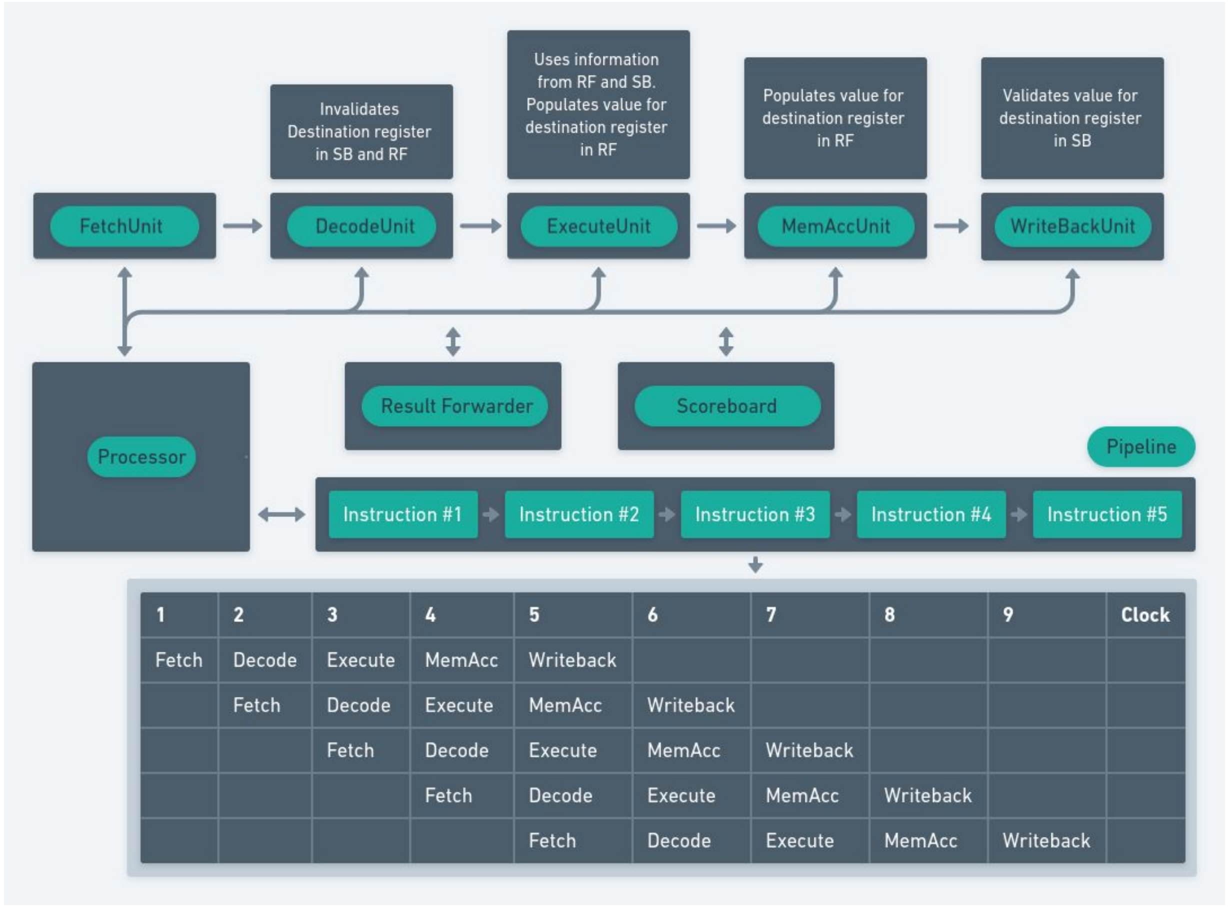
Advanced Computer Architecture Interim Submission

Rahat Muneeb (zl19450@bristol.ac.uk)

Architecture

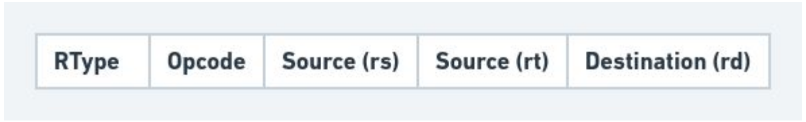

Invalidates Destination register in SB and RF
Uses information from RF and SB. Populates value for destination register in RF
Populates value for destination register in RF
Validates value for destination register in SB
FetchUnit
DecodeUnit
ExecuteUnit
MemAccUnit
WriteBackUnit
Processor
Result Forwarder
Scoreboard
Pipeline
Instruction #1
Instruction #2
Instruction #3
Instruction #4
Instruction #5

1	2	3	4	5	6	7	8	9	Clock
Fetch	Decode	Execute	MemAcc	Writeback					
	Fetch	Decode	Execute	MemAcc	Writeback				
		Fetch	Decode	Execute	MemAcc	Writeback			
			Fetch	Decode	Execute	MemAcc	Writeback		
				Fetch	Decode	Execute	MemAcc	Writeback	

Architectural Features

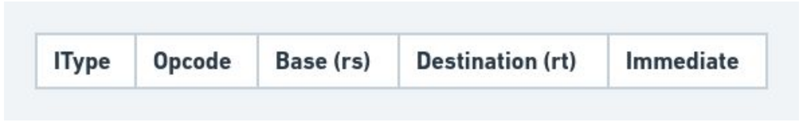
- 32 Registers
- 512 Instruction Memory Size
- 1024 Data Memory Size
- 5-Stage Scalar Pipeline.
  - Fetch
  - Decode
  - Execute
  - Memory Access
  - Write Back
- 5-way Scalar
- Result Forwarding in Pipeline
- Scoreboard validation
- Separate Processing Units for each Pipeline Stage
  - FetchUnit
  - DecodeUnit
  - ExecuteUnit
  - MemAccUnit
  - WriteBackUnit

ISA



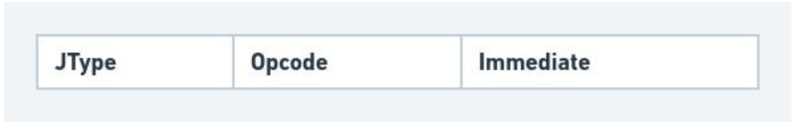
RType	Opcode	Source (rs)	Source (rt)	Destination (rd)

ADD , SUB , MULT, AND, OR , NOR , XOR , DIV, JR



IType	Opcode	Base (rs)	Destination (rt)	Immediate

ADDI, ANDI, ORI, LW, SW, BEQ, BL, BNE, BGTE, LSR, LSL



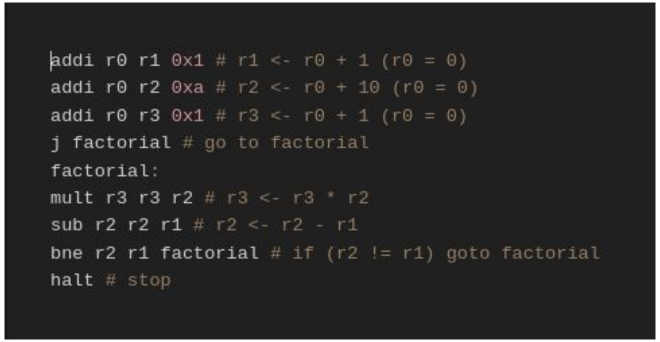
JType	Opcode	Immediate

J, JAL

- My ISA closely follows the MIPS32 Instruction Set
    - I have made some modifications to the Instruction format so as to ease the parsing process.
- Execution result of **RType** and **IType** instructions are put into result forwarding logic during **execution** stage except for **LW** instruction which populates the result forwarding logic at **Memory Access** Stage.
- At the **decode** stage the destination register of each instruction is invalidated in the scoreboard so as to avoid data hazards. This causes a stall in the pipeline incase an invalid register (that hasn't been written back to) is used as a source by another instruction.
- The result forwarding logic avoids a stall in the pipeline by populating the values of source registers (if present) prior to execution.

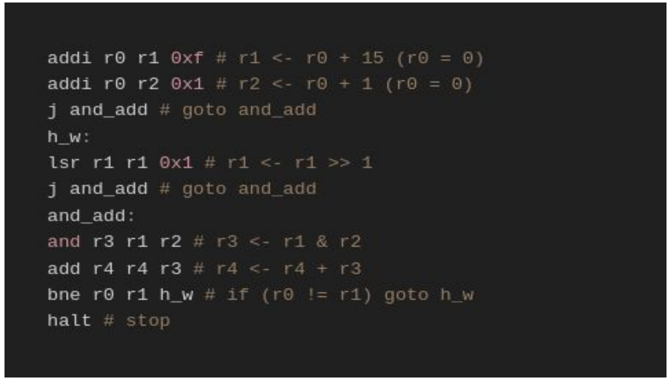
Testing

Factorial



```
addi r0 r1 0x1 # r1 <- r0 + 1 (r0 = 0)
addi r0 r2 0xa # r2 <- r0 + 10 (r0 = 0)
addi r0 r3 0x1 # r3 <- r0 + 1 (r0 = 0)
j factorial # go to factorial
factorial:
mult r3 r3 r2 # r3 <- r3 * r2
sub r2 r2 r1 # r2 <- r2 - r1
bne r2 r1 factorial # if (r2 != r1) goto factorial
halt # stop
```

Hamming weight



```
addi r0 r1 0xf # r1 <- r0 + 15 (r0 = 0)
addi r0 r2 0x1 # r2 <- r0 + 1 (r0 = 0)
j and_add # goto and_add
h_w:
lsr r1 r1 0x1 # r1 <- r1 >> 1
j and_add # goto and_add
and_add:
and r3 r1 r2 # r3 <- r1 & r2
add r4 r4 r3 # r4 <- r4 + r3
bne r0 r1 h_w # if (r0 != r1) goto h_w
halt # stop
```

- Factorial
  - Calculating factorial of 10
  - Running with Result Forwarding Logic
    - Program takes 72 clock cycles to complete
  - Running without Result Forwarding Logic
    - Program takes 81 clock cycles to complete
- Hamming Weight
  - Calculating factorial of 31
  - Running with Result Forwarding Logic
    - Program takes 80 clock cycles to complete
  - Running without Result Forwarding Logic
    - Program takes 86 clock cycles to complete

Next Steps

- Some of data structures have many of circular dependencies, I intend to make each of them as modular as possible. This will help in Super-scalar execution and benchmarking
- Implement a Debug mode such that I'm able to step through clock cycle and if need be through instructions in a clock cycle.
- Increase focus on unit testing using Catch2 (external testing library). Currently only parts of the processor are being unit tested, however I intend to move majority of testing to Catch2 tests.
- Fix bugs in JR and JAL implementation.
- Add data logic to load data into data memory, currently this is being done either by addi statements or manually.