

CS/EE 3710 — Digital System Design

Checkoff #1 — Register File and ALU

Overview

The instruction set of the CR16 uses standard 16-bit integer ALU operations. So, if we assume that the decoding has been done (actually you will design the decoder in a following lab), what you need to build to execute each instruction is an ALU to compute the numbers, and a register file to provide arguments and hold results. So, this checkpoint has you design and demonstrate those pieces.

As an aside, it's interesting to note how much of the machine can be designed before knowing the complete picture of what the machine will look like. There are a set of basic things that are common to a number of different designs. Building these basic components (memory, register file, ALU, etc) doesn't really require knowing much more than the word size and the overall plan. Filling in the details later will determine the precise character of the final machine. Most VLSI design systems, for example, have large libraries of these generic parts which can be used to design a wide variety of different machine types.

Note: This checkpoint, and all subsequent checkpoints, refer to the *baseline instruction set* for the class processor. You can see the details of the CS/EE 3710 baseline ISA on the class web site at www.eng.utah.edu/~cs3710/handouts/ISA.pdf. The instructions marked with * are the required baseline subset. The instructions marked with a *? are very strongly encouraged instructions. Basically your group will have to argue convincingly that your application will never use that instruction to get out of implementing it. The instructions with no marking are optional, and probably most groups won't deal with them. The optional instructions mostly deal with exceptions and system processor state, and I expect that most people can get by with polling in their programs and won't need the extra complexity of interrupts.

Execution Datapath

Looking at the Baseline instruction set you should notice a few things:

- The word size is 16 bits. Therefore, although it's not absolutely essential, it would be natural to have a 16 bit ALU. It's not required because you could clearly do the ALU operations with a smaller ALU in a sequential way. The 16-bit result could, for example, be generated using an 8-bit ALU and two timesteps. Going to the extreme, you could use a single-bit ALU and iterate 16-times for each ALU operation. As an aside this is actually the approach that was used on the Connection Machine. This was a machine designed to be massively parallel. The full configuration had 64k (i.e. 65,536) total processors. In order to physically assemble this machine, each processor had to be very small so that many of them could fit on a single chip. Each processor had a single-bit ALU and operations were either done in sequence on a single processor, or done by combining processors into groups.

Anyway, getting back to the point, your ALU should probably be 16-bits wide to match the data word size.

- The instruction set supports both ADD and SUB instructions, so your ALU should also support these operations. Reading the CR16 handout carefully you will note that signed numbers are represented as two's complement numbers. So, your ALU should operate on two's complement, and can use this representation for implementing subtraction. The main difference between a two's complement ALU and unsigned ALU is in the way condition codes are generated. Your ALU needs to generate condition codes for both assumptions. That is, the ALU doesn't know if the bit patterns it sees on the input are unsigned or two's complement, so it has to generate both signed and unsigned

condition codes on every operation and let the programmer sort things out by choosing to look at one code or the other.

The condition codes required by the Baseline instruction set are defined here and also in the ISA document. Note that the ADDU and ADDUI instructions are just like ADD and ADDI but they don't set any condition codes.

- C** — The Carry bit determines whether a carry or borrow occurred after *unsigned* addition or subtraction. 0 means no carry or borrow, 1 means a carry or borrow occurred.
 - L** — The Low flag is set by comparison operations. L is set to 1 if the Rdest operand is less than the Rsrc operand when they are both interpreted as *unsigned* numbers.
 - F** — The Flag bit is used by arithmetic operations to signal arithmetic overflow (this is sometimes called the V bit on other machines). This bit should be set if there's a signed overflow on a *signed* addition or subtraction.
 - Z** — The Z bit is set by the comparison operation. It is set to 1 if the two operands are equal, and is cleared otherwise.
 - N** — The Negative bit is set by the comparison operation. It is set to 1 if the Rdest operand is less than the Rsrc operand when both are considered to be *signed* integers.
- The processor supports logical operations of AND, OR and XOR. Like the arithmetic operations, these can either take two registers as arguments, or one register and an immediate value.

Note that the baseline instruction set does not have a NOT operation to invert all the bits of a register. It's worth considering if that might be useful for your application. One typical way to get around this issue is to XORI with a constant of all 1's, but in our case the 8-bit immediate values are zero-filled in the higher order bits, so that doesn't work. Alternatively you could load up a register with all 1's and then XOR against that register for NOT.
 - The Baseline instruction set has one shift instruction: logical shift. An obvious extension of the baseline set would include an arithmetic shift. These instructions have an immediate field as part of the instruction. In the full processor design this is used to indicate how many places to shift. A value of 1 in the immediate field shifts the argument register by 1 place to the left. A value of -3 indicates a shift of 3 places to the right. In our Baseline processor, you can assume that all shifts will be by one only. The only legal values for the immediate are 1 and -1 to indicate left and right shifts. You are, of course, free to augment this in your processor.
 - Branch targets are computed as a displacement from the current PC. To compute the branch target a sign-extended offset in the immediate field of the instruction is added to the current PC and written back to the PC if the branch condition is true. Because the displacement is signed you can branch forwards or backwards from the current PC. But, because the imm field is only 8 bits in our instruction encoding that means that you can only branch to -128 or 127 instructions past the current instruction using the baseline instruction set.
 - Jump targets are taken directly from a register. In the case of a jump the register that holds the jump target is written directly to the PC if the jump condition evaluates to true.
 - Jump and Link (JAL) is just like a jump, but the the PC+1 value is also written to a register. This is known as the link register and lets you jump to a subroutine and then return back to the point in the code where the subroutine was called. You can do this with a JUC Rlink instruction (jump unconditional to the value that you stored in the Rlink register). This is traditionally an instruction

that gives people trouble when they implement it. It's not hard, but you do need to get the details right and make sure that eventually you provide a path for the PC to get into the register file. That's actually for a later checkpoint though.

- Load and store instructions need to use values in registers as memory addresses. So, you might want a separate path to the Memory Address Register (MAR), or you might put these values through the ALU in some sort of "pass through" mode. Somehow these register values must be usable as memory addresses. Perhaps you can postpone this decision to a future checkpoint too when you implement the MAR and MDR.

There's not a lot of room in this instruction encoding for a LOAD or STORE with an indexed address. That is, some machines allow you to take the base address in a register and add an immediate from the LOAD or STORE to get the final memory address. You could do this with the last unused opcode on Page 2 of the ISA handout and use the xxx bits as a three bit offset, but I'm not sure that's worth the trouble.

Looking at these constraints it is possible to do a general layout for the execution datapath of the machine. The important thing at this point is not to get everything exactly right the first time, but to make sure that you haven't made anything impossible that you will need to do later. The set of control points in this datapath (function codes, mux select signals, latch enables, etc) will be controlled by the control state machine and the decoded instruction.

Register File

The register file must have two read ports. You must be able to read two arguments from the register file to feed the ALU. It also must have a single write port so that you can write back the result. However, since one of the source registers is overwritten with the result, it need only have two addresses specified. These addresses come from the decoded instruction word. You probably want to use Verilog similar to the register file description in the mips.v code for this, but remember that you don't need an independent write address. One of the read addresses is also the write address.

The ALU should be able to read these two register's contents and perform an operation on them. However, the ALU must also be able to operate on a register and an immediate value (an ADDI for example), and the PC might also need to be combined with an immediate value (for a displacement-branch for example). These immediates will need to be sign-extended before going into the ALU. You should also consider how to describe sign extension in Verilog (hint - think about concatenation). So muxes on the inputs of the ALU seem required. Also, it's possible that a register value might be sent to the shifter instead of the ALU. So, the output of the execution unit might come from either the ALU or the shifter. Sounds like we need another mux on the output side. One example of a possible organization is shown in Figure 1. *Don't take this block as the required organization!* This is just an example of how things might be organized. It's not even guaranteed to be correct for all the instructions that the Baseline instruction set needs. If you have a different organization in mind, by all means use that. Just make sure that it is able to compute all the values that will be required by the given instruction set! By being clever you may be able to reduce the number of muxes, or combine units together, or modify things in any number of interesting ways.

One approach to designing this ALU datapath is to look at every single instruction that you're interested in (i.e. the Baseline instruction set, plus any additional instructions you're thinking about), and check that each one can be accommodated by your datapath.

Some things not shown in Figure 1 are the control points in the datapath. The ALU, for example, will have a function code coming from the control path that tells it which operation it is actually doing at the moment. The muxes all have select inputs telling them which way they are switched. The latches have enable signals, etc. All these signals are the control points into the execute unit that allows the state machine controller to make it do the right things for each individual instruction. This is all very similar to

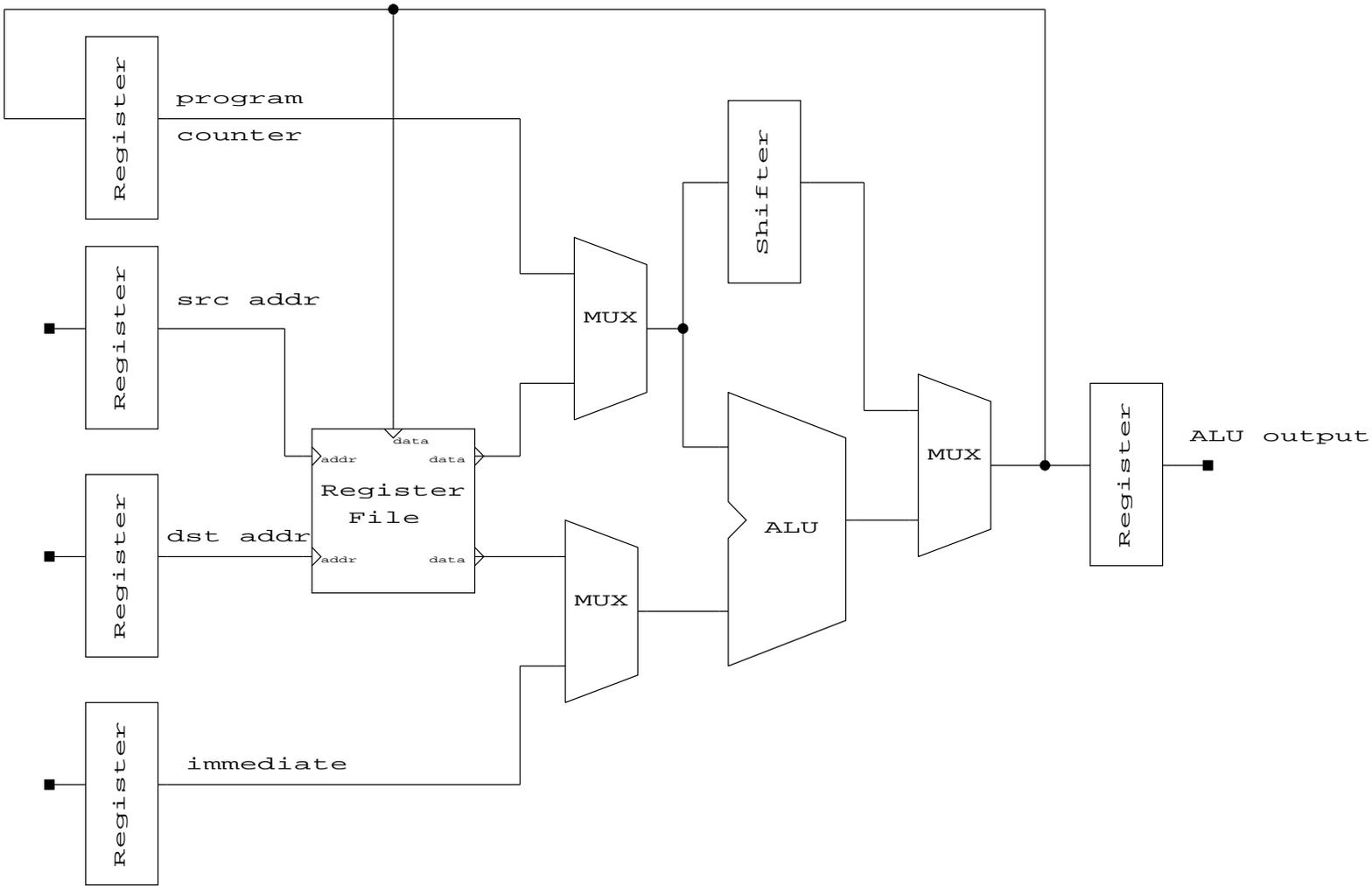


Figure 1: One Possible ALU Organization

the mips.v that we looked at in lab2, but of course it's all different because this is a different instruction set!

Remember that your ALU must generate condition codes that will be latched into the processor status register (PSR). Your ALU should generate all the Baseline condition codes. Also pay attention to which instructions update the condition code register! Not all ALU instructions update the PSR register, and even those that do, don't all update it in the same way! Some instructions only set some of the bits, and some instructions don't set any of the bits. Not even all the ADD instructions set the PSR. So, you need to have control over whether the PSR is updated or not. The decoder you design later can decide which PSR bits are updated, but you at least need to plan for the possibility of different sets of bits being set at different times.

What to Do

For this checkpoint you should design an ALU and a register file which will be used eventually in the execute unit of your processor. The overall block will look similar to Figure 1, but definitely will not be exactly like it. Your datapath should support all the instructions from the Baseline instruction set, although you do not have to design the control for those instructions for this lab. You should also be thinking at this point about what types of things you'd like to add to the instruction set. If you already have some ideas, adding support in the data path now might save redesign later. The register file should contain 16 16-bit registers, and your ALU should be capable of performing 16-bit arithmetic. The program counter and immediate register for this lab will also be 16-bit registers. These will likely be implemented slightly differently in the final circuit, so make sure that you can change things later.

Note that you can design this in pure Verilog, pure schematics, or a mix of the two. My choice would be to design the major blocks as Verilog, but use schematics to connect them up. That way you can debug things on your schematic (i.e. something that looks a little like Fig 1.53 for mips) instead of just staring at code. But, pure code certainly works fine too. Even if you use Verilog remember that you should think about the hardware first, and then write Verilog that describes that hardware. Think structurally in terms of system decomposition.

Testing You should test your register file and ALU using the simulator and be prepared to explain your testing procedure, demonstrate the functionality of the circuits, and show the command files and simulation logs from your own testing. Of course, if you can build in checks in the testbench it's much easier to see if things are still working when you make changes! If you have waveforms, make sure you annotate them so we can tell what's going on.

To meet the checkpoint you should have schematics, Verilog, etc. that defines your ALU and register file, along with testbenches and output from the simulator to show that it's working. You should also prepare a table that shows, for every instruction that you're planning on implementing, how each of the control points should be set. By control points I mean all the mux settings, enable settings, etc that will have to be generated by your controller in order for those instructions to work properly. It's all right, of course, if some of those control points are "don't care." This will help both you and us determine if you've thought of all the possibilities that your datapath will have to support.

You can check off this checkpoint in an upcoming meeting between your group and the instructor and TA.