work on the project sooner. While we ended up with a working design that satisfied most of the specifications we didn’t necessarily meet them in the optimal way and allowing more time would have given opportunity to investigate other methods of doing things. There were heavy constraints on available time however given other projects due before this one.



R0

R7

A0

A1

A2

Control Unit

Program Counter

Figure 3.2: Interconnections between modules

References:

[1] Altera Corporation, “VHDL Adder/Subtractor,” in “Design Examples”, 2011, Available: <http://www.altera.com/support/examples/vhdl/vhd-add-sub.html> (accessed 12.10.11)

[2] Green Mountain Computing Systems, “Behavioural Descriptions” in “An Introductory VHDL Tutorial“, 1995, Available: <http://www.gmvhdl.com/signals.htm> (accessed 12.10.11)

[3] Francis Bruno, “VHDL Concurrent Statements” in “VHDL Reference Material”, UMBC University, Maryland, Available: <http://www.cs.umbc.edu/portal/help/VHDL/concurrent.html> (accessed 12.10.11)

[4] Jiri Gaisler, “A structured VHDL design method” in “Fault-tolerant Microprocessors for Space Applications”, 2007, Available: <http://www.gaisler.com/doc/vhdl2proc.pdf> (accessed 12.10.11)

Notes for testbench results:

Ins\_data: the opcode

Dat\_data: the data data bus

Ins\_addr: The program counter register value

Dat\_addr: The value of the currently selected address register

Test: The register input, Data\_in

Regout1: The register output defined by SEL1

Regout2: The register output defined by SEL2

Alout: The ALU output.