CPU Design Project Part 2 - Datapath

Joe Driscoll

February 15, 2019

Contents

1 Introduction

This project report contains the single-cycle datapath designed for the ISA created in the previous project part, as well as various pieces of information about it. In addition to a visual representation of the datapath, the report demonstrates how the datapath would be utilized by each instruction from a register level, as well as at an even lower level with control signals.

2 Register-level datapath

This section includes an image of the datapath designed for the ISA designed earlier. Everything in the datapath are shown in Figure 1. The control signals (red) in the diagram are outputs of the controller block, but the signals are not shown as outputs of the controller block to save space and neatness. There are also a few inputs to the controller not shown for the same reason (such as a few bits from the instruction register). These details will be developed in the next project part, where the controller will be discussed in depth. The various control signals on the multiplexers allows for the execution of all the instructions in the ISA. The memory block on the right side of the diagram shows the memory interface between the processor and the memory, based on the von Neumann architecture. The connections between the memory and other components are mostly direct, but there is one multiplexer to select the read address to distinguish between instruct fetches and memory reads.

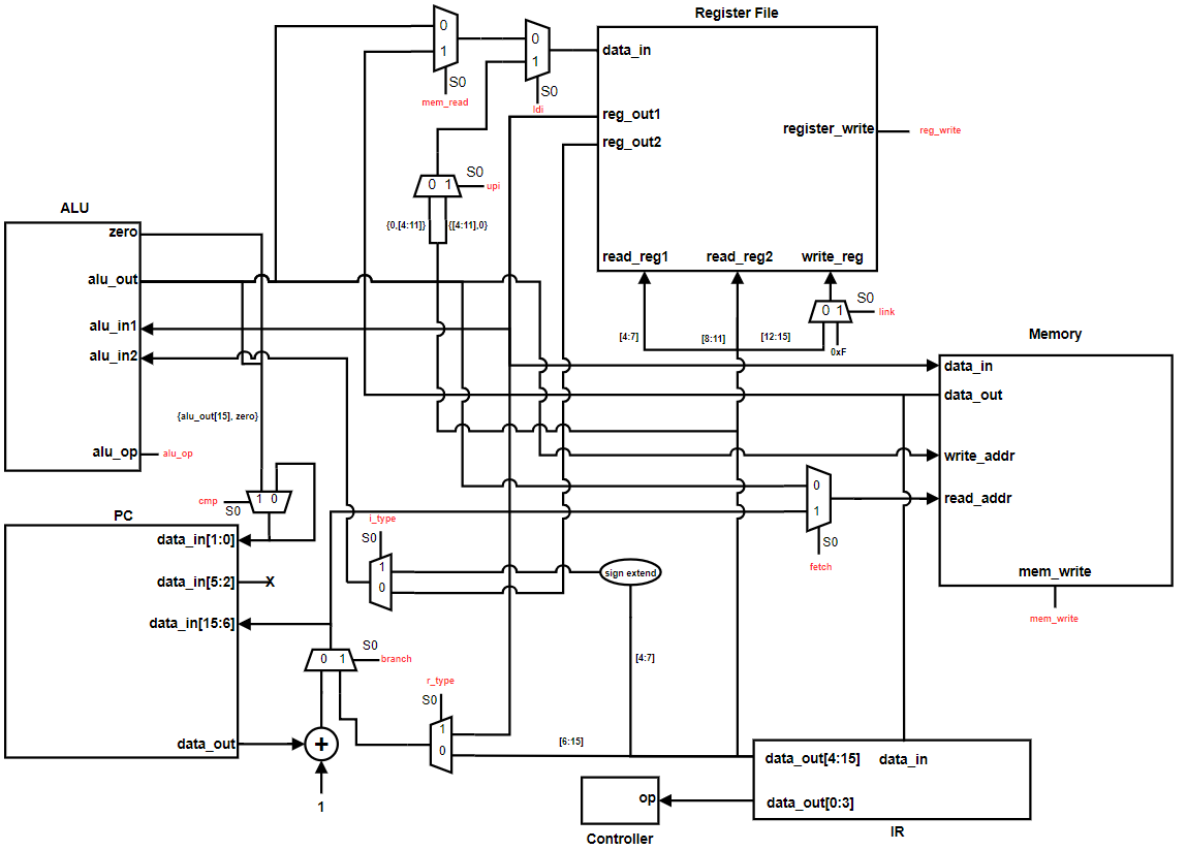

Figure 1. Datapath for the ISA designed in the previous project.

3 Datapath components

This section described each of the main parts of the datapath shown in Figure 1. These parts consist of the program counter, register file, the ALU, the memory, the controller, and the multiplexing logic interconnecting all of it. Each of the inputs and outputs referenced in the following section are the same ports shown in Figure 1.

3.1 Program Counter (PC)

The program counter consists of two main ports: data_in and data_out. The data_in port holds data that is to written to the register on a clock edge, while data_out shows the current value held by the register. As seen in the datapath, various bits of the data_in are driven by various signals, giving good reason to make the PC separate from the register file so as to not complicate the register file interface. The clock that will trigger a write to the register is not shown, as the dedicated registers (as discussed earlier) are synchronous with the system clock (where each cycle executes one instruction). The purpose of the PC is to point to the next instruction to fetch from memory, allowing for effective multiplexing during the fetch portion of the fetch-execute process.

3.2 Register File

The register file consists of a few main parts, namely the read register numbers, register data out, write register number, the data input, and the register write enable. The read register numbers (read_reg1 and read_reg2) determine which register numbers to output on the register output data buses (reg_out1 and reg_out2). The write_reg port determines which register the data on the data_in port will be written to. Lastly, the register_write port enables the writing of a register (writes data_in to write_reg). The register file holds all of the registers available to the user ($0-$15), which is one of the fundamental things made available in an ISA to the programmer. The instructions in the ISA use registers in the file as sources and destinations, making it a vital component.

3.3 Arithmetic Logic Unit (ALU)

Currently, the ALU consists of three main ports: the two inputs and the output. The ALU also has an input that comes from the controller that determines what arithmetic/logical operation the ALU should perform on the inputs, and it currently has one flag - the zero flag. The zero flag is driven high when the output of the ALU is 0, and this is used for some comparison-related operations in the current datapath design. The ALU performs all arithmetic and logical operations that the instructions of the ISA encompass. Most of the instructions will use the ALU in some capacity, making it one of the most important components.

3.4 Memory

While the memory isn't exactly part of the processor design, its interface with the processor is important. Instructions are directly read from memory, and some instructions operate on memory directly. The interface for the memory shown in Figure 1 demonstrates the primary functions needed to interface with the memory, like determining the memory read and write address, the write value, the output of the memory at the read address, and the ability to write to the memory. Some multiplexing logic surrounds the memory, but this is discussed in the last section, which discusses each of the multiplexers in detail.

3.5 Controller

While the controller is not shown in detail on the datapath diagram, it is still an important component of the system. Future project reports and design work will expand upon the exact operation of the controller. Directly relevant to the datapath are the control signals that come out of the controller (control signals shown in red in Figure 1, not shown as outputs of controller block to save space). These control signals direct the datapath, determining the direction of basically every multiplexer in the datapath and having a direct influence on the routing of the datapath. Currently, the only input to the controller is the op-code,

as well as bits (4-5) of the instruction register. These are used to decode the opcode, and the two extra bits are used for logic concerning conditional branching. Again, much of these details will be established in later reports, but these inputs along with the control signals shown in the datapath are the high-level details of the controller relevant to the datapath.

3.6 Multiplexers

The last component in the datapath is really multiple components, as many multiplexers are inside of the datapath to support every instruction in the ISA. The purpose of each multiplexer, denoted by the control signal they use, is shown in the sections below.

3.6.1 cmp

The multiplexer belonging to the cmp signal chooses whether to keep the current status bits in the PC (the lower two bits) or to load in the upper bit and the zero flag from the ALU to set two new status bits. This is used in the compare instructions, where the result of a subtraction in the ALU will be used to determine whether one value is less than or equal to another.

3.6.2 i_type

The multiplexer belonging to the i_type signal chooses whether the value of a register or an immediate value in the instruction should be in the second input of the ALU. This is used to distinguish between R-type and $I_{1,2}$-types, as R-type instructions use the value of a register in the ALU while the I-types use immediate values in the instruction as an input to the ALU.

3.6.3 r_type

The multiplexer belonging to the r_type signal allows the different types of branch instruction to be executed (immediate vs. register branching). The output of this multiplexer is fed to the input of the multiplexer that uses the branch signal, which is discussed next.

3.6.4 branch

The multiplexer belonging to the branch signal determines whether PC should be loaded with itself incremented by one, or if it should be loaded with either a register value or an immediate value.

3.6.5 link

The multiplexer belonging to the link signal is used to automatically write to the link register upon bl instructions. This is needed because the write_reg port of the register file only comes from the instruction, so without this multiplexer, saving the incremented PC value to the register would not be possible. The constant 0xF is the register number of the link register.

3.6.6 fetch

The multiplexer belonging to the fetch signal is used to determine whether the next instruction should be read (memory read address = PC) or if another memory address should be read for say load instructions (memory read address = base pointer + offset).

3.6.7 mem_read

The multiplexer belonging to the mem_read signal determines whether the input data to the register file should come from memory or from the output of the ALU. Memory reads must write data from the memory, while practically every other instruction type must take the output of some operation from the ALU as the input data (like an add instruction).

3.6.8 ldi

The multiplexer belonging to ldi determines whether a load immediate instruction is occurring, as an eight-bit constant must be multiplexed to the input of the register file. This multiplexer along with the ldi signal control that action.

3.6.9 upi

The multiplexer belonging to upi determines whether a load immediate is an upper or lower immediate to determine which side of the constant to pad with zeros. To give some insight on the name, upi stands for "upper immediate".

4 Register transfers for instructions

This section contains the register transfers that are needed to fetch and execute each instruction in the ISA. For more information on what each instruction does from a higher level perspective, refer to the previous project report that details the function and format of each instruction in detail. In each case, GPR[i] stands for General Purpose Register i. This is used in the MIPS Reference Manual, so this notation will be used throughout for relatability. Also, it should be noted that no register transfers occur directly between the ALU and other registers, as the inputs and output of the ALU are not registers. This is a consequence of using a single-cycle datapath, as multi-cycle datapaths do contain these registers. Thus, the ALU is implicitly involved in the register transfers with the logical and arithmetic operators used in the expressions, but not explicitly in that the ports of the ALU are not directly set.

4.1 Some definitions

The following definitions will be used throughout each of the examples. This makes the register transfer expressions more concise and readable.

```
rd      := IR[15:12]
rs1     := IR[11:8]
rs2     := IR[7:4]
cond    := IR[5:4]
address := IR[15:6]
kk      := IR[11:4]
k       := IR[7:4]
```

4.2 add

```
IR        ← memory[PC[9:0]]
GPR[rd]   ← GPR[rs1] + GPR[rs2]
PC[15:6]  ← PC[15:6] + 1
```

4.3 sub

```
IR        ← memory[PC[9:0]]
GPR[rd]   ← GPR[rs1] - GPR[rs2]
PC[15:6]  ← PC[15:6] + 1
```

4.4 str

```
IR                          ← memory[PC[9:0]]
memory[GPR[rs1] + GPR[rs2]] ← GPR[rd]
PC[15:6]                    ← PC[15:6] + 1
```

4.5 ldr

```
IR       ← memory[PC[9:0]]
GPR[rd]  ← memory[GPR[rs1] + GPR[rs2]]
PC[15:6] ← PC[15:6] + 1
```

4.6 and

```
IR       ← memory[PC[9:0]]
GPR[rd]  ← GPR[rs1] & GPR[rs2]
PC[15:6] ← PC[15:6] + 1
```

4.7 or

```
IR       ← memory[PC[9:0]]
GPR[rd]  ← GPR[rs1] | GPR[rs2]
PC[15:6] ← PC[15:6] + 1
```

4.8 not

```
IR       ← memory[PC[9:0]]
GPR[rd]  ← not(GPR[rs1])
PC[15:6] ← PC[15:6] + 1
```

4.9 cmp

```
IR       ← memory[PC[9:0]]
PC[1:0]  ← {(GPR[rs1] - GPR[rs2])[15], zero}
PC[15:6] ← PC[15:6] + 1
```

4.10 br

```
IR       ← memory[PC[9:0]]
PC[15:6] ← GPR[rd]
```

4.11 b(,eq,lt,gt)

```
IR ← memory[PC[9:0]]

if (cond == blank):
    PC[15:6] ← address

elsif (cond == eq and (PC[0] == 1)):
    PC[15:6] ← address

elsif (cond == lt and (PC[1] == 1)):
    PC[15:6] ← address

elsif (cond == gt and (PC[1] == 0)):
    PC[15:6] ← address

else:
    PC[15:6] ← PC[15:6] + 1
```

4.12 bl(,eq,lt,gt)

```
IR ← memory[PC[9:0]]

if (cond == blank):
    GPR[$15] ← PC[15:6] + 1
    PC[15:6] ← address

elsif (cond == eq and (PC[0] == 1)):
    GPR[$15] ← PC[15:6] + 1
    PC[15:6] ← address

elsif (cond == lt and (PC[1] == 1)):
    GPR[$15] ← PC[15:6] + 1
    PC[15:6] ← address

elsif (cond == gt and (PC[1] == 0)):
    GPR[$15] ← PC[15:6] + 1
    PC[15:6] ← address

else:
    PC[15:6] ← PC[15:6] + 1
```

4.13 loadil

```
IR       ← memory[PC[9:0]]
GPR[rd]  ← kk
PC[15:6] ← PC[15:6] + 1
```

4.14 loadiu

```
IR       ← memory[PC[9:0]]
GPR[rd]  ← (kk << 8)
PC[15:6] ← PC[15:6] + 1
```

4.15 addi

```
IR       ← memory[PC[9:0]]
GPR[rd]  ← GPR[rs1] + k
PC[15:6] ← PC[15:6] + 1
```

4.16 lsr

```
IR       ← memory[PC[9:0]]
GPR[rd]  ← (GPR[rs1] >> k)
PC[15:6] ← PC[15:6] + 1
```

4.17 lsl

```
IR       ← memory[PC[9:0]]
GPR[rd]  ← (GPR[rs1] << k)
PC[15:6] ← PC[15:6] + 1
```

5 Instruction control signals

This section contains a two truth tables that show the values of all the control signals for each instruction. Because there are too many control signals to fit into one table, the otherwise single table is split in two. These tables demonstrate the purposes of each control signal, and they serve as a reference and guide for

later implementation of the controller itself. The mem_read, mem_write, and reg_write control signals in Figure 1 are abbreviated to m_read, m_write, and r_write respectively to save space in the table.

Table 1: Control signal values for each instruction in the ISA.

op	alu_op	cmp	i_type	r_type	ldi	upi
add	000	0	0	0	0	0
sub	001	0	0	0	0	0
str	000	0	0	0	0	0
ldr	000	0	0	0	0	0
and	010	0	0	0	0	0
or	011	0	0	0	0	0
not	100	0	0	0	0	0
cmp	XXX	1	0	0	0	0
br	XXX	0	0	1	0	0
b[1]	XXX	0	0	0	0	0
bl	XXX	0	0	0	0	0
loadil	XXX	0	0	0	1	0
loadiu	XXX	0	0	0	1	1
addi	000	0	1	0	0	0
lsr	101	0	1	0	0	0
lsl	110	0	1	0	0	0

[1]Signals are not shown for the beq, blt, bgt, as the branch control signal would depend on the status bits in the PC. The unconditional branch does not have this dependency, so it is shown here. The same remark applies to the bl instruction.

Table 2: Control signal values for each instruction in the ISA.

op	fetch	m_read	m_write	r_write	link	branch
add	0	0	0	1	0	0
sub	0	0	0	1	0	0
str	0	0	1	0	0	0
ldr	0	1	0	1	0	0
and	0	0	0	1	0	0
or	0	0	0	1	0	0
not	0	0	0	1	0	0
cmp	0	0	0	0	0	0
br	0	0	0	0	0	1
b	0	0	0	0	0	1
bl	0	0	0	1	1	1
loadil	0	0	0	1	0	0
loadiu	0	0	0	1	0	0
addi	0	0	0	1	0	0
lsr	0	0	0	1	0	0
lsl	0	0	0	1	0	0

6 Design discussion

In terms of the speed versus cost tradeoff, cost was mostly ignored in favor of speed. In reality, time limits the design of this sixteen-bit processor to be only so complex, and there is no requirement for cost or even resource utilization of the design. Ideally, the design can be synthesized and implemented on a readily available FPGA, but the design would need to be incredibly complex for this to not be possible. To this end, no compromises are currently or plan on being made on speed with regards to cost/resource use. The main design objective for the current design and later design is to make the processor as quick and efficient as possible, disregarding cost unless complexity becomes a noticeable issue.

The choice of implementing a single-cycle datapath was two-fold. First, single-cycle datapaths are relatively simple to implement, as there is no complicated sequential logic needed to design it. Secondly, multicycle and pipelined datapaths had not been introduced in class at the time of this design, so neither were used to prevent the pitfalls resulting from not being familiar with either scheme.

The current design only consists really of two dedicated/specialized components: the ten-bit adder and the sign extender. As mention earlier, cost is not yet and likely will not become an issue, and the inclusion of relatively simple, dedicated hardware will not cost nearly enough to introduce reason to not take the speed and convenience they provide. The exact design of the dedicated components has not been done yet, but there could also be some optimizations done for both. For example, the dedicated adder is always adding 1, so the adder does not necessarily need to be a full-adder, as it basically has a constant carry-in. Facts like this further reduce the complexity of the inclusion of dedicated components, making the gain in speed and practicality well worth the cost.

Many of the reserved registers, like the PC and the IR, are synchronous with the system clock, as they only need to be updated once per clock cycle. Other components, like the memory, register file, ALU, are not synchronous with the same clock, as they must perform operations within a clock cycle.

A Design updates/changes

A.1 Status bits and xor removal

A vast majority of the design from the former project was kept, but two main things changed. First, two bits are used in the PC for holding status bits, as opposed to the three used before. Second, the "xor" instruction was replaced with the "not" instruction, as having the ALU support the xor would introduce needless complexity to it, while xor's can be performed with a composition of not's, and's, and or's. This slight change should make the resource usage in the ALU slightly smaller. The first change arose from trying to slightly simplify the datapath, but the number of bits used in the PC for status may change later, especially if exception bits are to be stored.

A.2 Register additions/removals

Before, the PC was considered a user-register that would apparently be in the register file. This is not convenient for the design of the datapath, so the PC was replaced by an additional $s register in the register file and made a dedicated register outside of the file. This decision is heavily reflected in the datapath, as the PC is not inside of the register file (and for good reason).