AM04 MICROPROCESSOR

Description

The AM04 is a RISC microprocessor based on the MIPS I ISA Rev 3.2. It implements a subset of the full MIPS I ISA and is designed around a 32-bit architecture. The AM04 is a working synthesisable MIPS-compatible CPU, that uses a memory-mapped bus, allowing it to access system memory and various peripherals. The AM04 can be integrated on any FPGA or ASIC having been tested on Cyclone IV E.

AM04 Architecture

The AM04 defines a 32-bit word, a 16-bit half word and an 8-bit byte. The byte ordering is Little-Endian format.

31 24	23 16	15 8	7 0	Word Address
Byte-F	Byte-E	Byte-D	Byte-C	**C**
Byte-B	Byte-A	Byte-9	Byte-8	**8**
Byte-7	Byte-6	Byte-5	Byte-4	**4**
Byte-3	Byte-2	Byte-1	Byte-0	**0**

Figure 1 – Little Endian Byte Alignment

Processor Resources

The AM04 provides thirty-two 32-bit wide registers referred to as GPRs (General Purpose Registers). The AM04 also contains 3 special registers: the program counter (*PC*), the results of multiply / divide *Hi* and *Lo*. Furthermore, AM04 also utilises an ALU for arithmetic calculations, comparisons, and operations.
The ALU performs 25 different operations in order to be able to execute the 48 different instructions, which can be grouped into the following 3 types:

- **Manipulation operations –** These operands perform an operation on a value(s) and have an output to ALURes. In AM04 they include basic arithmetic operations (both signed and unsigned), bitwise operands & logical operations.
- **Conditional check operations** – These operands check conditions and have an output to *ALUCond*. In AM04 these operands include =, <, <=, >, >=, as well as a negative equality check. These operands are all signed.
- **Implementation Operations** – These operands are needed for the implementation of our ISA. They include *MTHI, MTLO, MFHI & MFLO* all of which allow for the movement of content between our General-Purpose Registers and the special registers *Hi* and *Lo*.

ADDIU	ADDU	AND	ANDI	BEQ	BGEZ
BGEZAL	BGTZ	BLEZ	BLTZ	BLTZAL	BNE
DIV	DIVU	J	JALR	JAL	JR
LB	LBU	LH	LHU	LUI	LW
LWL	LWR	MTHI	MTLO	MULT	MULTU
OR	ORI	SB	SH	SLL	SLLV
SLT	SLTI	SLTIU	SLTU	SRA	SRAV
SRL	SRLV	SUBU	SW	XOR	XORI

Figure 2 – AM04 Full Instruction Set

The program counter provides the address of the next instruction. The *Hi* register stores the result of the most significant 32 bits in a multiply instruction, and the remainder in a divide instruction. The *Lo* register stores the result of the least significant 32 bits in a multiply instruction, and the quotient in a divide instruction.

The Control unit is used to control the Datapath on a cycle-by-cycle level so that each instruction is executed correctly. The AM04 control unit takes in 2 inputs: a 32-bit instruction word and a 1-bit condition (*ALUCond*). 9 necessary control signals are then set by AM04 to execute a given instruction in a given cycle. They fall into 3 groups:

- **ALU** – *ALUOp, CtrlShamt*
- **Multiplexers** – *CtrlRegDst, ALUSrc, CtrlMemtoReg, CtrlPC*
- **Storage** – *CtrlRegWrite, CtrlMemWrite, CtrlSpecRegWriteEn*

For all instructions, AM04's control module uses: an *Opcode* (the most significant 6 bits of the instruction word); a function *funct* code (the least significant 6 bits of the instruction word); or *rt* (bits 20 to 16 of the instruction word); in order to identify which instruction must be executed.

The control module then combinatorially generates output signal *ALUOp* which specifies which operation the ALU should perform. If a shift instruction such as *sllv* (Shift left logical variable) is being executed than then control also outputs the required shift amount to the ALU *Ctrlshamt* so that the shift instruction can be executed correctly.

The control module simultaneously generates various storage control signals to ensure no data is overwritten in registers. This is required as on a cycle-by-cycle basis there will always be values present on data lines even if they are not in use, hence it is necessary to use control signals to determine whether they make changes to specific components i.e. the register files, Data Memory. For example, during any given instruction cycle there will always be a value on the *writedata* line hence AM04 uses the *CtrlRegWrite* signal to determine whether that value should be written to the register on *writereg*.

The multiplexer control signals similarly are needed in order control the flow of data and allow instruction to execute correctly. These signals are used by AM04 to decide what path is taken by data in any given cycle.

AM04 Overall Architecture

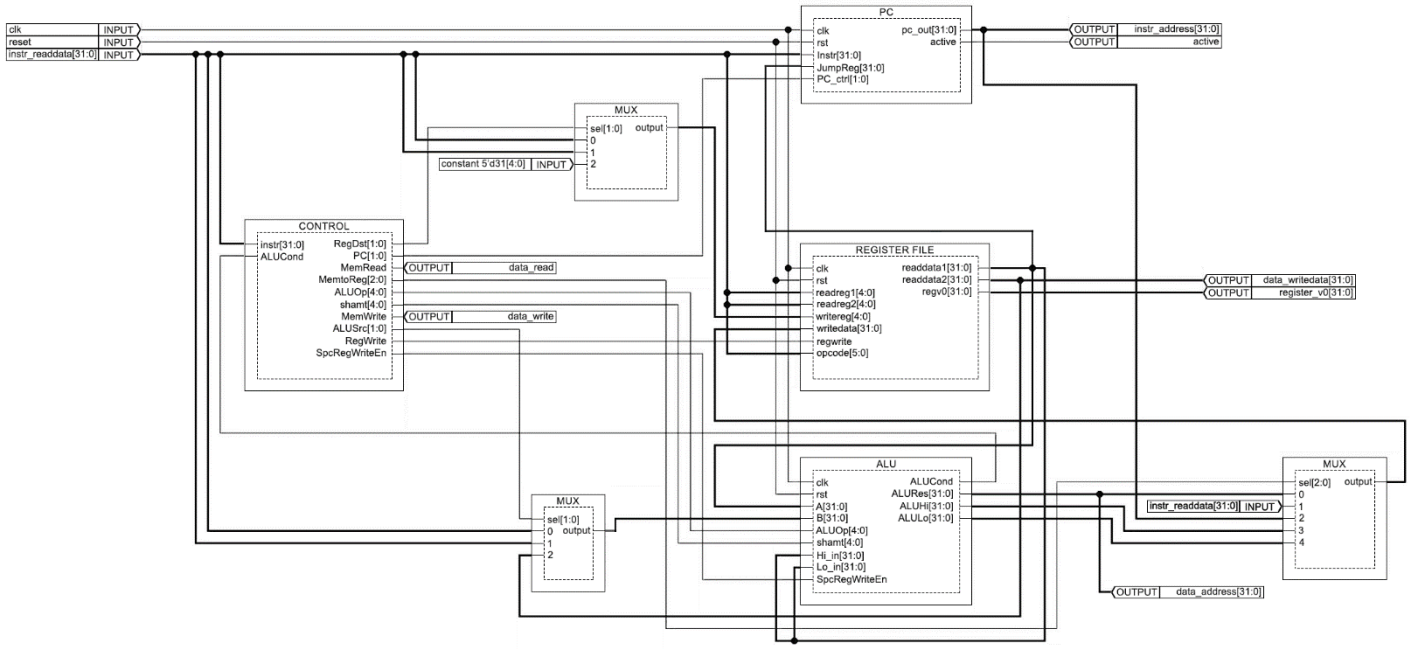

Figure 3 – Overall AM04 Architecture

Design Features/Decisions

An important design feature implemented in AM04 is the Avalon-compatible memory mapped interface [1]. This allows for instruction and data memory to be freely mapped within the address space, as well as for connections to memory mapped devices, e.g. GPUs or Network Interfaces. This allows a physical implementation to utilise a single memory module for both instruction and data memory. Without this feature, separate instruction and data memories would greatly increase logic and storage cell requirements. Consequently, this feature has significant upside both in memory storage and compatibility – Avalon is an industry specification allowing AM04 to be compatible with industry standard IP blocks.

This feature was implemented by taking the previous Harvard interface and wrapping it in a 'bus wrapper' module transparently adapting it into the bus-based interface. The wrapper stalls the Harvard submodule by holding the clock input high till the required instruction and data read lines are valid, allowing the submodule to continue operating as a single cycle CPU. The wrapper also handles conversion from word misaligned addresses to word aligned addresses with byte enables for partial writes or where only some bytes of the word are written to.

Partial stores (Store Byte, Store Halfword) are implemented using the byte enable lines of the Avalon Interface which allows bytes to be selectively written to without affecting other bytes within the word. Partial loads can use the byte enables however a connected Avalon device/slave may still return a full 32-bit word; instead, partial loads (Load Byte, Load Halfword, Load Word Left/Right) request the full word and are handled within the register file where the incoming value is shifted and masked, either with 0s or the MSB for sign extension.

[1] Avalon MM Interface Datasheet: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/manual/mnl_avalon_spec.pdf

Meeting ISA specifications, AM04 also supports the use of the branch delay slot in order to execute the instruction following a branch, reducing the occurrence of stalled cycles. The load delay restriction is part of the MIPS I ISA but as our CPU is single cycle rather than pipelined, all data is accessible by the next cycle.

Testing

The AM04 is tested to verify the capability and functionality of instructions listed in Figure 2 as defined by the MIPS I ISA. The AM04 is tested against a testbench that is driven by a wide set of testcases for each instruction that are specifically designed to examine its functionality and any edge cases present within the specification. The general approach taken when generating testcases was to take designed testcases in MIPS assembly and manually convert to hexadecimal instruction and data memory initialisation files. The testcases are organised per instruction where each instruction test set contains various testcases as initialisation files. There are two types of testcases: those which test the fundamental operation of an instruction, where it may not necessarily test edge cases, and the testcases specifically aimed at various specific cases, including edge cases. These two aspects of the test set enable effective detection of faulty implementation of an instruction. The presence of such edge testcases lead to more robust design of the AM04. A loose implementation could pass the first simple set of testcases of a few instructions but may fail on the more complex test or edge case sets depending on the degree of accuracy and detail in the implementation of the CPU. The complex testcases are constructed in a way that aims to break potentially defective implementations of the design. Examples of specific functionality checks done within the complex test cases are described below in Figure 4.

Types of instructions	Functionality check
Instructions with “and link”	PC+8 value should be always stored in the $ra ($31) register (except for JALR)
Instructions with signed/unsigned implementation	The interpretation of the most significant bit should be different
LB/LH/LWL/LWR	All the possible byte locations within a word should be implemented correctly
Shift Instructions by variable	The shift amount should be determined by the low-order five bits of the register value
Jump/Branch/Load Instruction	The CPU should be compatible with negative offsets
Instructions with Immediate	The immediate should be sign/zero extended according to the specification

Figure 4 – Examples of Specific Functionality Checks

Figure 5, seen on the following page, displays the simplified version of the test script where it iterates through the testcases within each instruction test set. An active flag from the CPU is set and remains high when the CPU is reset and is only set inactive when the CPU halts, which signifies the end of test for the respective testcase. Out of 32 GPRs the register $v0 ($2) is utilised to assess the correctness of the instruction operation by comparing the value in the register against the reference output for the testcase. Passing all testcases of an instruction test set verifies a valid implementation of the respective parts used for the instruction whereas a failure in some testcase(s) can provide a way of debugging a faulty design. Several auxiliary files are generated for efficient debugging.

As a part of the testbench, a memory block is designed according to the Avalon Memory-Mapped Interface (Avalon MM Interface)[1]. Both CPU and Testbench follow the “3.5.1. Typical Read and Write Transfers” timing diagram, with no pipeline or burst functionality but with the support for variable latency using the *waitrequest* signal which is asserted by the MM-Slave until it is ready for the next memory access.

The 4-byte width memory-mapped interface is subdivided into the instruction memory that is mapped to the address of the reset vector 0xBFC00000 and the data memory that is mapped to the address 0x1000. The CPU halts when the PC reads the address 0x0 from the memory and sets active low. The memory is initialised for each testcase before it gets compiled into an executable file.

[1] Avalon MM Interface Datasheet: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/manual/mnl_avalon_spec.pdf

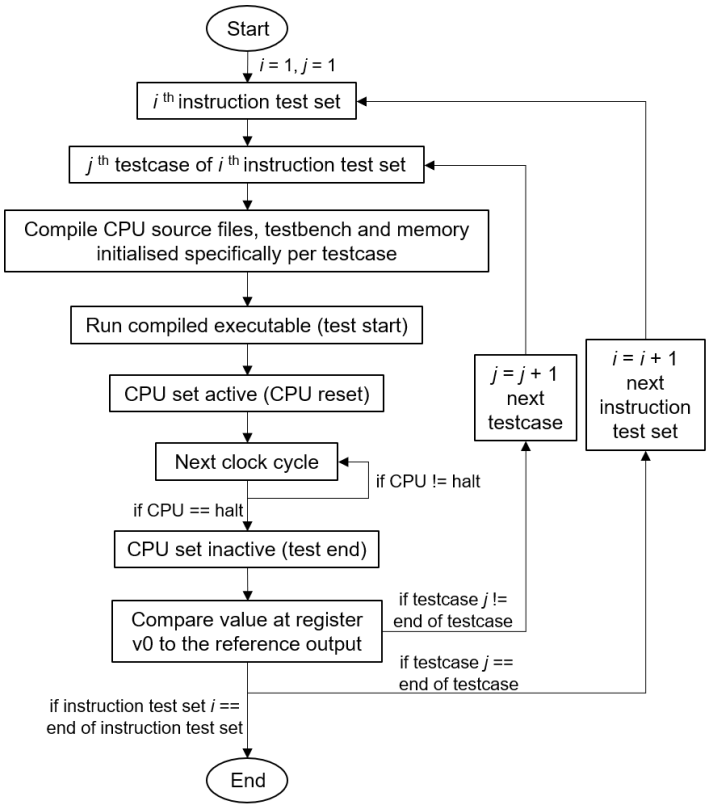

Figure 5 – Instruction Set Testing Flowchart

Area and Timing Analysis

The area (resource utilisation) and timing tests were done using Intel Quartus Prime Lite v19.1.0 with the FPGA family for the fitter set to “Cyclone IV E” and the device set to “Auto”. Figure 6 below shows the number of blocks, or area, taken up by each module within the Harvard and Bus CPU, where submodules count towards the upper module utilisation. For both designs, the fitter selected device “EP4CE10E22C6” and allocated 35 I/O pins.

Compilation Hierarchy Node	Combinational ALUTs
|AM04_CPU	227 (0)
|mips_cpu_harvard_tb:DUT|	227 (0)
|mips_cpu_harvard:cpuInst|	222 (25)
|mips_cpu_alu:alu|	64 (64)
|mips_cpu_control:control|	3 (3)
|mips_cpu_pc:pc|	80 (37)
|mips_cpu_cpc:cpc|	17 (17)
|mips_cpu_npc:npc|	26 (26)
|mips_cpu_regfile:regfile|	50 (50)
|mips_cpu_harvard_memory:ramInst|	5 (5)

Compilation Hierarchy Node	Combinational ALUTs
|AM04_CPU	3648 (0)
|mips_cpu_bus_tb:DUT|	3648 (0)
|mips_cpu_bus:cpuInst|	207 (6)
|mips_cpu_harva...ps_cpu_harvard|	201 (0)
|mips_cpu_alu:alu|	64 (64)
|mips_cpu_pc:pc|	105 (60)
|mips_cpu_cpc:cpc|	43 (43)
|mips_cpu_npc:npc|	2 (2)
|mips_cpu_regfile:regfile|	32 (32)
|mips_cpu_bus_memory:memInst|	3441 (3441)

Figure 6 – FPGA Resource Usage in Logic Blocks (Left: Harvard, Right: Bus)

For each design, a constraint file was provided to select the clock pin and a target clock rate of 40MHz. The estimated maximum clock rate is calculated at junction temperatures of 0°C and 85°C as shown in Figure 7.

Test Model	Harvard Design Fmax (MHz)	Bus Design Fmax (MHz)
Slow 1200mV 0°C	62.24	168.49
Slow 1200mV 85°C	56.04	153.00

Figure 7 – Maximum Clock Rate Analysis

The Harvard design achieves a ~60MHz clock rate however this is with a CPI (clocks per instruction) of 1, as this is a single cycle CPU, resulting in a MIPS (million instructions per second) of ~60. The Bus design achieves a much higher clock rate of ~160MHz, however CPI is increased to 2 for most instructions and 3 for instructions involving memory accesses as the same interface is used for fetching the instruction as well as interacting with the Avalon MM Interface. This gives a MIPS of ~53-80, similar to the Harvard design. The increased clock rate may be beneficial to other devices in the system which can retain the same latency in number of cycles (but improved due to shorter cycles).