ELEC4700 Report

Joshua Beverley (c3195884) & Radhika Feron (c3256870)

# Aim

The aim of this project is to design and implement a multi-core MIPS based processor on the Altera FPGA development board. The SSRAM on-board memory chips will be used as the main memory and storage of the final test results of the benchmarking algorithm. The memory blocks within the FPGA will be used for the L1 cache. Performance will be measured using the SPEC4700 benchmark based on the matrix multiplication of , where is a matrix of 32-bit unsigned numbers.

# Design

The processor is based upon the pipelined MIPS based processor created in ELEC3720. A schematic diagram of the processor can be found in the Appendix, along with the MIPS instructions that have been implemented. The main change to this processor schematic is within the memory pipeline stage, where the RAM and memory module are replaced by a cache and its cache controller. The cache controller has access to the SRAM controller outside of the processor. The SRAM controller provides the reading/writing stall signals, as well as any data relevant to the successful completion of a read or write request. This SRAM controller has direct control over the SRAM module, and chooses which core will have access to the SRAM to read or write to it. The final multiprocessor has 2 processors, each with a L1 cache with a maximum size of 2 kB.

## Memory hierarchy

The cache memory follows the standard memory hierarchy as shown below in Figure 1.

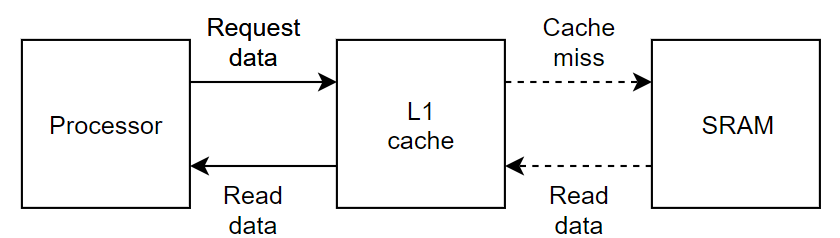


Figure : Memory Hierarchy

## Cache design

### Block size

The cache word size is 32 bits (4 bytes) and the cache block size is 1 word. The larger size will not improve the performance because all the reads have the same execution time. The SSRAM is running at the clock speed with a 16-bit data bus, thus it will take 2 cycles to retrieve a 32 bit value. Making the cache line wider will no necessary improve the time. For example, if the block size was 2 words, then 4 cycles will be needed to retrieve the block from SSRAM, which is equivalent to reading 2 words with a block size of 1.

### Replacement policy

The Least Recently Used (LRU) policy has been chosen due to the small cache memory size where it will quickly reach capacity and result in a capacity cache miss. It has also been shown that for smaller associativity, such as 2 or 4 way, the LRU policy performs best by increasing the probability of cache hits. To load a new block from main memory, an existing block in the cache has to be replaced. According to the LRU policy, the least recently used block from the cache is replaced and new data is loaded into it.

In the SystemVerilog implementation, a single LRU bit identifies if a specific block has been recently used. When a way in a particular set is accessed through a read or a write, the LRU bit is set. When all of the bits are set, all except the currently accessed bit are reset. When the block needs to be replaced, it chooses a block in the way whose LRU bit is off.

### Write through policy (adv and disadv)

Due to the nature of the benchmarking algorithm, the program will only be writing to main memory is once the calculations of an element in the B matrix is complete. Due this calculation, any intermediary result will be stored in the processor registers to avoid the penalty of writing to the main memory. Additionally, once this final value is written, it will never need to be read or written to again, and it is not useful to write the final value to the cache where they will evict elements in the A matrix required for further calculations. Thus, the write through policy is the most appropriate where on every write the data is written directly to main memory; especially since there is no additional delay to performing the write immediately.

### No-write allocate policy

In the no-write allocate policy, on a write miss the block is modified in the main memory and is not loaded into the cache. The write miss does not affect the cache which is desirable so any writes to the B matrix won’t affect the A matrix already stored in the cache. Additionally, if there were to be subsequent writes, they will automatically be written to memory anyway due to the chosen write through policy.

### 4-way Cache structure

A 4-way set associative cache structure was chosen because it best suits the tiling method used for the matrix multiplication with its matrix blocks. There are sets, thus 7 bits are needed to address each. From the 17-bit address, the top 10 bits are dedicated to the tag number. Table 1 shows the structure of the cache and the number of bits used in each cache column.

A multi-way set associative cache will have a lower miss rate than a direct mapped cache of the same size. It is slightly slower due to the extra comparators and MUX’s taking up more hardware resources, but this is counteracted by reducing the number of memory accesses required to complete the program.

Table : 4-way set associative cache structure

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Way 0 | | | | Way 1 | | | | Way 2 | | | | Way 3 | | | |
| Set | V | Tag | Block | LRU | V | Tag | Block | LRU | V | Tag | Block | LRU | V | Tag | Block | LRU |
| <7> | <1> | <10> | <32> | <1> | <1> | <10> | <32> | <1> | <1> | <10> | <32> | <1> | <1> | <10> | <32> | <1> |

### Flowchart for Read and Write operation

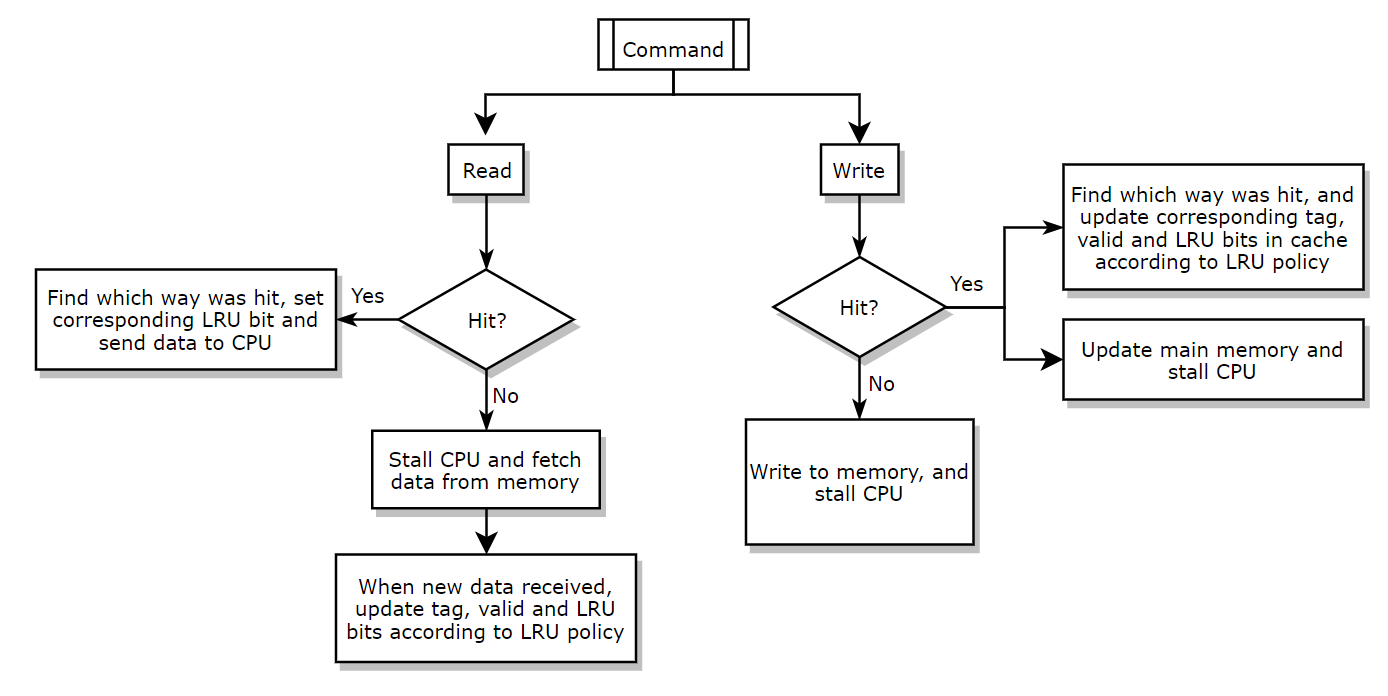


Figure : Flowchart for the cache read and write operations

## 2-core SRAM controller

The basic structure of the multi-core processor is shown in Figure 3. There are two cores that share the main memory SRAM which requires an arbiter (SRAM Controller) that delegates control over the SRAM using a round-robin method.

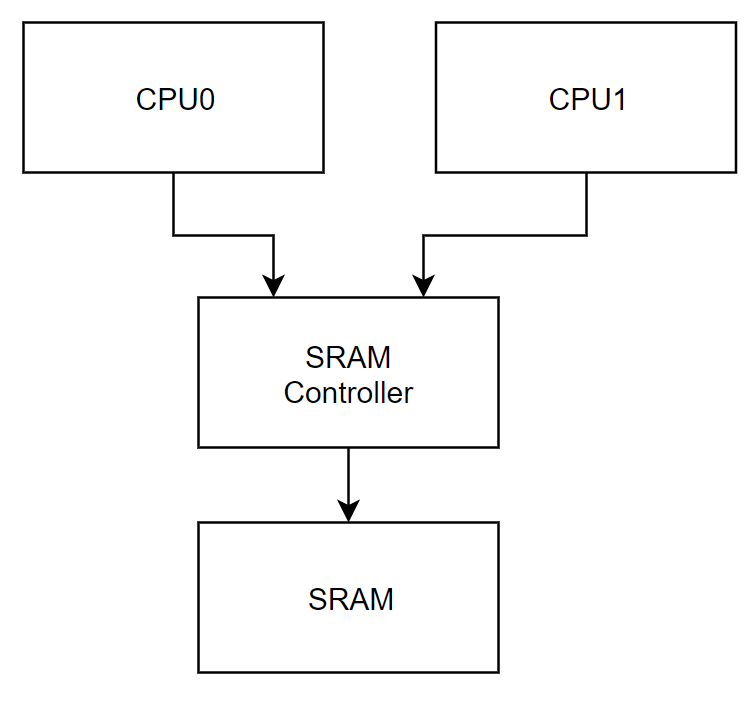


Figure : Multi-core processor block diagram with shared memory

Each core outputs their read/write requests together with the write data and SRAM address. The requests are only activated when the data was not found within their respective L1 caches. A single register will hold a bit indicated which of the two cores currently has access to the SRAM; if then CPU0 has control, if then CPU1 has control. On each clock cycle, the SRAM controller will check which CPU is requesting control and provide them access. If both CPU’s are requesting access, then whoever has used it previously will have to pass over control to the other CPU. This is shown in the flowchart in Figure 4.

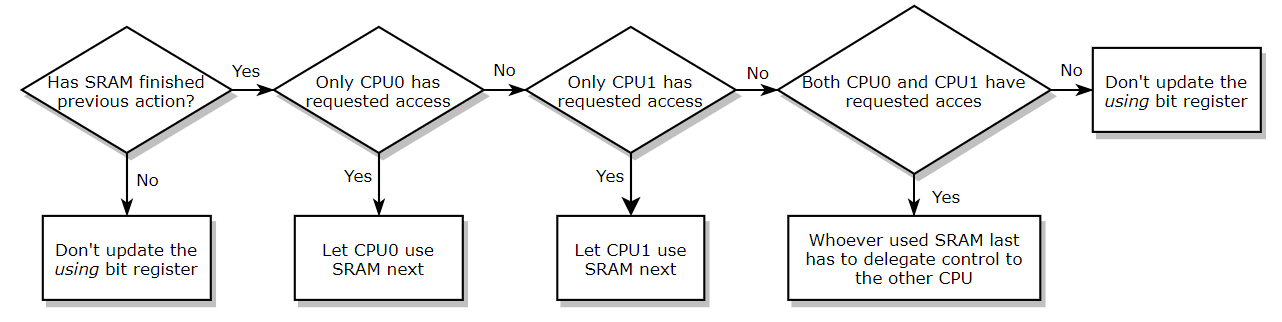


Figure : SRAM controller logic

## Matrix multiplication optimisation

Add the matrix multiplication stuff here

Do we include the link to the youtube video providing a visual explanation of the method??

<https://www.youtube.com/watch?v=aMvCEEBIBto>

# Innovation

1. Multi-way set associative cache with 4 ways instead of a direct mapped cache
2. Assembler to speed up manual assembly translation
3. Assembly code optimisation speeding up matrix computation time (tiling method)
4. Full MIPS instruction set

# Results

Maybe include some results from using dual core versus 1 core, or tiled method versus regular matrix multiplication

# Conclusion

# Appendix

## Schematic diagram of original pipelined MIPS-based processor

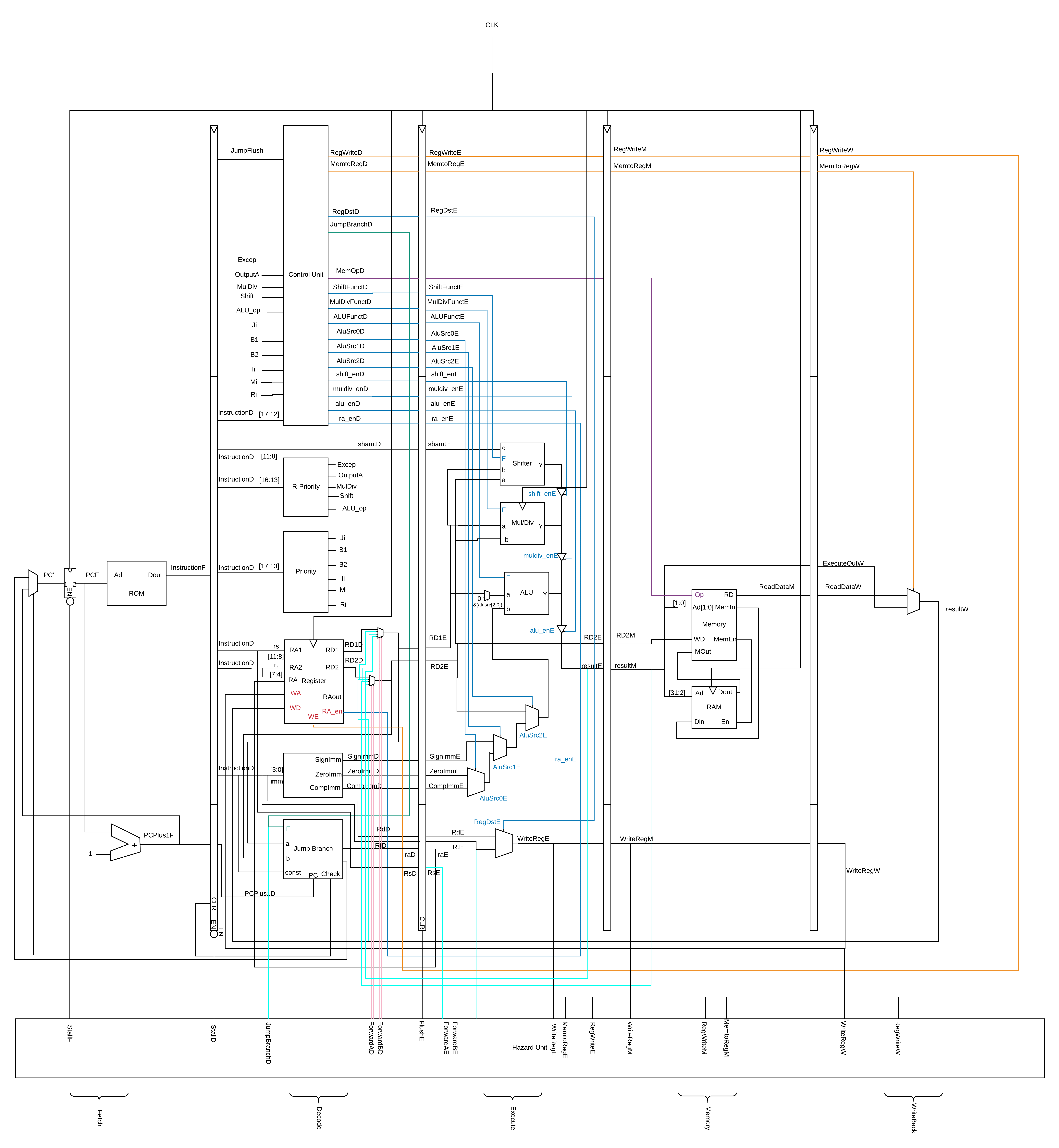


Figure : Block diagram of MIPS based processor from ELEC3720

## MIPS instructions implemented

**Shift**

sll $rd, $rt, $rs

srl $rd, $rt, $rs

sra $rd, $rt, $rs

sllv $rd, $rt, shamt

srlv $rd, $rt, shamt

srav $rd, $rt, shamt

**Multiply & divide**

mfhi $rd

mflo $rd

mult $rs, $rt

div $rs, $rt

mthi $rs

mtlo $rs

**Arithmetic & logic**

add $rd, $rs, $rt

addu $rd, $rs, $rt

sub $rd, $rs, $rt

subu $rd, $rs, $rt

and $rd, $rs, $rt

or $rd, $rs, $rt

xor $rd, $rs, $rt

nor $rd, $rs, $rt

slt $rd, $rs, $rt

sltu $rd, $rs, $rt

**Memory**

lw $rt, $rs, const

sw $rt, $rs, const

**Immediate**

addi $rt, $rs, SignImm

addiu $rt, $rs, ZeroImm

slti $rt, $rs, SignImm

sltiu $rt, $rs, ZeroImm

andi $rt, $rs, $ZeroImm

ori $rt, $rs, $ZeroImm

xori $rt, $rs, $ZeroImm

lui $rt, $rs, $ZeroImm

**Branch**

blez $rs, bta

bgtz $rs, bta

bgez $rs, bta

bltz $rs, bta

beq $rt, $rs, bta

bne $rt, $rs, bta

**Jump**

jr $rs

jalr $rs

j jta

jal jta