MIPS32™ CPU (T501)

Data sheet last updated: 21/12/2020

Key Feature Overview

- Implements a powerful reduced 32-bit MIPS™ instruction set architecture on a robust IP core
  - Subset of MIPS1 32-bit little-endian instruction set
  - Includes 50 powerful instructions: (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, MFHI, MFLO MTHI, MTLO, MULT, MULTU, OR, ORI, SB, SH, SLL, SLLV, SLT, SLTI, SLTIU, SLTU, SRA, SRAV, SRL, SRLV, SUBU, SW, XOR, XORI)
  - 32 32-bit internal general purpose working registers (GPR) + 2 dedicated arithmetic registers HI and LO
  - Performant arithmetic capabilities served via on-chip ALU, single-cycle multiplier and division circuits
- Capable of high-speed data interface to interact with external memory and peripherals
  - Compatible with 32-bit word Intel® Avalon® Memory-Mapped Interface slaves
- Minimum FPGA Requirement: Cyclone IV E class FPGAs or equivalent.
  - Low area & power usage due to reusability of modules and design optimisations for efficiency.

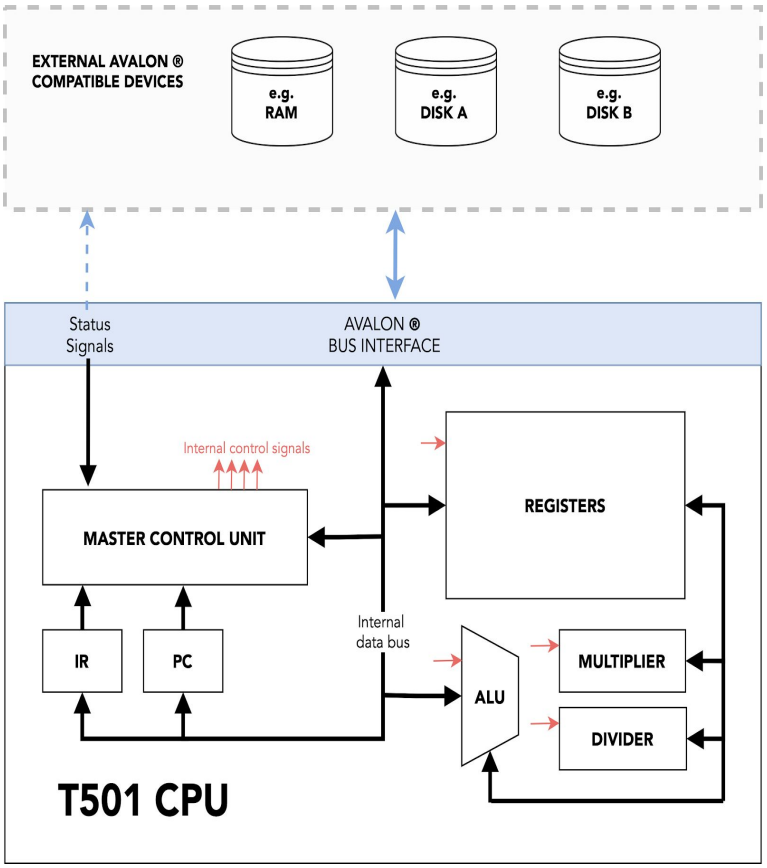
Table of contents

Electronic & Performance Characteristics

CHARACTERISTIC	VALUE	UNIT
Cycles per instruction	6.08	-
Max. clock frequency at 85°C on Slowest Silicon at 1200mV	106*	MHz
Max. clock frequency at 0°C on Slowest Case Silicon at 1200mV	116*	MHz
Max. clock frequency at 0°C on Fastest Silicon at 1200mV	185*	MHz
Total logic elements used	4863*	-
Core Static Thermal Power Dissipation @ 150MHz	45*	mW
Estimated Core Dynamic Thermal Power Dissipation @ 150MHz	100*	mW

** Figures obtained from simulation on an Intel®/Altera EP4CE10F17C6 using Intel® Quartus® Prime Lite*

T501 CPU Architecture Overview



Master Control Unit

The MCU provides control for all the blocks in the CPU, including the memory interface, registers (such as the program counter and instruction registers) and arithmetic modules that are not externally accessible. The PC and IR are internal registers which contain the memory address of the current instruction and the current instruction respectively. These help manage the state of the CPU.

ALU, Multiplier, Divider

The ALU handles adds, subs, shifts and all logical operators. *The multiplier* uses 16 FPGA DSP elements each, to return the 64-bit product of two 32-bit ints (unsigned & signed). *The divider* takes 32 cycles for non-zero inputs, to reduce logic use (by 500 elements) and permit higher clock rates. The infrequent use of division, makes this a worthwhile tradeoff for a higher speed for other instruction.

34 32-bit MIPS™ registers

The MIPS™ based register block contains 34 working registers, of which some are reserved for specific purposes according to convention:

PREFIX	FUNCTION
$zero	hard-wired to zero
$at	reserved for pseudo-instructions
$v0, $v1	function return values
$a0-$a3	function arguments, not preserved by subprograms
$t0-$t9	temporary data, not preserved by subprograms
$s0-$s7	saved registers, preserved by subprograms
$k0, $k1	strictly reserved for the kernel.
$gp,$sp, $fp	global area pointer, stack pointer, frame pointer
$ra	return address
HI / LO	store results from MULT/DIV instructions (Can be read and written to using MFHI/MTHI & MFLO/MTLO instrs)

Avalon® bus interface

The industry standard Intel® Avalon® interface is used to communicate and transfer data between compatible memory devices, such as RAM, and the CPU.

I/O	NAME	WIDTH	FUNCTION
in	waitrequest	1	Indicates the memory peripheral not being ready for transfer
in	readdata	32	Transferred data to CPU
out	address	32	Accessed address
out	write	1	Indicate start of write transfer
out	read	1	Indicates start of read transfer
out	writedata	32	Transferred data from CPU
out	byteenable	4	Indicates byte section that is being written to

Further information: Intel® Avalon® Reference

Design Decisions

5-cycle design

Our MIPS1 based CPU executes all instructions in the following 5 cycles:

Instruction Fetch	(IF)	fetches the next instruction from memory at the address specified by PC and stores it in the instruction register
Instruction Decode	(ID)	decodes the instruction and reads any operands needed to execute the instruction; Note: Usually MIPS architecture calculates PC_next here, including the result of a conditional branch, however in order to use less area on the chip, we have decided to move this functionality to the later EX and MA cycles
Execute	(EX)	actually "executes" the instruction. This is where all ALU operations take place, including the multiplier, divider and conditional branch operations. This is also where the base and offset additions, in load and store instructions, take place
Memory Access	(MA)	performs any memory access required. Load instructions will load data from memory and store instructions will store data into memory. Nothing happens in this stage for all other instructions
Write Back	(WB)	writes the result of the instruction into the destination register if applicable

Advantages

- ★ Reduced Complexity
- ★ Reduces Critical Path by adding registers in between stages, increasing clock speed.
- ★ Easily modifiable to enable Instruction Pipelining due to lack of interdependency on data and components within different cycles, which in turn if implemented with the necessary solutions to dealing with data hazards, control hazards, structural hazards and pipeline interlocking could ...
  - … increase instruction throughput
  - … reduce cycles per instruction
  - … potentially Increase clock speed

Branch and Jump

The MIPS™ delay slot is implemented using a 2-bit branch flag. When a jump/branch instruction is executed and the condition is true, the branch flag is set to 1. After the delay slot instruction is executed, the branch flag is set to 2 and this indicates to the CPU to jump to the address and reset the branch flag.

```
1 BRANCH = 0
2 if (JUMP OR BRANCH_COND == TRUE)
3     -> BRANCH = 1; PC_temp = specified address;
4 if (BRANCH == 1)
5     -> PC<=PC+4; BRANCH = BRANCH+1;
6 else if (BRANCH == 2)
7     -> PC<=PC_temp; BRANCH = 0;
8 else
9     -> BRANCH = 0; PC <= PC+4;
```

Pseudocode showing implementation of branch/jump instructions using branch flag

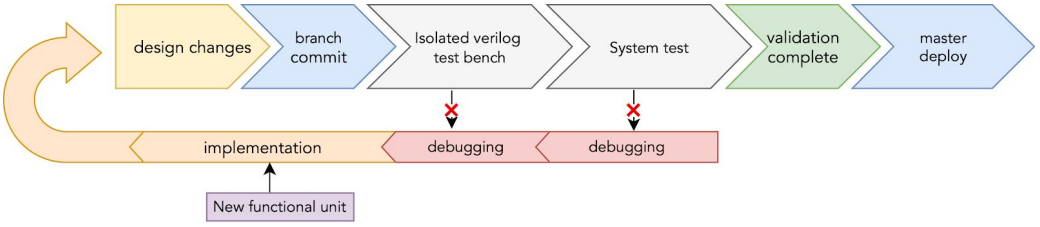
This approach causes repeated jump/branch instructions to only execute the last jump instruction and ignore the earlier jump instruction.

Design verification process

The T501 development team followed a strict design verification (testing) process, which was clearly defined for all areas of development. The design verification was tailored to the two main areas of design, which were *1: CPU functional unit development (ALU, Multiplier, Divider, Registers…)* and 2: *MIPS™ instruction development*.

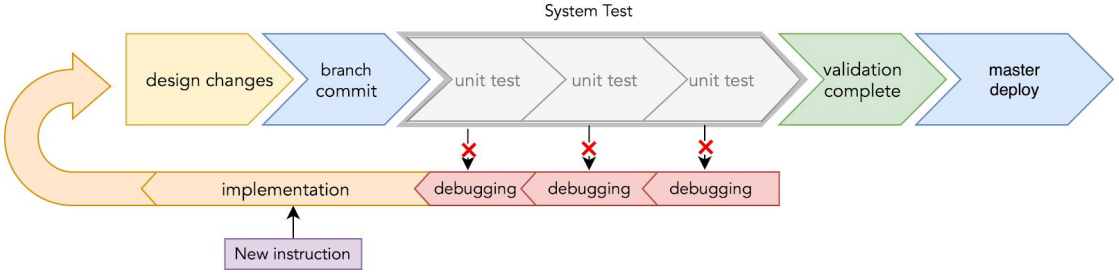
CPU functional unit development

When functional hardware modules of the CPU are tested, these are initially committed to a separate development branch. A module specific Verilog test bench is developed alongside, which covers specific edge cases as well as randomly generated test cases, with a reference output calculated using non-synthesizable Verilog functionality. If the functional unit passes the isolated test, it is then later tested in the whole system test, which is discussed in detail below. If both verifications pass, the feature is deployed to the master branch on our version control system (GIT + GitHub).



MIPS™ instruction development and testing

When the development of a new instruction is started, changes are made and initially only committed to a branch. The instructions on the branch are then tested alongside all other instructions in a system test. This allows for errors to be detected not only in the new instruction, but also exposes errors in other areas of the CPU and seemingly unrelated instructions. Once the system test has passed, the validation of the new development is complete and only then deployed to the master branch. This diagram summarises the described development and testing pipeline:



The system test itself is a strictly defined process: For every instruction, multiple assembly test cases alongside pre-calculated reference outputs are prepared. The assembly testcase is assembled by a compatible MIPS™ assembler, and the whole CPU HDL project is compiled with the given assembly test case in memory. The simulation is run, and the final output is compared to the reference. The test passes if all of these consecutive stages have succeeded and fails immediately if any of these stages fail. Using pre-calculated references reduces test time and allows for faster development. This instruction testing pipeline is repeated for all test cases of all instructions and in sum make up the system test.