Abstract

This project's goal was to design a Central Processing Unit (CPU) incorporating a fairly large instruction set and a multistage pipeline design with the potential to be used in a multi-core system. The CPU was coded and synthesized with Verilog. This was accomplished by building on the CPU design from fundamentals learned in CSE320 and increasing the instruction set to resemble a proper Reduced Instruction Set Computing (RISC) CPU system. A multistage pipeline was incorporated to the CPU to increase instruction throughput, or instructions per second. A major area of focus was on creating a multi-core design. The design used is master-slave in nature. The master core instructs the sub-cores where they should begin execution, the idea being that the operating system or kernel will be executing on the master core and the “user space” programs will be run on the sub-cores. The rationale behind this is that the system would specialize in running several small functions on all of its many supported cores. The system supports around 45 instructions, which include several types of jumps and branches (for changing the program counter based on conditions), arithmetic operations (addition, subtraction, or, and, etc.), and system calls (for controlling the core execution). The system has a very low Clocks per Instruction ratio (CPI), but to achieve this the second stage contains several modules and would most likely be a bottleneck for performance if implemented. The CPU is not perfect and contains a few errors and oversights, but the system as a whole functions as intended.

Table of Contents

Design Goals

The overall objective for this project was to become more familiar with complex logical circuit design. This is why the classic five stage RISC pipeline was used, since it is relatively simple in terms of modern CPUs, yet a very solid starting point. The five stages would be Fetch, Decode, Execute, Memory Access, and Writeback.

I wanted a design that would have some sort of niche use, even in the modern age. It was eventually decided to have a massively parallel system, comprised of cheap and simple cores. This type of design would lend itself towards mathematical applications, since operations could be split up between several cores. As an example, if a person wanted to casually find prime numbers, they could run all the cores searching within ranges.

In order to meet this target, I would take the rather barebones five stage pipeline and add a few features that would closer lend itself to this ideal. These features include sleeping, a simple multi-core interface, and remaining in small footprint. Sleeping would allow the controller to accurately schedule procedures while remaining economic by freezing the clock. The emphasis on having a simple multi-core interface was an effort to remain end user friendly, since a tool should work for you and not against you. The goal to remain small would be an effort to keep the overall fabrication cost low, since a high cost would hardly be justifiable considering the final product.

Architecture

see Figure1.1 for visual representation - Pg. 13

The architecture is based on the classic five stage pipeline, similar to the MIPS32 processor. The five stages are Fetch, Decode, Execute, Memory Access, and Writeback. These stages were chosen because they were a good starting point for learning. The cores utilize a DDR RAM design, however this is inconsequential since two distinct memory units can be used.

The first stage, fetch, is where the instruction is pulled from memory. It contains the PC module and an adder which increments the PC by 4, since the memory is stored in 4 groups of 8bits, to construct a 32bit instruction word.

The second stage is where the instruction is decoded and signals are produced and operands for modules are determined. The most important module is the control, which parses the instruction and sets the proper control bits. The control unit also determines whether the bypass line, which holds the ALU's previous output, should be used for this instruction. This is important for when an instruction needs the output of the previous instruction for an operand so that no delay is needed. The register block is write-first, so this combined with the bypass line allows for typical data hazards to be avoided. The BranchCPU is the module the master CPU uses to control the sub-cores. The BranchCPU module loads sub-core PCs to the address determined by the master-core. The Branch-Jump module was moved forward, as compared to other designs, in order to determine a branch in one clock. This decision was an effort to reduce the CPI of the CPU, however it most likely created a bottleneck. The Clockwait module is in charge of taking the global system clock and feeding it to the CPU module. This allows cores to kill their clock, for sleep and program exit, without affecting the other cores.

The third stage is where the Arithmetic Logic Unit (ALU) executes. This stage's main focus is to run operands through the ALU. The possible inputs to the ALU are the previous ALU output from the bypass line, the two registers that are being used, or a signed or unsigned immediate value. The inputs are determined by the control unit from the previous stage. This stage is also where links are written to the registers in order to reduce CPI.

The fourth stage has the option to either write to the register block or write to the RAM memory. The decision to allow writing to the registers in this stage as opposed to the fifth stage in MIPS32 was made to avoid unnecessary stages. None of the arithmetic or logical operations use the RAM memory, so avoiding a useless stage was deemed a valuable optimization.

The fifth stage is where the load operation saves read data out of RAM into a register. This stage is only used by one instruction, Load Word. Removing this stage would have made the other stages unnecessarily complicated and muddled the distinctions of the stages.

Features

The system provides the ability to run programs in true parallelism. It was designed to have one program running on every core and not context switch very often. For this reason, the CPU would be limited to small embedded systems applications. Each CPU core provides 41 instructions and the master-core has 4 additional instructions to run the BranchCPU. Of the 41, all the usual instructions are supported; several jumps and branches, as well as the very common arithmetic and logical operations. See the OpCodes file for a complete index of supported instructions.

The sub-cores use a system called Runtimes. The register blocks on the sub-cores have 64 entries. However, only 32 of these registers are accessible at a time. A core has two runtime contexts it can run in, using either the top or bottom 32 registers in the block. This provides the advantage of low resource use while still providing the feature of context switches. The reason this implementation was chosen was because the necessity to have a backup module to store all register data into the memory is not present. With 32 worker cores, each with 2 Runtimes, it is highly improbable that an embedded system would utilize all the resources this system provides. In the instance where the need does arise that a new process needs to begin execution but no more room is available, there is still the option to run a software subroutine to manually backup the registers. When not restoring from memory, the execution can begin right where it left off, avoiding wasted cycles. The context switching is not fully tested, so problems could occur when switching between Runtimes.

At the moment there is no way to communicate between cores. It is possible to have a program write to a predetermined address in memory for another core's data, but the layout provided does not have a unified RAM. This means that the system only works for a core's two Runtimes, which do share RAM. The use of caches would resolve this limitation.

As mentioned in a previous section, data hazards are largely nonexistent. The only one that remains is with Load Word, which requires a two cycle delay for the data to be written to register. Otherwise, the code below will run without any delays needed.

```
ADD 1, 2, 3
ADD 2, 1, 4 – using bypass line for 1
ADD 2, 1, 6 – using bypass line for 2, pulling data out of write-first registers for 1
```

Programming

Interface

The interface used to program the CPU within a Verilog testbench is synchronous to the system clock. The **load** and **reset** signals should be pulled high in order to activate the programming mode. After this, simply specify the address in RAM memory you wish to write the instruction to and the instruction itself. These two values should be inputs to the CPU module instantiated within the testbench.

```
#1 reset=1;
#0 load=1; //pull reset and load signals high
#1 addr = 32'h60; //RAM address is 0x00000060
#0 instr = 32'b10100000001000010000000000000001; //ADDI 2, 2, #1
#1 addr = 32'h64; //RAM address is 0x00000064
#0 instr = 32'b00010100000000000000000000010111; //J 5c
```

When all instructions are loaded, simply pull down reset and load. The system will begin execution at address 0x00000000. This mode is only for testbench simulation; for synthesis and implementation a preprogrammed DDR RAM module must be used.

Programming Practices

A comprehensive list of instructions can be found in the Opcodes document.

It is recommended that the first two instructions of a process be NOPs, opcode 0x00000008. This allows the system to flush the first two stages and finish bootstrapping. After this, programs will behave as expected. System limitations and programming practices to be aware of are as follows:

SLEEP will stop the clock, so instructions in the pipeline will be stalled until the sleep in ended. If it is important that instructions be finished before SLEEP, NOPs should be used to delay SLEEP to allow for the prior instructions to finish.

It is illegal to try and write to the same register at the same time using two different instructions. For example, when using a LW followed by an ADD with the same destination register, they will reach their respective write stages at the same time. However, the register block will favor the ADD and ignore the LW in this situation. For this reason, the program should wait the necessary two cycles to finish a LW instruction to avoid glitches.

For the bypass line to work correctly, the **rs** field of the opcode must be the same as the prior instruction's **rd** field. Follow the code below as an example. The rationale behind this is that it is unnecessary and a waste of area to include a check on both operands. This also allows for the ability to use the original value if the programmer desires, similar to a nonblocking assignment in Verilog. The programmer or assembler must take this into consideration.

```
ADD rd, rs, rt – code format                 //c representation                      //Verilog
ADD 1, 2, 3 – initial instruction            d=c; c=a+b; //d=old c, c=new value
--followed by
ADD 3, 1, 2 – bypass will be enabled         b=c+a; //compute b using new c          c=a+b; b=c+a;
--or
ADD 3, 2, 1 – bypass will not be enabled     b=d+a; //compute b using old c          c<=a+b; b<=c+a;
```

The algorithm used to calculate whether bypass is needed or not defaults the most recent destination to register 1 for instructions that do not specify a destination, like Jump. This was an unfortunate decision and will be addressed in further revisions. However, for this reason, it is advisable to avoid using register 1 on any core. The groundwork for the fix is within the master-core control module, but not implemented due to lack of testing. The fix would increase the size of **reg1** in the control module by one bit, from 5bits to 6bits. Then when a Jump or similar instruction is received, **reg1** would be set to 6'b100000, rather than the current 6'b000001. A similar change would be made to the Control for sub-cores.

There is simple branch prediction. It will always be assumed that a branch will not be taken. If a branch is taken, the pipeline registers will be cleared; otherwise they will remain intact with the instructions following the branch instruction. This avoids the need for NOP after a branch or jump instruction.

For this version, the BCPU instruction is considered dangerous and therefore should be handled with care. It has not gone through thorough testing. For responsible use, verification of the program is needed to ensure no errors occurred from a context switch.

Conclusion

Overall the design goals for the system were met. The system is small, efficient, and functional. The system is not perfect and oversights were made, stemming from lack of experience. However, this personal project taught the intricacies of CPU design better than any class I could ever take as an undergraduate. With the experience I gained, I feel very confident in revising my design to create a final and potentially useful CPU.

Future Work

The future of this project will consist of major revisions. The groundwork for the next version will largely be the design from this report.

Firstly, the pipeline will be modified. The Decode stage will be split in two. This is to increase the overall speed of the system by removing a major bottleneck from the current system. The system control modules will be moved to a new stage that will be called Pre-Execute. This change will also allow for easy expansions, such as I/O, in revisions later down the line, since modules for system tasks will have their own stage. The second modification to the five stage pipeline will be adding a second clock stage for the Fetch. The Fetch accesses memory every clock cycle and therefore will miss often, so the added clock will allow the cache to sort out most misses without disrupting execution.

The second revision will be to expand on the BranchCPU. In its current state, the master core has no way of knowing if a core is currently working or not and if cores need further input. This limitation could be solved using a unified memory with the current configuration, by having cores write to specific memory addresses with state codes. These codes could then be read by the master so that it could have insight into what the sub-cores are doing. The new version would implement this into hardware as well as using unified memory. As an example, core-1 can request data from core-0. When core-0 sees that core-1 needs data via the new module, core-0 can look into a per-specified memory location for information as to what kind of data is needed. From there, core-0 can resolve the query by itself or command another sub-core to resolve the query. The advantage of using the unified memory to handle the specifics of data requests is that it simplifies the hardware; however, constantly checking memory addresses is very costly. This revision will keep hardware simple as well as resolve the main issue of the current design, which is constantly checking the main memory for status values.

The MOD, MUL, and DIV instructions would also be moved out of the ALU and onto a new module that would perform these special operations. These operations are very slow and will cause a lot of delay if left in the main pipeline, so the decision to move them to a separate execution line will alleviate this issue. The operations would write to special registers, in order to avoid over complicating the register block. Creating a new, separate module for these operations will also allow for the complete removal of the functions if the fabricator wanted a smaller, cheaper design.

Two thread hardware multi-threading is the final revision that would be made. The groundwork is already there with the Runtime interface. The multi-threading system however would be programmable by the kernel. Meaning that the kernel could choose to designate whether a sub-core would use multi-threading in the traditional sense, switch between the two threads after a kernel specified number of clock cycles, or use the classic Runtime environment. The benefit, over the current system, being efficiency through automation. There could be wasted cycles between when a thread terminates and the master-core realizes to switch it out. Setting the cores to run this system automatically increases performance. Using the classic system gives the kernel the ability to have sleeping processes that can be activated at a moments notice if need be.

Index of Modules

Adder – Increments PC value by 4.

ALU – Performs the desired arithmetic operation on its two operands. Throws an exception signal when overflow is encountered.

BJComp – Handles Branches and Jumps. Outputs the destination address of a branch or jump. If a branch is not taken, the output will be Input_PC+4 and fed to the PC.

BranchCPU – This is the controller for the sub-cores. At the moment it is hardcoded for only one sub-core, so modification will have to be done user side to add more core functionality. The max sub-cores allowable by the instruction set is 32.

ClockWait – Outputs the clock for a core. The input is the global system clock. The sleep time is calculated using the formula *count=value*(divider+1)+1*, where count is decreased every negative edge of the system clock.

Control – Determines the control bits for all modules depending on the input instruction. The total code string for core0 is

5'b BJComp Opcode	1'b EXIT	1'b Load Address	1'b BCPU	1'b Ex Op	1'b SLEEP	1'b EWrite	1'b CWrite	1'b Bypass Enable	1'b IMM Enable	1'b Reg Dst	6'b ALU Opcode	1'b MWrite	1'b MemEN	1'b R/W'	1'b WWrite	1'b MemToReg

The string for core1-32 is

5'b BJComp Opcode	1'b EXIT	1'b Load Address	1'b CPUSw	1'b BCPU	1'b ExOp	1'b SLEEP	2'b EWrite	2'b SWrite	1'b Bypass Enable	1'b IMM Enable	1'b Reg Dst	6'b ALU Opcode	1'b MWrite	1'b MemEN	1'b R/W'	1'b WWrite	1'b MemToReg

DReg – Holds the information from the Decode stage.

EReg – Holds the information from the Execute stage.

FReg – Holds the information from the Fetch stage.

MReg – Holds the information from the Memory stage.

PC – Holds the current program counter.

RAM – DDR write-first RAM.

RegisterBlock – For master-core, contains 32 registers. Receives four signals.

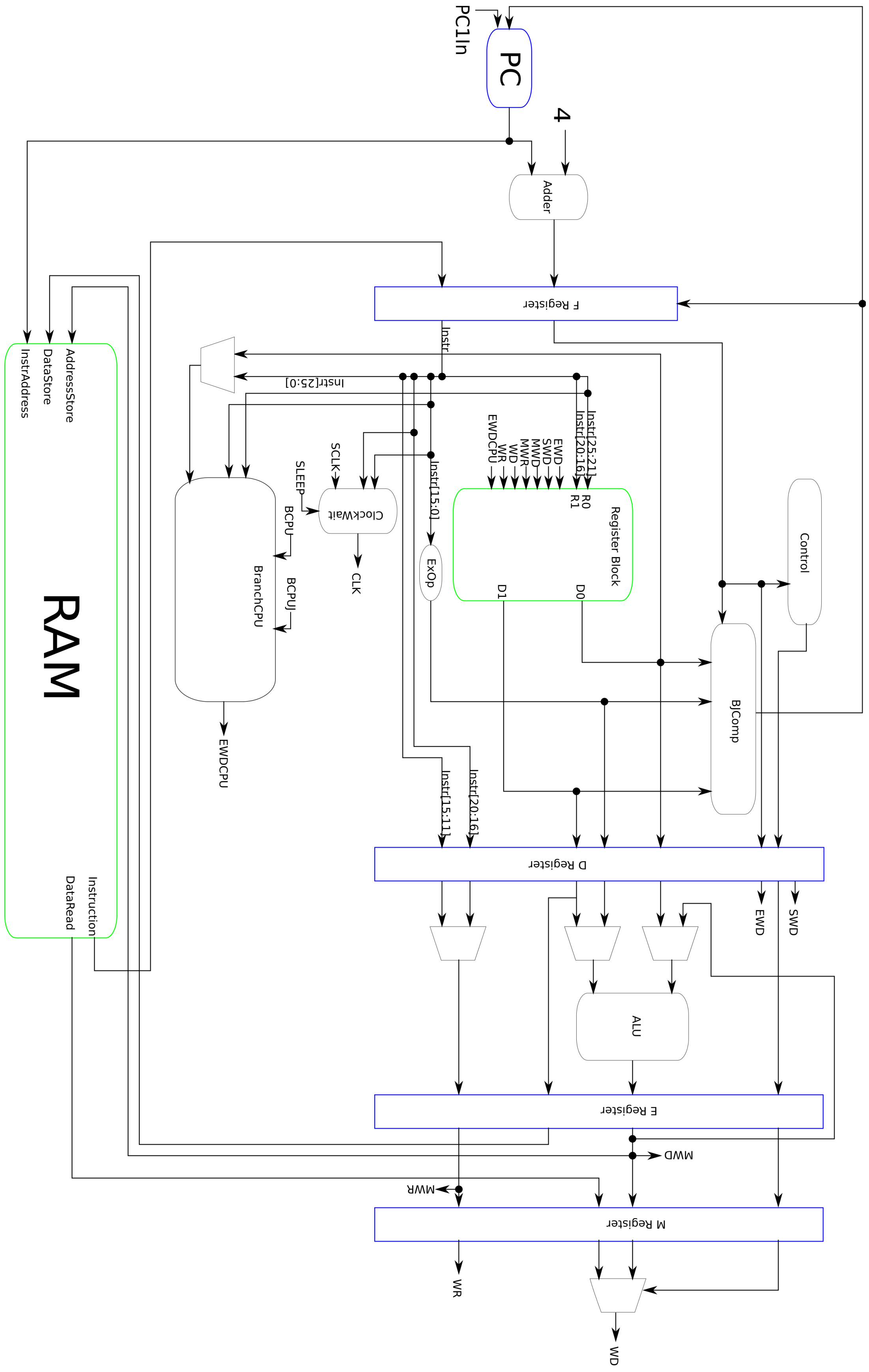
EWrite – Signal for link address in Execute stage. Saves to register 31.
CWrite – Signal for link address from sub-core in Execute stage. Saves to register 30.
MWrite – Signal to save output from ALU. Saves to registers 1-29.
WWrite – Signal to save read from RAM. Saves to registers 1-29.

For sub-cores, contains 64 registers. Usage of which discussed in the *Architecture* section.
[1:0]EWrite – LSB is write enable, MSB is which runtime register set.
[1:0]SWrite – for special link address. Saves to register 1. LSB is write enable, MSB is which runtime register set.
MWrite – Signal to save output from ALU. Saves to registers 1-29 for current runtime.
WWrite – Signal to save read from RAM. Saves to registers 1-29 for current runtime.

Outputs two registers' data on the positive edge of the clock. Saves data on the negative edge of the clock. The RegisterBlock is write-first.

SignExtender, aka ExOp – Extends a 16bit value to 32bits. If ExOp=1, the 16bit immediate will be sign extended. If ExOp=0, the 16bit immediate will not be sign extended.

Figure 1.1


PC1In
PC
4
Adder
F Register
Instr
Instr[25:0]
Instr[25:21]
Instr[20:16]
EWD
SWD
MWD
MWR
WD
WR
EWDCPU
R0
R1
Register Block
D0
D1
Control
BJComp
SCLK
SLEEP
ClockWait
CLK
BCPU
BCPUJ
BranchCPU
EWDCPU
Instr[15:0]
ExOp
Instr[15:11]
Instr[20:16]
D Register
EWD
SWD
ALU
E Register
MWD
MWR
M Register
WR
WD
RAM
AddressStore
DataStore
InstrAddress
Instruction
DataRead