State University of New York at New Paltz

Department of Electrical and Computer Engineering

EGC453
Intro to Computer Architecture

Intro to Computer Architecture Final Project: Design of a MIPS-like processor

Group Members	Department	Major Contribution
Kevin Anderson	CE	ALU Design; Datapaths
Mike Facelle	CE	Machine codes; Report

Course Professor: Dr. Baback Izadi

12/10/13

Table of Contents

Introduction

A MIPS (microprocessor without interlocked pipeline stages) processor is very efficient because it uses Reduced Instruction Set Computing (RISC). This type of architecture uses basic instructions that are very simple and usually all the same length. The simpler instructions allow for faster cycles and optimized code. This makes designing hardware to accommodate these instructions simpler but all code must be reduced to the simple instructions that are part of the instruction set architecture (ISA). In this paper we are designing the datapath for a MIPS-like processor with a limited set of instructions.

This processor was to be designed for the instructions found in Table 1.

Instruction	**Assembly Code**	**Operation**
Add	add $rd, $rt, $rs addi $rd, $rt, #n	$rd ← $rt + $rs $rd $rd ← $rt + #n
Subtract	sub $rd, $rt, $rs subi $rd, $rt, #n	$rd ← $rt – $rs $rd $rd ← $rt – #n
Load	lw $rd, addr($rt)	$rd ← memory[$rt + addr]
Store	sw $rd, addr($rt)	$rd → memory[$rt + addr]
And	and $rd, $rt, $rs andi $rd, $rt, #n	$rd ← $rt & $rs $rd $rd ← $rt & #n
Or	or $rd, $rt, $rs ori $rd, $rt, #n	$rd ← $rt | $rs $rd $rd ← $rt | #n
Jump	jmp addr	PC ← addr
Brach Conditional	beq addr	PC ← (RC[2]==1) ? PC+BD : PC+4

***Table 1**: Instructions to be implemented by this processor*

In order to design this processor, first an ALU must be designed to handle arithmetic and logical instructions. Then, using predefined blocks such as instruction and data memory, a register read/write block to hold data and memory locations, control units and mulriplexers, the general datapath can be designed.

Design

The processor design consists of several sections: Machine Code, ALU design, Datapath design, and Control Unit design.

Machine Code

First, the instruction set must be converted into 32-bit machine code instructions which consist of 6-bit OpCodes and a 9-bit FUNC field, in the case of R-Type instructions. These values can be found in Table 2. The idea in doing this was to keep all R-type ALU instructions' Opcode the same so that the FUNC field will determine which ALU operation to perform. Next, I-type ALU instructions were given a bit in common (Opcode[5]), which keeps instructions like jump, branch, load word, and store word from sending incorrect instructions to the ALU.

Operation	OPCODE	FUNC
add	00 0000 = 0x00	0 0000 0010 = 0x002
addi	10 0110 = 0x26	x
sub	00 0000 = 0x00	0 0000 0011 = 0x003
subi	10 0111 = 0x27	x
lw	00 0110 = 0x03	x
sw	00 0111 = 0x07	x
and	00 0000 = 0x00	0 0000 0000 = 0x000
andi	10 0100 = 0x24	x
or	00 0000 = 0x00	0 0000 0001 = 0x001
ori	10 0101 = 0x25	x
jmp	01 0000 = 0x10	x
beq	01 1000 = 0x18	x

Table 2: *Opcodes and FUNC field for this processor's instruction set*

The Opcode field will, through the Control Unit, set the 3-bit ALU_CTRL bus that will, through the ALU Control Unit, set the ALU_OPCODE control signals. These values can be found in Table 3. The idea for these codes was to make the Opcode, ALU_CTRL, and ALU_OPCODE share as many similar values as possible, as illustrated by the similarities in machine code values found in Table 4.

Operation	OPCODE	ALU_CTRL
arithmetic	00 0000	000
lw	00 0110	110
sw	00 0111	110
addi	10 0110	110
subi	10 0111	111
andi	10 0100	100
ori	10 0101	101

Table 3: *Relationship between Operation, Instruction Opcode, and ALU_CTRL signal*

Operation	ALU_CTRL	FUNC	ALU_OPCODE	ALU Op
arithmetic	000	add = 0x002	10	add
	000	sub = 0x003	11	sub
	000	and = 0x000	00	and
	000	or = 0x001	01	or
lw/sw	110	x	10	add
addi	110	x	10	add
subi	111	x	11	sub
andi	100	x	00	and
ori	101	x	01	or

Table 4: *Relationship between Operation, ALU_CTRL, FUNC field, and ALU_OPCODE*

ALU Design

General Design

Table 5 describes which ALU_OPCODE corresponds to what ALU Operation. Since addition and subtraction both require addition, they both share ALU_OPCODE[1] in common, ALU_OPCODE[0] will determine whether or not to invert the *B* operand.

From Table 4, it can be seen that the Boolean expressions used by the ALU Control unit to determine ALU_OPCODE[0:1] are given by:

$$ALUOPCODE[0] = (ALUCTRL[2])(ALUCTRL[0]) + (\overline{ALUCTRL[2]})(FUNC[0])$$
$$ALUOPCODE[1] = (ALUCTRL[2])(ALUCTRL[1]) + (\overline{ALUCTRL[2]})(FUNC[0])$$

It is obvious that these equations represent multiplexers with control signal ALU_CTRL[2], allowing for a simple hardware implementation.

The ALU consists of 32 1-bit ALU blocks, which include a multiplexer that determines whether operands *A* and *B* will be *ANDed*, *ORed*, *added*, or *subtracted* (by performing two's-complement addition). This makes use of ripple-carry addition, which is not the quickest method of performing addition, but it makes it much easier to design, allowing for the 1-bit blocks to be built. Figure 1 contains the design of the ALU, showing only the first two, and last bits: ALU[0], ALU[1], and ALU[31]. ALU[2:30] are not necessary to show, since they are identical to ALU[1].

The block ALU[0] receives the *subtraction* signal, though CarryIn, in the form of the logical expression, (`ALU_OPCODE[0] & ALU_OPCODE[1]`), which determines when the subtraction operation is performed.

ALU_OPCODE	ALU Operation
00	and
01	or
10	addition
11	subtract

***Table 5**: ALU_OPCODE and the corresponding operation performed*

Status Flags

The status flags, which will be stored in the RC register if the control signal dictates it, are the following: S (*sign*), CY (*carry*), EQ (*equal*, or *zero*), and OV (*overflow*). The equal flag is also the zero flag because if two numbers are subtracted, and the result is 0, then they must be equal; this is how computers determine whether two numbers are equal or not. The *Sign* flag is simply the highest bit of the ALU's result, since this is how a MIPS processor stores signed numbers. The *Carry* flag is just the carry output from the highest bit of the ALU, CarryOut[31]. The *Equal,* or *Zero,* flag is a 32-input *NOR* gate from the output of every ALU bit. This will output a logical '1' only if every bit output from the ALU is a logical '0'. The *Overflow* flag is thrown by *XORing* the highest two carry output from the ALU, CarryOut[30] and CarryOut[31]. This is due to the nature of signed numbers and how they are stored. The Overflow flag being thrown means that the number has exceeded its magnitude in either the positive or negative direction; meaning the sign bit has swapped when it shouldn't have and the number is now incorrectly the wrong sign.

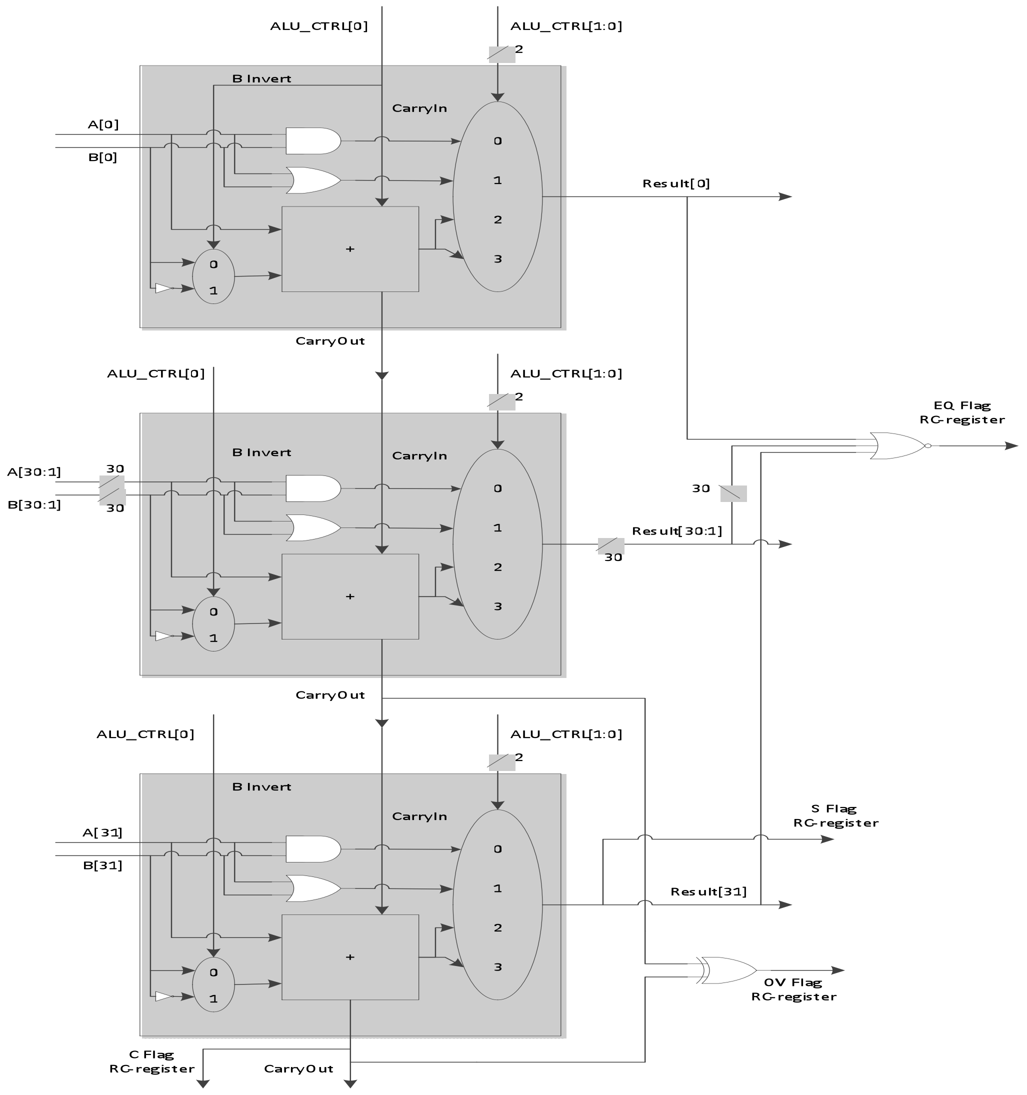

***Figure 1**: ALU Design: ALU[0], ALU[1], ALU[31]*

Datapath

The datapath for this processor requires the following components:

Adders – to perform PC+4 for the next instruction, or add branch addresses to PC
Instruction Memory – to decode the instruction and pass the correct values to Registers
Registers – to read and write data to and from registers used in an instruction
ALU – discussed in the previous subsection, to perform arithmetic and logical operations
ALU Control – to pass the correct ALU_OPCODE to the ALU based on the instruction and signal from the Control Unit
Control Unit – to take information from the Instruction Opcode
Sign Extend – to extend the 16-bit Immediate value used in I-type instructions to 32-bit
Data Memory – RAM memory used to store data outside of registers; for additional information
RC Register – to store the flags set by R-type or ALU-based I-type instructions
Append2 - to append the most significant bits of PC onto branch or jump addresses

These components are linked together and used during certain instruction executions. Figures 2-8 contain the datapaths used by this processor:

Figure 2: R-type
Figure 3: I-type
Figure 4: Load word
Figure 5: Store word
Figure 6: Jump
Figure 7: Branch
Figure 8: Full datapath

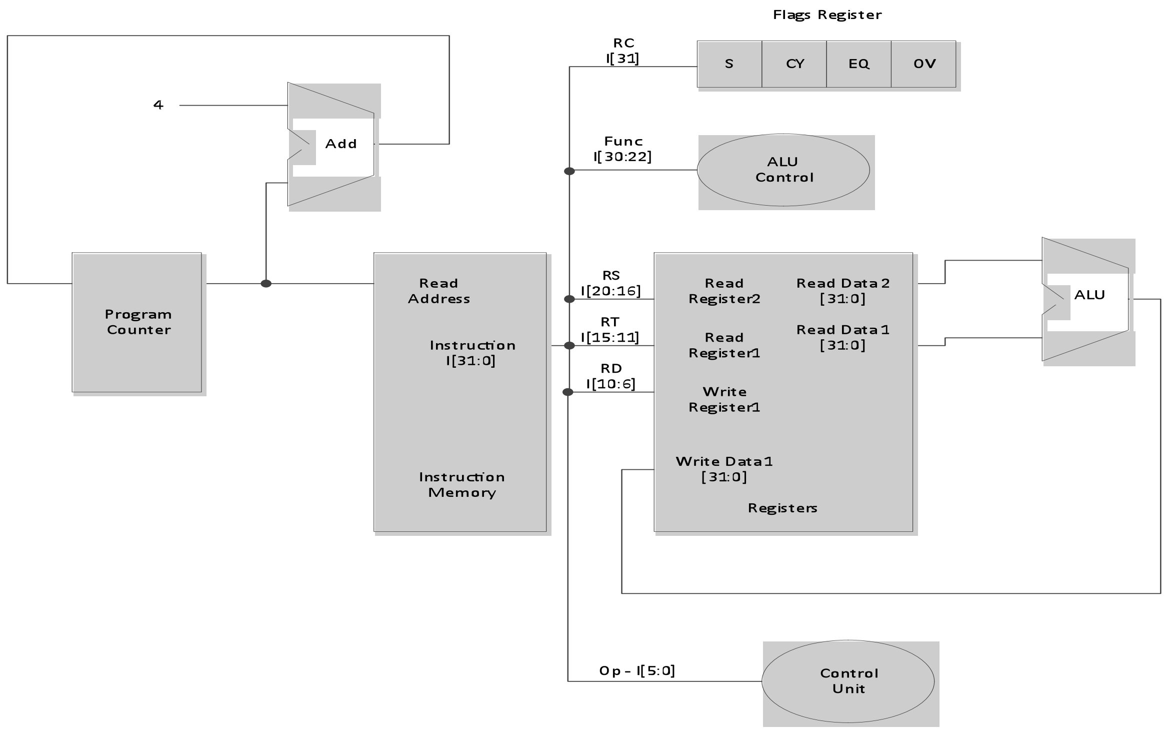

***Figure 2**: R-type **(add, sub, and, or)** instruction datapath*

The R-type instruction takes the values in the registers referred to by $RS and $RT, performs an operation on them in the ALU and writes the result into the register referred to by $RD. The Program Counter is incremented by four to access the next instruction in the instruction memory.

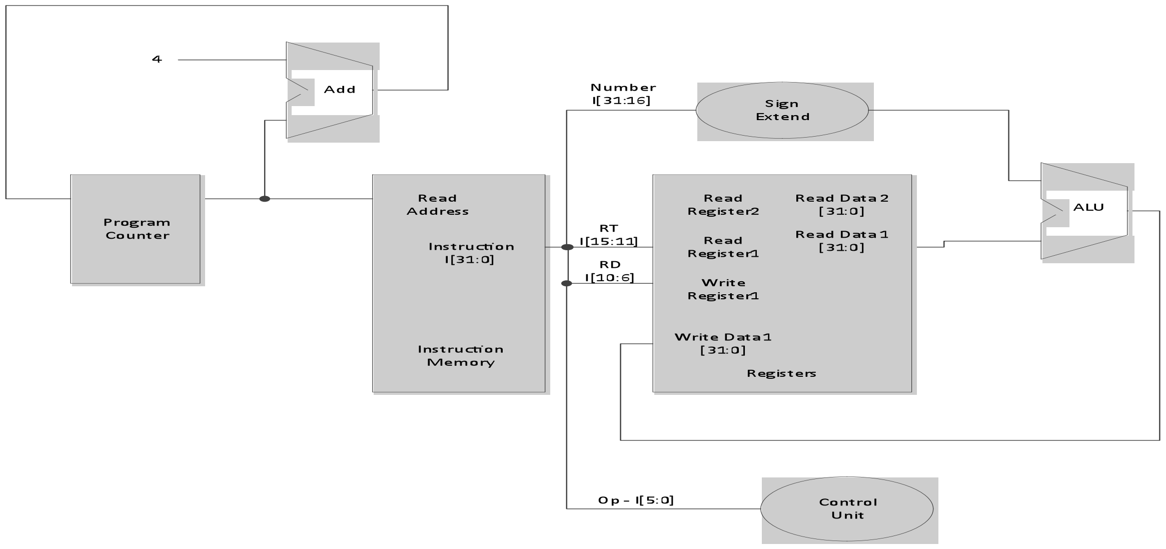

***Figure 3**: I-type (**addi, subi, andi, ori**) instruction datapath*

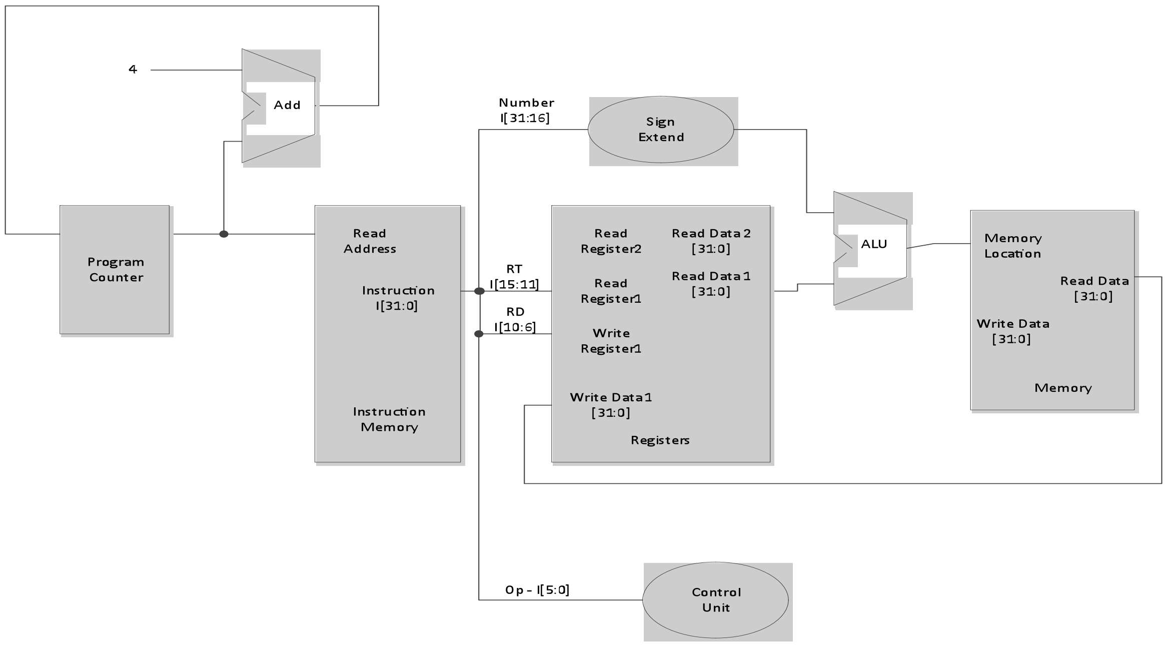

***Figure 4**: Load Word (**lw**) instruction datapath*

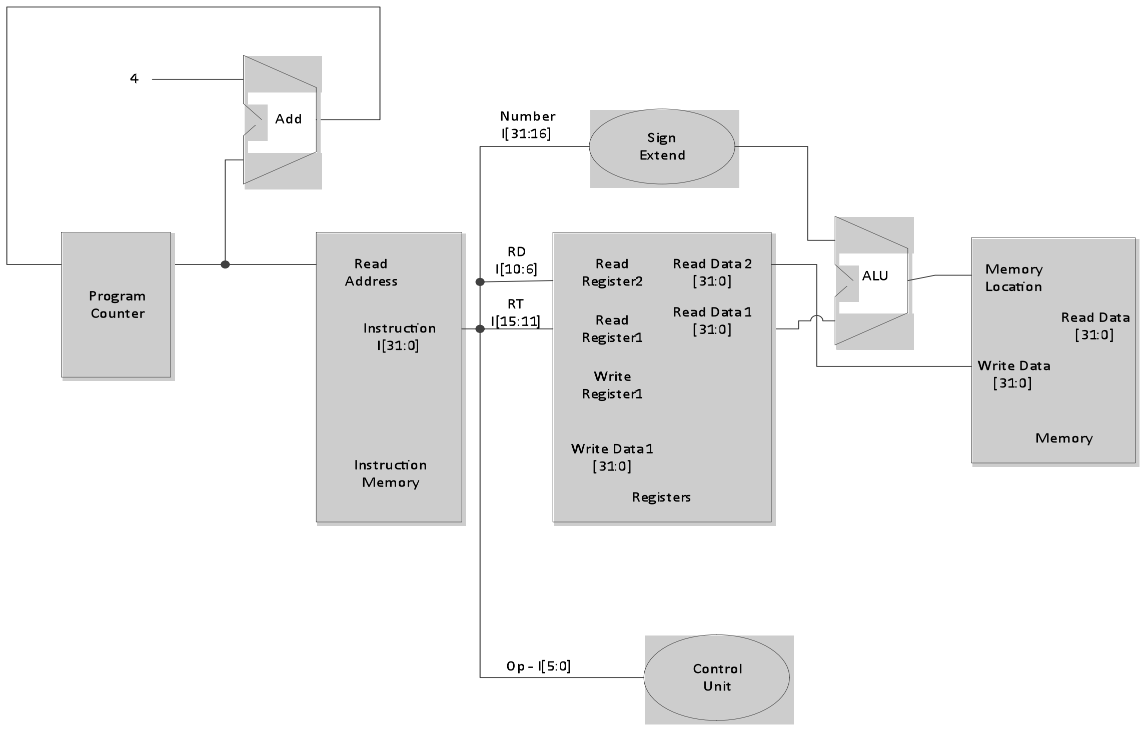

***Figure 5**: Store Word (**sw**) instruction datapath*

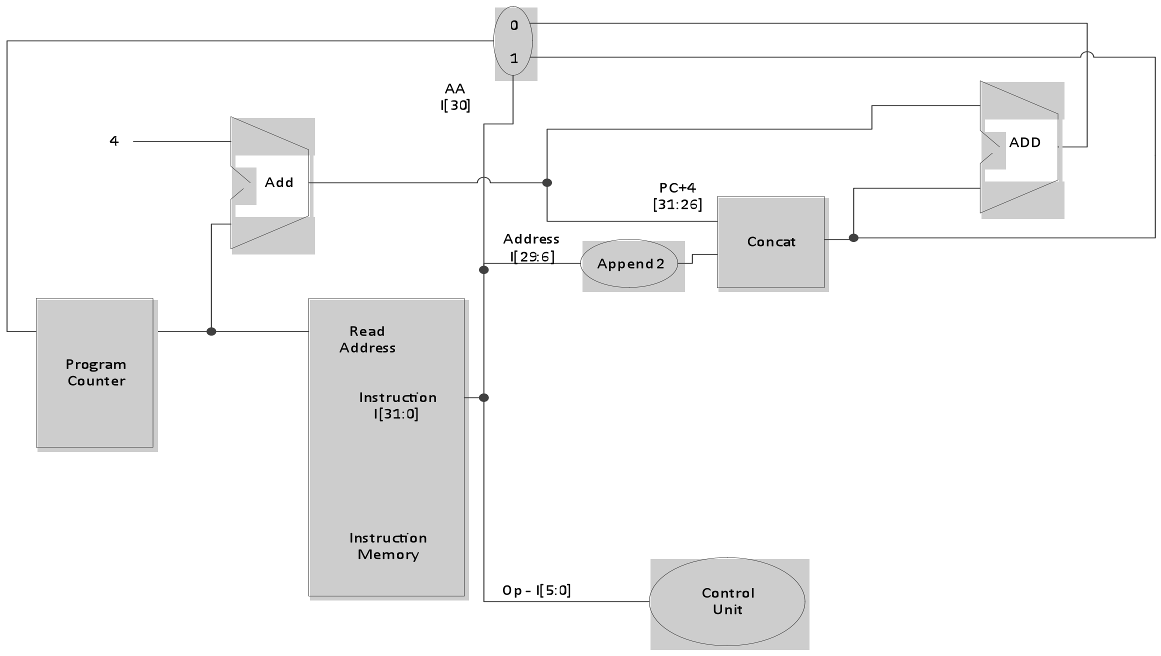

***Figure 6**: Jump (**jmp**) instruction datapath*

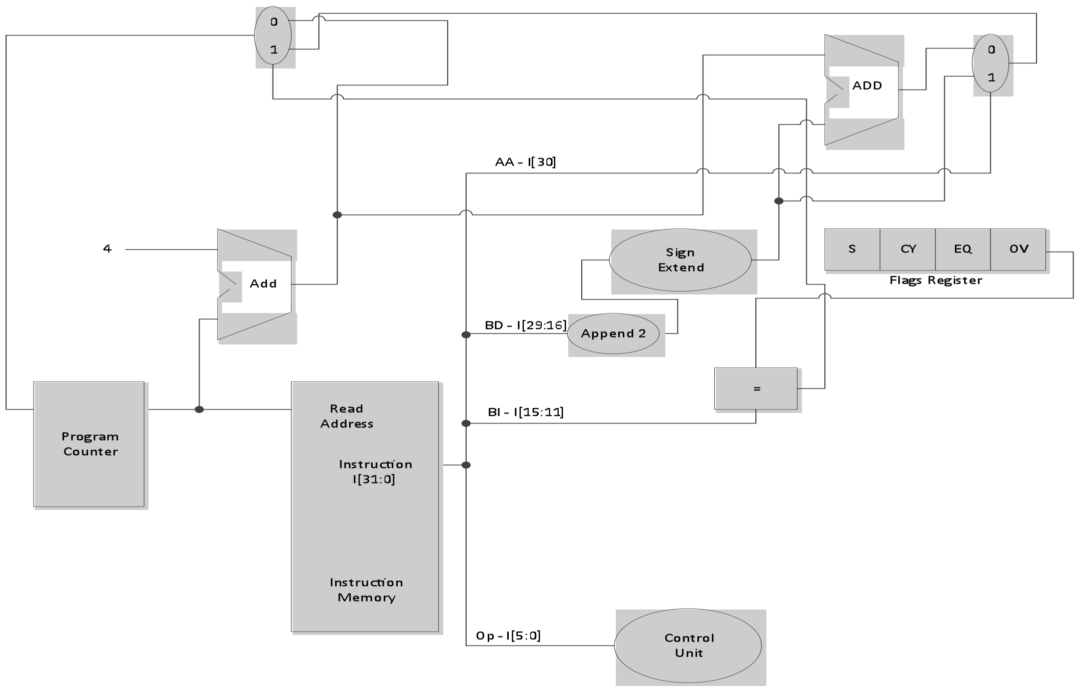

Figure 7: *Branch Conditional (**beq**) instruction datapath*

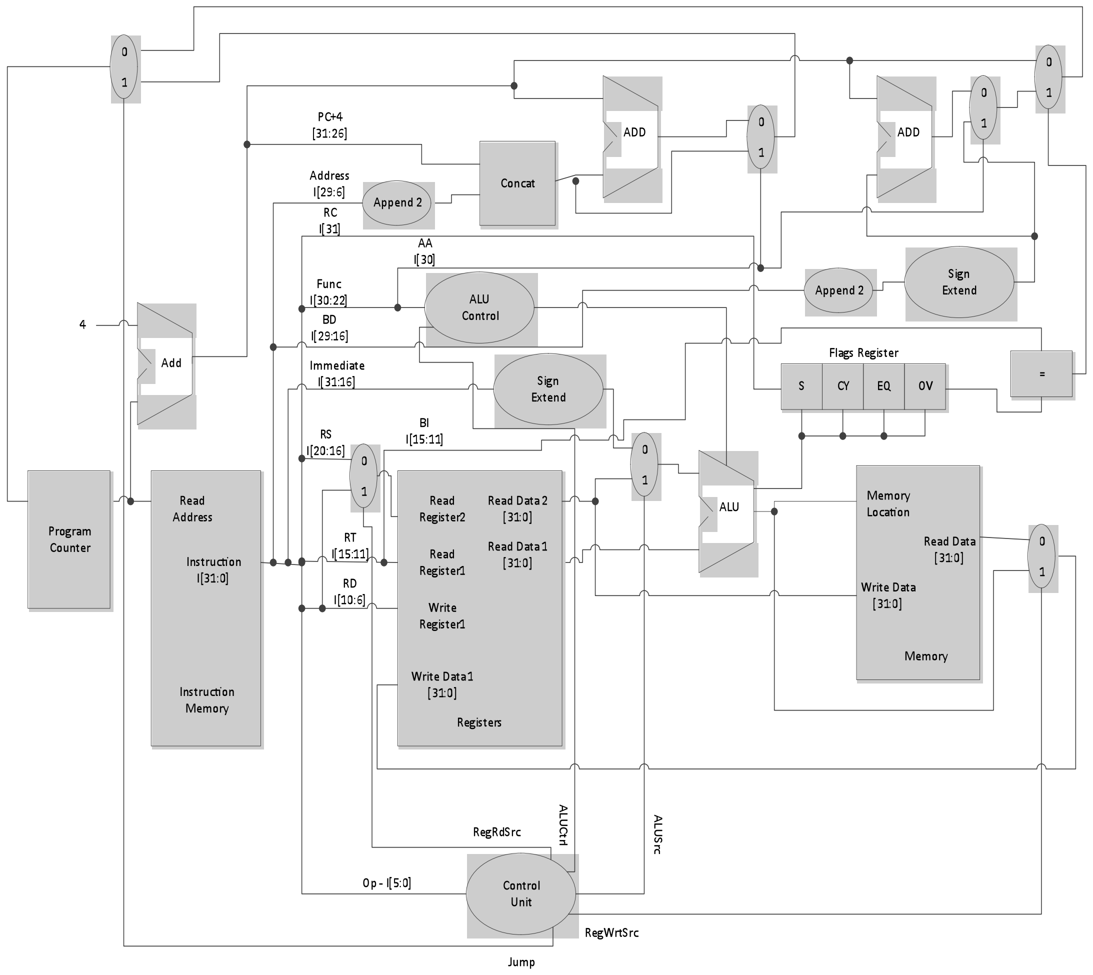

Figure 8: *Full Datapath*

Control Unit

The control unit takes in the instruction Opcode (instr[0:5]) and determines which blocks on the datapath to select, by sending control signals to multiplexers or to the ALU Control unit (discussed in the "ALU Design" section). The signals it outputs are:

RegRdSrc – to determine which registers are read (determines R-type or I-type instruction)
ALU_CTRL – to determine which ALU Operation performed. Sent to ALU Control
ALUSrc – to determine whether the first ALU operand should be from registers or the sign-extended Immediate value
RegWtSrc – to determine if data memory or ALU output should be written to register $rd
MemWt – to determine if data memory should be written to
MemRd – to determine if data memory should be read from
SetFlags – to determine if the flags in the RC register should be written to (R-type or ALU I-type instructions)

Table 6 illustrates the relationship between Opcode, FUNC field, and the outputs of the Control Unit.

RegRdSrc is given a '1' during any non-R-type instructions, because $rs is not used in these instructions. Instead, $rd is used in its place and the 16-bit immediate address is sent through sign extend to the ALU.

The 3-bit *ALU_CTRL* signal is sent to the ALU Control unit (discussed under "Design/ALU Design/*General Design*" and determines which ALU Operation is performed. The relationship between Opcode, FUNC field, ALU_CTRL, and ALU_OPCODE are illustrated in Tables 3 and 4.

ALUSrc is set to '1' when an I-type instruction is performed. This is because the operand must come from the sign-extended Immediate value, rather than from a register.

RegWtSrc is set to '1' when the output of the ALU is to be directly written to register $rd (*add, addi, sub, subi, and, andi, or, ori*), and 0 when data memory is to be loaded and written to $rd (*lw*).

MemWt is set to '1' when data memory is to be written to (*sw*), and '0' otherwise.

MemRd is set to '1' when data memory is to be read from (*lw*), and '0' otherwise.

SetFlags is set to '1' when the flags in the RC Register are to be written to, which happens for an R-type instruction (*add, sub, and, or*), or an I-type instruction which uses the ALU for more than just adding addresses (*andi, subi, andi, ori*). The RC bit in the instruction is always '1' during R-type instructions, but can vary during other types of instructions, so the Boolean expression which describes it is given by:

$$SetFlags = (RC)(\overline{OPCODE[5:0]}) \ + \ (OPCODE[5])$$

Which says that when the Opcode is 0x00 (which it is for all R-type instructions), use the RC bit, which will be '1' for all R-type instructions. When the Opcode is not 0x00, then set the RC flags when Opcode[5] is '1,' which is true for all I-type instructions that use the ALU (*addi, subi, andi, ori*).

Operation	Opcode	FUNC	RegRdSrc	ALU_CTRL	ALUSrc	RegWtSrc	MemWt	MemRd	SetFlgas
add	0x00	0x002	0	000	0	0	x	x	1
addi	0x26	x	1	110	1	0	x	x	1
sub	0x00	0x003	0	000	0	0	x	x	1
subi	0x27	x	1	111	1	0	x	x	1
and	0x00	0x000	0	000	0	0	x	x	1
andi	0x24	x	1	100	1	0	x	x	1
or	0x00	0x001	0	000	0	0	x	x	1
ori	0x25	x	1	101	1	0	x	x	1
lw	0x03	x	1	110	1	1	0	0	0
sw	0x07	x	1	110	1	x	1	1	0
beq	0x10	x	1	x	x	x	x	x	0
jmp	0x18	x	1	x	x	x	x	x	0

***Table 6**: Relationship between Opcode, FUNC field, and the output of the Control Unit*

Conclusion

A MIPS-like processor running a RISC-style architecture is quite simple to implement, especially when only a few instructions are considered at first. Adding instructions to this architecture is not difficult, as it only involves choosing new datapaths, opcodes, and editing the control unit to handle these new instructions. The basic blocks used for every instructions will stay the same (unless new ALU operations, such as multiply and divide are added), but several multiplexers with new control signals will most likely need to be added to determine which signal is to be chosen.

RISC processors may take a few extra lines in assembly to program, due to the simplistic nature of their instructions, but they can perform these operations, in general, faster than processors which use variable-sized instruction sets, and are far easier to design hardware for.

To make this design more efficient, hardware for pipelining can be added. This requires registers in between each "section" of blocks, and several control units for hazard detection and forwarding. This would require the addition of several new blocks, for the control units, and multiplexers, to determine if forwarding should be done or not.

This design can be rather simply implemented in an FPGA by writing several Verilog modules, for each block on the datapath, and linking them together with the correct wires, multiplexers, and control signals. A compiler and assembler can also be written to interpret C code and assembly language, making this processor more versatile.