EE345 Quartus 2 example simple RISC project

John Gilmer Oct 31 2015

Background:

This example project is intended to be useful in starting a project for EE345 by illustrating the use of various components needed to complete a datapath. The datapath it intended to implement a 10 bit wordlength machine having 4 registers and a "minimal" instruction set. Within the constraints, the machine is roughly "MIPS-like" but in addition to the very limited number of registers, it uses two address instructions instead of three address instructions. At this time no provision has been made for input, output, display of current registers and such. (That will be added later.) Several instructions, including those which support jumps or branches, have not yet been added. As shown, the design compiles, but has not really been tested. The datapath has no interlocks.

The Instruction Set and Example Program:

The instruction set is minimal, consistinmg of the instructions and formats shown in Figure 1 below. The formats include a "J" type (for j and jal), an I type (for bez, lui, and addi) and an R type (for the remainder of the instructions, the "operate" types).

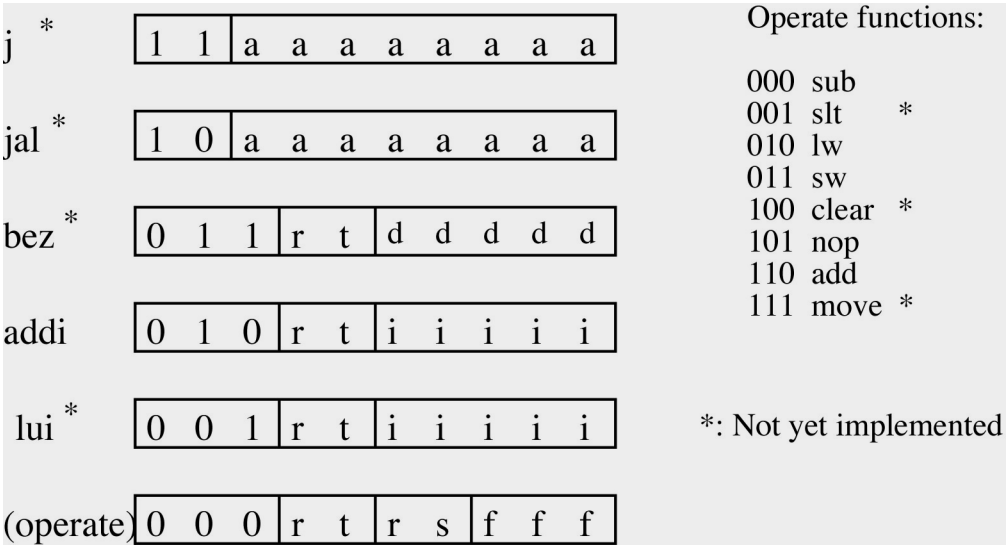

Figure 1 Example project instruction set

There is only one data type, a "word" of 10 bits. Because the jump and jump and link instructions have only 8 bits of operand, the machine is limited to 256 words of instructions. The data memory is 1024 words since registers are used for addressing. Unlike the original MIPS, register 0 does not always have the value zero. That is why a clear instruction was included. However, the clear could be accomnplished by subtracting a register from itself, so that opcode could be used othgerwise. There is not now a "jr" (needed for a return from a subroutine). It was thought that "addi" of the value 0 could be used for that, or some other non-useful combination. (The addi of 0 could instead be used as a nop.) So, as it stands now, there are still some loose ends to be addressed in the instruction set.

This is a two addrerss machine. The register used for both source and destination is designated “rt”. If a second register is used (as with the operate instructions), it is called “rs”. Most instructions both take data from rt and modify rt. A move instruction is needed since the add to zero can’t be used for that purpose. The store and load word instructions use only a register indirect mode, allowing this to be a four stage pipelined machine.

Because of the limited number of bits, the “bez” compares a register to 0 and branches if equality. This can be used for an unconditional branch only if the register is cleared first. A jump can be used instead.

An example program to add the elements of a constant length vertor is shown in Figures 2 and 3. This program can be used as a test of the machine. (It has not yet been developed to use a function call. Some instructions needed for this remain unimplemented at this time.)

```
#define N  3
int sum;
int A[3]={1,2,3};
main () {
        int i;
        sum=0;
        for(i=0;i!=N;i++) sum=sum+A[i];
}
```

Figure 2 C program for adding elements of fixed length vectors

label	instruction	machine(hex)	comment
start:	clr $1;	046	$1 will be used for variable “sum”
	clr $0;	04e	$0 will be used for variable “i"
	clr $2;	056	$2 will be used for A[]
	addi $2,A;	141	We assume A is at address 1 in data memory.
loop:	addi $0,N;	11d	We add –N (1d) to I to see if we get zero
	bez $0, exit;	188	If zero, we exit at +8 from current address
	addi $0, N;	103	add N back to i so it’s at correct value
	lw $3, ($2);	072	load the word A points to
	add $1, $3;	03e	add the value of A[i] to sum
	addi $2,1	141	increment A to point to the next A[i]
	addi $0,1	101	increment i
	clr $3	064	clear $3 for the bez to make it unconditional
	bez $3, loop	1f7	branch back for next iteration
exit:	clr $3	064	clear $3 for the bez to make it unconditional
	addi $3, sum	160	sum is at address 0 in data ram
	sw $1, ($3)	03b	store the accumulated sum into the variable “sum”
	clr $3	064	clear $3 for the bez to make it unconditional
halt:	bez $3, halt	1ff	branch to here forever to halt execution

Figure 3 Assembly and Machine program for adding elements of fixed length vectors

Implementation in Quartus 2 using a schematic:

Figure 4 below shows the overall schematic. The arrangement is generally as we have illustrated such pipelined datapaths before, with the PC on the left and the writeback stage on the far right, though in this case the writeback signals have to be led back to the left to where the register file is. Notice that the macrocells all have lists of parameters that are important to set correctly. Figure 5 zooms in on the left half of the machine including the fetch and decode machinery, and Figure 6 focuses on the execute (and memory) and writeback functions.

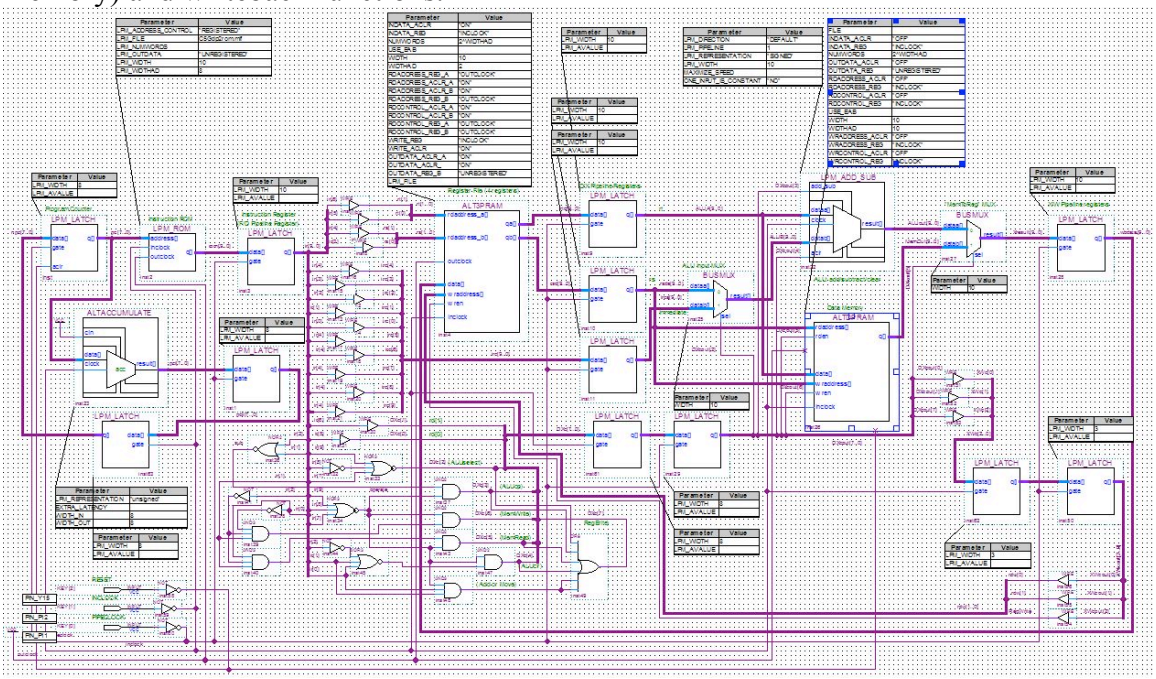

Figure 4 Datapath implementation (overview)

An initial description needs to be made about clocking. The Quartus 2 latch macrocell "LPM_LATCH" implements a level sensitive latch. In contrast, as we have been studying the operation of RISC datapaths, we have assumed edge triggered latches. In actual practice, level sensitive latches are preferred, and that is what Altera gives us. The "pipeline registers" including PC and IR are thus level sensitive latches. However, most of the functional blocks provided in the Altera macrocell library are "synchronous". That is, they include registers on either their inputs, outputs, or both. We do not have the option to use these macrocells in an entirely unclocked mode. In all of these cases, therefore, the input clock is used to provide a second stage of latching in each pipeline stage. That is necessary since, with transparent latches, data might flow through more than one pipeline stage at a time. Where units have two clocks (for input and output), the output clock has either been deselected when the unit is defines or has been tied high to leave it in a transparent mode.

The two clocks, PIPECLOCK and INCLOCK have been connected to the pushbuttons "KEY0" and "KEY1" respectively. Inverters are used to make them active high. We never want more than one of these clocks enabled. As the clocks alternate, the instructions will march down the pipeline. (Pushbutton "KET2" clears the PC to start.)

In some cases, where data flows from one pipeline register to another with no intervening clocked macrocell, an extra latch has to be inserted to be a barrier that is synchronized with the macrocells (and this connected to “INCLOCK”). An example is seen in the feedback path for the PC, and in the control signal latches.

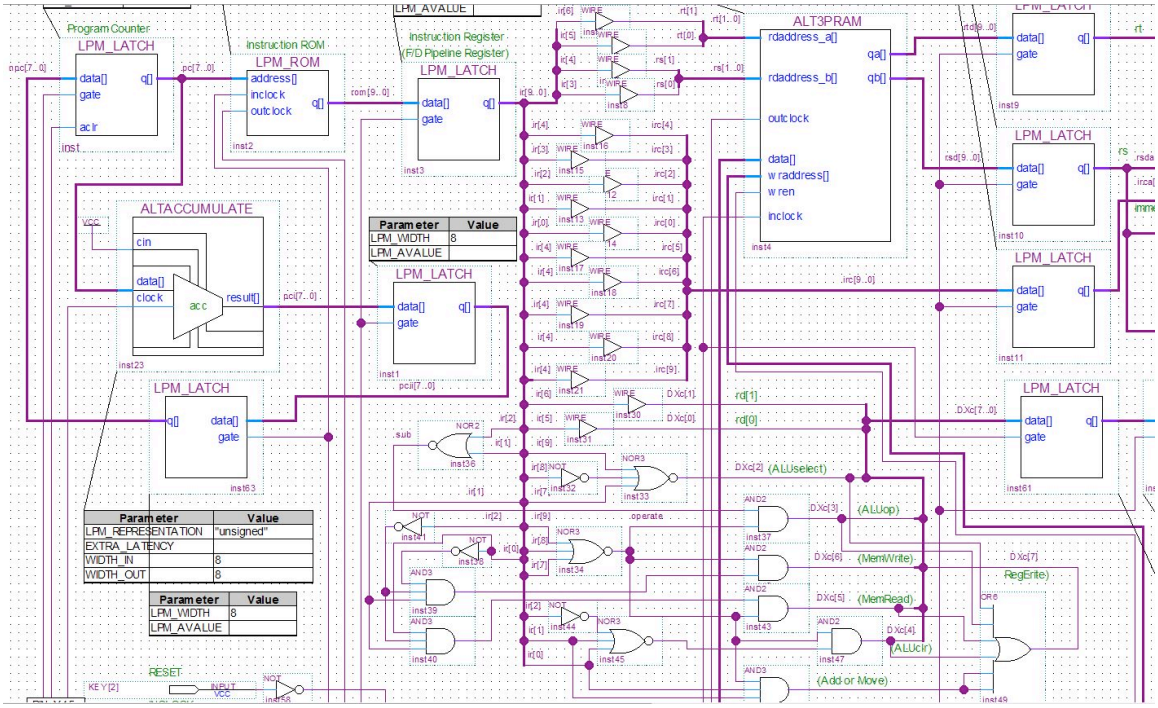

Figure 5 Datapath implementation (Fetch and Decode stages)

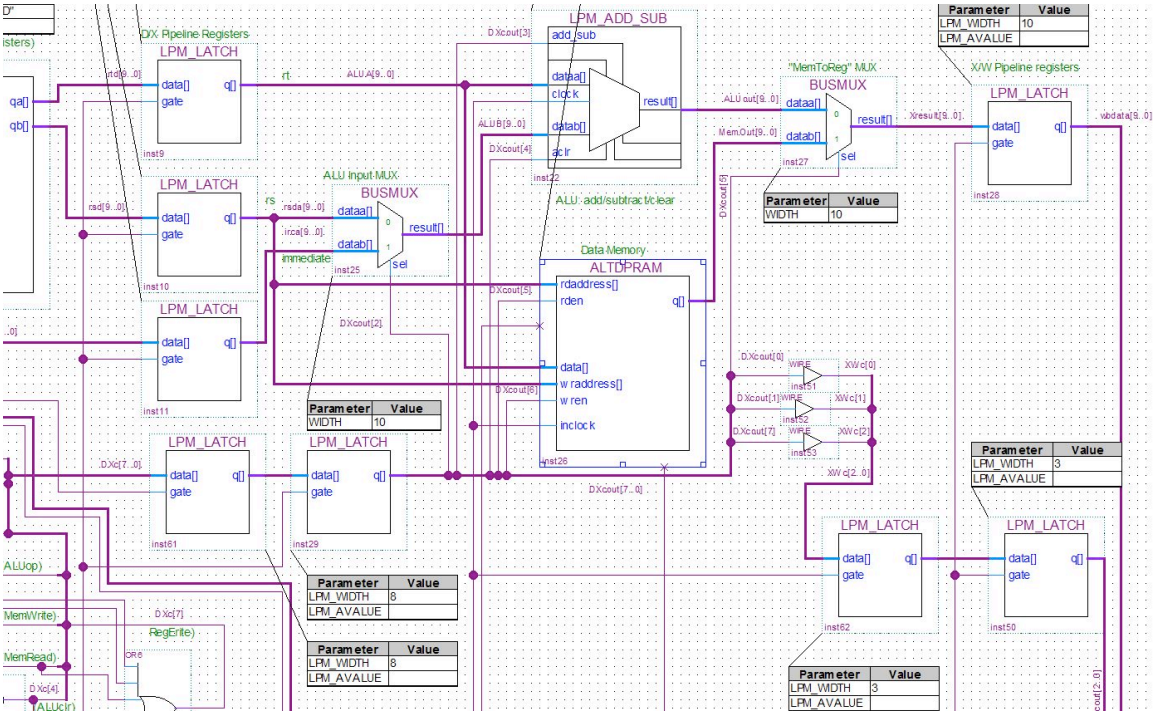

Figure 6 Datapath implementation (Execute and Writeback stages)

Starting with the fetch stage, notice that we have a ROM to hold the program. The ROM must be initialized by an "initialization" file. In this case it is named "C5Gdp2rom.mif". If that file is not found during compile, the compile fails. The initialization contains the program machine code seen in Figure 3 earlier. The unit "ALTACCUMULATE" is a convenient macrocell for adding one to a number. It's less flexible than a full ALU, but more convenient when you just want to add 1. Note that we get +1 by making cin = Vcc (1). Note also that this unit is synchronous – it contains a register. It actually has the option to contain two registers. (We could have used the output register as a substitute for the next LPM_LATCH clocked by PIPECLOCK.) That is true in a number of other places as well, such as the register file.

Notice the number of "wire" buffers in the decode stage. This is necessary because a given wire may need to have two names. For example, all wires coming out of the IR are named ir[i] for some bit number i. The rt field of the instruction is bits 5 and 6 of the IR, but we need to rename those wires rt[1] and rt[0] so that we can bus them together for the rt read address of the register file. Similarly, bits 4 to 0 of the IR are the constant needed for the add immediuate instruction, and we need to replicate IR bit 4 into the other 5 bits of the immediate value to extend the sign. We thuus have to use the "wire" buffer to break the wire up for naming purposes, while still teling Quartus 2 that this is still physically one wire.

The control unit is implemented here with discrete logic. It would have been much better to make the control unit a subcircuit with a bus (for IR) going in and the various control signals coming out. A NOR gate is used to detect whether the instruction is an "operate" type. That then is ANDed with logic that detects various function codes for different instructions. Ultimately the various control signals are bussed together into 8 bits for the D/X pipeline register. (This logic is very messy.) Note that none of the control signals are used in the Decode stage. (Later the jump and branch will be done in this stage, but those are not implemented yet.)

I believe that it will be possible to forward through the register file, but that has not yet been tested.

In the Execute phase, either the ALU or the RAM will be used. It took some fiddling with the RAM parameters to get it to compile. At one point the exit clock was in use, and the wire to it still terminates (unconnected) at the RAM macrocell block. This particular ALU does either add or subtract, but it also has a clear which, it is hoped, will set the output to 0. The most significant bit of the output can be used to indicate "negative" for purposes of implementing the "slt" instruction. That will require another (or a wider) MUX in the future, and another control signal. (Right now "slt" operates as a subtract.) Notice again the extra LPM_LATCHes in the flow of the control signals to give them delays for INCLOCK at the same point as for the data macrocells. The 8 control signals are reduced to just 3 bits (RegWrite and rd) for the Writeback pipeline stage.

Something else discovered is that busses do not have to be named if they go from just an output port on one macrocell to a similarly sized port on another macrocell. Naming them is necessary if there is any sort of "breakout" of signals though.

Conclusion:

It is hoped this example will be useful in starting the projects.