Patrick Coleman

Computer Architecture 6pm

03/24/2022

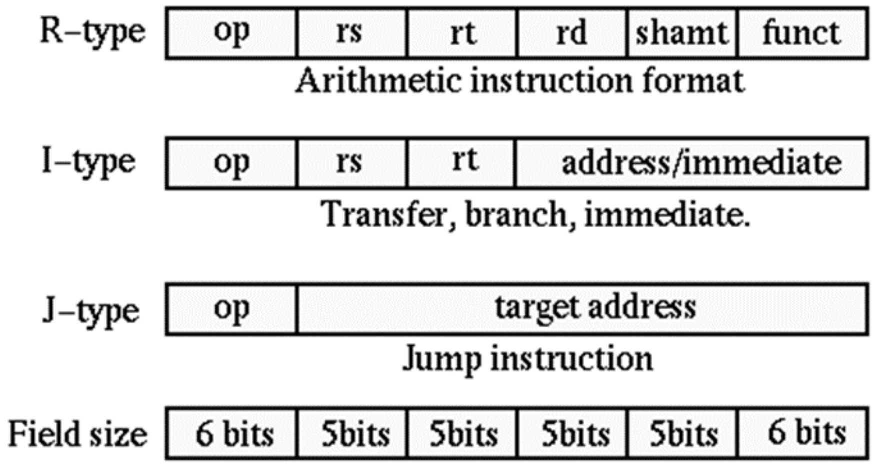
My Proposed Microprocessor

Abstract

I have split my proposed processor design into multiple sections addressing each aspect of the design. I envision a processor that can handle large amounts of data, while also being able to be used to run a general-purpose machine. The initial target audience for the processor would be data scientists.

My Proposed Language

I modeled my proposed language after the MIPS architecture since that's what I'm the most familiar with. The traditional MIPS Language is structured likened to the chart included below:



R–type	op	rs	rt	rd	shamt	funct
	Arithmetic instruction format					
I–type	op	rs	rt	address/immediate		
	Transfer, branch, immediate.					
J–type	op	target address				
	Jump instruction					
Field size	6 bits	5bits	5bits	5bits	5bits	6 bits

My processor will utilize these instruction types with the addition of my own proposed s-type instruction. This will make for 4-supported instruction types with the S-type formatted as shown below.

S-Type	op	rs	address/immediate	func

The ‘Field Size’ that my processor will accept is also different:

Field size	6 bits	12 bits	12 bits	12 bits	6 bits	16 bits

The field size chart shows the size and format of each instruction as the only change (between my proposed instruction format and the ones shown in the original MIPS chart) is the number of bits allotted to each section.

I chose to give the processor more registers so that more data could be stored at a given time and to be able to handle vectors/blocks of data.

Some of the proposed instructions my proposed processor should be able to handle are:

“sub.4D” *:: subtracts the data within two vectors*

“addi” :: adds an immediate with a register into a different register

“subr” :: subtracts the values within two registers into a third register

“subi” :: subtracts an immediate from a given register’s value and outputs into a different register

“spdr” :: prints to console a value from a given register as a double

“spfr” :: prints to console a value from a given register as a float

“sw” :: stores a value from a given register into 1 of the 8 memory addresses (as denoted by an int)

“spii” :: prints to console an immediate as an integer

“spdi” :: prints to console an immediate as a double

“spfi” ” :: prints to console an immediate as a float

“ld” :: retrieves a value from the given memory address into a given register

“maxr” :: does a bit comparison between 2 registers for the larger value and stores it into a separate one

“maxi” :: compares a register’s value & a given immediate for the larger number and stores in rs register

“minr” :: does a bit-comparison between 2 registers for the smaller value and stores it into a separate 1

"mini" :: compares a register's value & a given immediate for the smaller number; stores in rs register

Regarding access to memory, load and store instructions will be the only instructions allowed to access the memory.

S-Type commands include:

S-type commands are special in that they can take one argument of either the register-type or immediate type. The opcode details which field of the binary code to read to get the instruction and parameters (it is like R-types in this way). S-type instructions are complex instructions as I originally created them to run corresponding multi-line MIPS syscall instructions.

I chose to go with a fixed instruction size (64 bits) because I believe that the predictability of incoming bytes is an advantage. Additionally, with the adage of the S-Type command, which brings about a change to the instruction format, I believe that enough bits are allotted to support additional (possibly more complex, data-process oriented) instructions. The increased bits to the "funct" field not only adds more space for instructions, but also bits for memory addresses to store data when used in non-RType instruction formats.

For syntax:

- I separated from the MIPS language by using "-" to denote registers. This choice was primarily a result of the structures of keyboards today. "-" is simply faster to type than to hold "shift" and press "4" to get the "$" character every time I wanted to reference a register.
- Another separation was the choice to remove commas. Instructions could be parsed on whitespace instead of commas.

The instructions with the applied syntax looks like so:

```
addr -a0 -a1 -a2

addi -a0 -a0 100

sub.4D -a1 -a1 -a2

lw.D -s0 a

spir -a0

sw -a0 3

spii 10

lw -a3 0(-s0)

maxr -a0 -a0 -a1
```

```
maxi -a0 -a3 100
minr -s0 -a0 -a1
mini -s1 -s0 100
```

Registers:

12 bits for the registers means that I'm using approximately 4096 registers. I need about this many registers to organize large amounts of data into vectors. The way I would split it up would be:

Register Number	Register Name	Usage
1-64	-zero thru zero63	Zero value
65-86	-asm0 thru -asm31	For the assembler
87-102	-a0 thru -a15	The first 16 parameters passed to a procedure. Not preserved across procedure calls
103-1102	-s0 thru -s999	Saved values. Preserved across procedure calls
1103-2102	-sfp0 thru -sfp999	Saved floating point values. Preserved across procedure calls
2103-2602	-t0 thru -t499	Temporary registers. Not preserved across procedure calls
2603-3102	-tf0 thru -tf499	Temporary floating-point registers. Not preserved across procedure calls
3103-3110	-fpp0 thru -fpp7	The first eight floating point parameters. Not preserved across procedure calls
3111-3360	-fp0 thru -fp249	Floating point procedure results
3361-3610	-v0 thru -v249	Expression evaluation and procedure return results
3611-3674	-k0 thru -k63	Reserved for kernel usage
3675	-gp	Global pointer
3676-3803	-sp0 thru sp127	Stack pointer

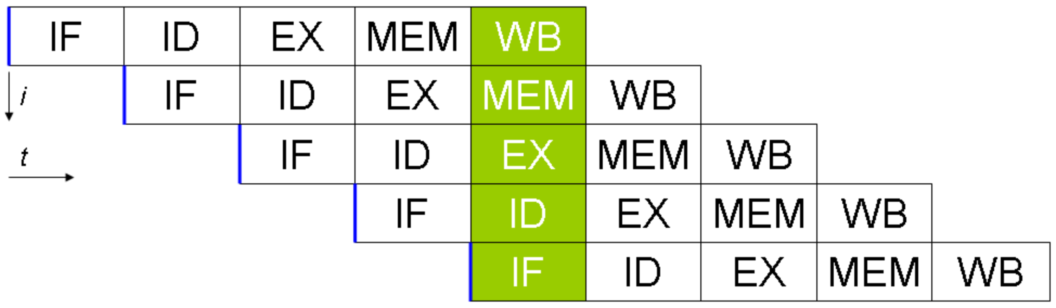
…	…	…
3968-4095	-ra0 thru -ra127	Return address

Figure Note:

> " 'Not preserved across procedure calls' means that the register may change value if a procedure is called. If some value is stored in that register before the procedure is called, then you may not make the assumption that the same value will be in the register at the return from the procedure.
>
> 'Preserved across procedure calls' means that the value stored in the register will not be changed by a procedure. It is safe to assume that the value in the register after the procedure call is the same as the value in the register before the call." (http://www.cs.iit.edu/~virgil/cs402/Labs/Lab4.pdf)

My goal with this register layout was to allow more flexibility to the various components of the system. Most parts of the system received more registers to work with to allow space for future innovation, and thread exclusivity.

Pipelining:



My processor would utilize the current 5-stage pipeline convention (for the general purpose aspect of the system) while leveraging non-unified memory and a forwarding unit. My reasoning for this decision is because I cannot think of any modifications to the pipeline structure that would be an improvement on this configuration.

The 5 conventional stages are:

> Instruction Fetch Cycle (IF)

Fetches the instruction from main memory and places it into registers.

The processor knows the bits of the instructions.

Instruction decode/register fetch cycle (ID)

The processor performs an equality test to determine if the instruction relates to branching and will compute the branch target address if it does.

As the processor knows what operation it needs to perform, it fetches the data from registers in preparation.

Execution/Effective address cycle (EX)

The ALU operates on the operands and performs either a memory reference, Register-to-Register instruction, or Register-Immediate ALU instruction.

Memory Access cycle (MEM)

A cycle for instructions that access the memory. The memory gets read and wrote to in this stage.

Write-back cycle (WB)

The processor writes data to registers.

The hazards that are inherited from the 5-stage pipeline are:

Structural hazards

In this context, it would be a conflict of resources, or more specifically the memory, as both the instruction fetch cycle and the memory access cycle must access the computer's memory concurrently.

Control hazards

The possibility of branching and changing the order of instructions to be ran through the pipeline.

Data Hazards

A conflict of data consisting of data not being ready to be read by one instruction as it is still being operated on by another instruction.

Aside from stalling, I plan to overcome these hazards by utilizing:

A 2-bit branch predictor to make an educated guess on control hazards.

2 bits were chosen in order to keep the branch prediction in the cache as opposed to having n-bits (which would slow down the processor by having to do branch prediction in main memory).

A forwarding/chaining unit to shorten pipelining throughput by forwarding data between cycles.

A non-unified memory would remove the potential for a structural hazard in the portion of the system sectioned out for normal operations. Additionally, memory ports would be added to other sections of the memory which is intended to store vectors.

Cache ~~Money~~ Memory:

My cache will be 2-way associative with the conventional 25-bit tag, 9-bit address, and 6-bit block offset for each set of blocks. I went with 2-way associative because it has improved search time over fully mapped and better hit rates than direct mapped.

I have chosen non-unified memory for the first two caches to treat both instructions and data equally. The separation would limit pipelining hazards and increase average instruction time processing by the CPU for normal operations. This structure will allow for frequently use data to remain as close to the CPU as possible without the possibility of being replaced, while allowing for instructions to be switched in and out as needed. This is especially helpful as it is common to perform multiple instructions on a group of data.

I chose a smaller size for the first level cache as a means of having the cache close to the processor. The third level cache will be much larger to handle large amounts of data and will be purely for data (no instructions as it is unnecessary). Because I am lacking in notion of decent caches sizes, I would say, if I had to give a number, 32KB for the 1st level cache, 512KB for the 2nd level cache, and 1GB for the third level cache.

The algorithm that my processor will use for cache block replacement will be LRU. I chose this algorithm as it made the most sense when considering the principle of temporal locality which states "If an item is reference, it will tend to be reference again soon." Inversely, if an item is least recently used, it is very likely that it will remain least recently used compared to the other items in a collection.

Regarding cache writing, the strategies my cache will use will be write-through for hits and write-allocate for misses.

> Write-through is chosen because it prioritizes consistency over speed. For applications that process large amounts of data, between accuracy, space and speed, speed is the least important factor. Inaccurate data defeats the purpose of the system and space is required to store the density of the data.
>
> For similar reason write-allocate was chosen: by giving up the speed boost acquired from not writing the data in the cache, the processor performs fewer misses based on the principles of temporal locality and special locality.

Multithreading:

As I intend for my processor to be able to compute large amounts of data, it will be built to use SIMT (single instruction multiple threads) architecture. Memory blocks will have ports with thread locks on them. For vector parallelization, my processor will group instructions into conveys and execute across multiple lanes. Lastly, my processor will utilize threads to hide any memory latency issues from having such a high-capacity third-level cache.

Summary

Components:

My processor will contain:

-A section of ALUs containing multiple lanes for processing vector data in conveys

-A forwarding/chaining unit to pipeline instructions

-Sectioned memory to avoid structural hazards

-SIMT/SIMD hybrid architecture

-2-way associative cache with 3 levels that runs LRU algorithm for block replacement

Closing Statement:

My processor is designed to handle both large amount of data and normal operations expected from general purpose processor. My design can be summed up in the statement: “what if I combined a GPU with a CPU?”