O trabalho II compreenderá um arquitetura RISC implementada em FPGA. A CPU em questão será baseada nas CPUs MIPS. No entanto, não utilizará o mesmo instruction set. Este será modificado de acordo, para cada grupo.

OBRIGATORIAMENTE, UTILIZAR FPGA DA FAMÍLIA CYCLONE IV GX (qualquer uma da família que caiba o circuito).

OBRIGATORIAMENTE, SIMULAR EM GATE LEVEL.

Características Principais:

- A Word da arquitetura é definida em 32 bits;
- Todo o sistema é implementado em pipeline;
- Todas as instruções são formadas por 4 bytes;
- A memória de programa tem 1kWord alocados a partir de 0000h;
-A memória de dados tem 1kWord alocados a partir de ***Número do grupo * 500h*** (o módulo ***ADDR Decoding*** deverá fazer a decodificação de endereços apropriada para selecionar, apropriadamente, dados da memória de dados interna ou externa)
- A cada Reset, o Program Counter sempre aponta para o endereço 0 da memória de programa;
- Para essa versão, o *instruction set* não contemplará instruções de Branch/Jump;
- Para essa versão, o módulo *register file* conterá apenas 16 registros ($s0 a $s7 e $t0 a $t7);

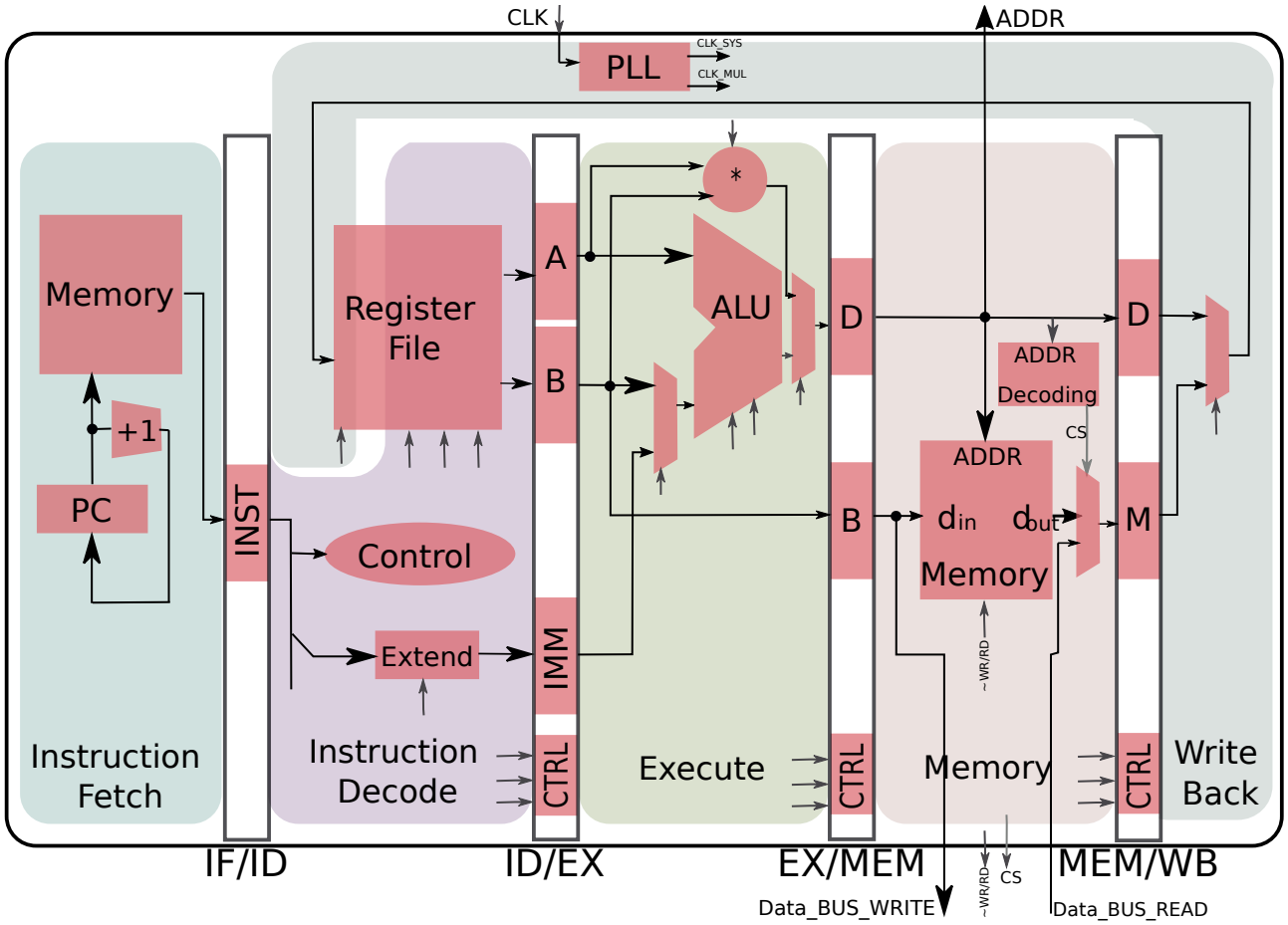

Fig. 1 – Arquitetura MIPS a ser Implementada

A figura acima mostra a arquitetura proposta, com todos os seus componentes, e seus 5 estágios de pipeline (*Instruction Fetch*, *Instruction Decode*, *Execute*, *Memory* e *Write Back*).

O projeto deverá ser feito utilizando hierarquia, ou seja, cada módulo deve ser feito e testado separadamente com *TestBench* apropriado. A adoção de abordagem ascendente ou descendente fica a cargo do grupo. Uma descrição estrutural deverá conectar todos os módulos, formando a MIPS_CPU (hierarquia top), que também deverá ser testada.

Instruction Set compreenderá apenas as instruções marcadas a seguir. Os 6 primeiros bits de cada instrução, que definem seu tipo, serão definidas pelo número do grupo.

rs	Primeiro registrador Fonte
rt	Segundo registrador Fonte para instruções tipo R Ou Registro destino para instruções tipo I
rd	Registrador destino para Instruções tipo R

Tab. 1 – Legenda

Instruction name	Mnemonic	Format	Encoding (10)			
Tamanho em Bits			**6**	**5**	**5**	**16**
Load Byte	LB	I	32	rs	rt	offset
Load Halfword	LH	I	33	rs	rt	offset
Load Word Left	LWL	I	34	rs	rt	offset
Load Word	LW	I	(Grupo+1)	rs	rt	offset
Load Byte Unsigned	LBU	I	36	rs	rt	offset
Load Halfword Unsigned	LHU	I	37	rs	rt	offset
Load Word Right	LWR	I	38	rs	rt	offset
Store Byte	SB	I	40	rs	rt	offset
Store Halfword	SH	I	41	rs	rt	offset
Store Word Left	SWL	I	42	rs	rt	offset
Store Word	SW	I	(Grupo+2)	rs	rt	offset
Store Word Right	SWR	I	46	rs	rt	offset

Instruction name	Mnemonic	Format	Encoding (10)					
Tamanho em Bits			**6**	**5**	**5**	**5**	**5**	**6**
Add	ADD	R	Grupo	rs	rt	rd	10	32
Add Unsigned	ADDU	R	0	rs	rt	rd	10	33
Subtract	SUB	R	Grupo	rs	rt	rd	10	34
Subtract Unsigned	SUBU	R	0	rs	rt	rd	10	35
Multiplication	MUL	R	Grupo	rs	rt	rd	10	50
And	AND	R	Grupo	rs	rt	rd	10	36
Or	OR	R	Grupo	rs	rt	rd	10	37
Exclusive Or	XOR	R	0	rs	rt	rd	10	38
Nor	NOR	R	0	rs	rt	rd	10	39
Set on Less Than	SLT	R	0	rs	rt	rd	10	42
Set on Less Than Unsigned	SLTU	R	0	rs	rt	rd	10	43
Add Immediate	ADDI	I	8	rs	rd	immediate		
Add Immediate Unsigned	ADDIU	I	9	$s	$d	immediate		
Set on Less Than Immediate	SLTI	I	10	$s	$d	immediate		
Set on Less Than Immediate Un	SLTIU	I	11	$s	$d	immediate		
And Immediate	ANDI	I	12	$s	$d	immediate		
Or Immediate	ORI	I	13	$s	$d	immediate		
Exclusive Or Immediate	XORI	I	14	$s	$d	immediate		

Tabela 2 – MIPS Instruction Set (ISA) Modificado

O Programa de TestBench da CPU deverá executar a seguinte expressão matemática, salvando o resultado na ***última posição da memória de dados interna***:

MemDados [última posição] ← (A*B) – (C+D),

onde: **A=2001$_{(10)}$, B=4001$_{(10)}$, C=5001$_{(10)}$** e **D=3001$_{(10)}$.**

Em *Portugol*, o código seria do tipo (exemplo):

1. Carrega A em R0;
2. Carrega B em R1;
3. Carrega C em R2;
4. Carrega D em R3;
5. R4 recebe A*b;
6. R5 recebe C+D;
7. R6 recebe [R4] – [R5]

Código 1 – Exemplo

Obs.: Traduza o portugol acima para *MIPS Assembly*, montando assim um programa binário (utilize como base as Tabela 2 e 3). Após gerar o código binário, gere a memória de programa com este código binário inicializado na mesma .

Inicialize a memória de programa com dois códigos:
- o primeiro representará exatamente a expressão, de forma a ser verificado o *pipeline hazzard (código 1 de exemplo)*;
- o segundo deverá conter algum artifício de forma a não se verificar o *pipeline hazzard* e a execução ocorrer normalmente (inserção de bolhas).
(Atente que, os dois códigos serão inicializados na memória de programa ao mesmo tempo, de foma subsequente).

Lembre-se: arquiteturas MIPS não fazem operações aritméticas/lógicas diretamente em memória. Os dados precisam primeiramente ser carregados nos registros com instruções Load.
Tabela 3 exemplifica a codificação Assembly MIPS
Assuma que a arquitetura seja *BigEndian*.

Obs.: A instrução de multiplicação operará em 16 bits, ou seja, os operandos correspondem aos 16 bits menos significativos do conteúdo de ***rs*** e ***rt***. Dessa forma, o resultado será de 32 bits, a ser armazenado em ***rd***.
Obrigatoriamente, o hardware responsável pela multiplicação será o abordado no laboratório 4, modificado para operandos de 16 bits. Ele deverá, OBRIGATORIAMENTE, funcionar com um clock diferente do clock do sistema. Mesmo com a adição do multiplicador, o sistema ainda deverá operar com *throughput* de 1 instrução/clk.
O clock do sistema (***systemClock***) e clock do multiplicador (***multiplierClock***) devem ser saídas do bloco de propriedade intelectual (IP) ALTPLL.

Avaliação (responda em forma de comentários no módulo TOP MIPS_CPU):
Após a implementação e verificação do correto funcionamento do circuito, responda (respostas dentro do módulo MIPS_CPU como comentários):
a) Qual a latência do sistema?
b) Qual o throughput do sistema?
c) Qual a máxima frequência operacional entregue pelo *Time Quest Timing Analizer* para o multiplicador e para o sistema? (Indique a FPGA utilizada)
d) Qual a máxima frequência de operação do sistema? (Indique a FPGA utilizada)
e) Com a arquitetura implementada, a expressão (A*B) – (C+D) é executada corretamente (se executada em sequência ininterrupta)? Por quê? O que pode ser feito para que a expressão seja calculada corretamente?
f) Analisando a sua implementação de dois domínios de clock diferentes, haverá problemas com metaestabilidade? Por que?
g) A aplicação de um multiplicador do tipo utilizado, no sistema MIPS sugerido, é eficiente em termos de velocidade? Por que?
h) Cite **modificações cabíveis na arquitetura do sistema** que tornaria o sistema mais rápido (frequência de operação maior). Para cada modificação sugerida, qual a nova latência e *throughput* do sistema?

O que entregar? Arquivo zip com a estrutura hierárquica do projeto como a seguir (delete todas as pastas *simulation* que encontrar na estrutura de diretórios. Caso contrário, o arquivo zip ficará muito grande e não será possível enviá-lo pelo SIGAA):
(arquivos qws, qpf e qsf são gerados automaticamente ao se criar o projeto)
TestBench mostrando ***clock, systemClock , multiplierClock, reset***, **demais saídas apresentadas na fig. 1, além do barramento interno do Write Back, rodadno os dois programas (com e sem pipeline hazzard), que deverão estar na memória ao mesmo tempo e serem executados sequencialmente.**

- → MIPS_CPU (pasta zipada)
  - → cpu.v (descrição estrutural ligando os módulos)
  - → TB.v (testbench da cpu)
  - → cpu.qws
  - → cpu.qpf
  - → cpu.qsf
  - → DataMemory
    - → datamemory.v
    - → datamemory_TB.v
    - → datamemory.qws
    - → datamemory.qpf
    - → datamemory.qsf
  - → InstructionMemory
    - → instructionmemory.v
    - → instructionmemory_TB.v
    - → instructionmemory.qws
    - → instructionmemory.qpf
    - → instructionmemory.qsf
  - → MUX
    - → mux.v
    - → mux_TB.v
    - → mux.qws
    - → mux.qpf
    - → mux.qsf
  - → PC
    - → pc.v
    - → pc_TB.v
    - → pc.qws
    - → pc.qpf
    - → pc.qsf
  - → ALU
    - → alu.v
    - → alu_TB.v
    - → alu.qws
    - → alu.qpf
    - → alu.qsf
  - → Multiplicador

- multiplicador.v
- multiplicador_TB.v
- multiplicador.qws
- multiplicador.qpf
- multiplicador.qsf

→ Control
- control.v
- control_TB.v
- control.qws
- control.qpf
- control.qsf

→ RegisterFile
- registerfile.v
- registerfile_TB.v
- registerfile.qws
- registerfile.qpf
- registerfile.qsf

→ Extend (estende o offset de 16 bits para 32 bits nas instruções lw e sw).
- extend.v
- extend_TB.v
- extend.qws
- extend.qpf
- extend.qsf

→ Register
- Register.v
- Register_TB.v
- Register.qws
- Register.qpf
- Register.qsf

→ ADDRDecoding
- ADDRDecoding.v
- ADDRDecoding_TB.v
- ADDRDecoding.qws
- ADDRDecoding.qpf
- ADDRDecoding.qsf

MIPS operands

Name	Example	Comments
32 registers	$s0, $s1, . . . , $t0, $t1, . . .	Fast locations for data. In MIPS, data must be in registers to perform arithmetic.
2^{30} memory words	Memory[0], Memory[4], . . . , Memory[4294967292]	Accessed only by data transfer instructions in MIPS. MIPS uses byte addresses, so sequential words differ by 4. Memory holds data structures, such as arrays, and spilled registers.

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add $s1,$s2,$s3	$s1 = $s2 + $s3	three operands; data in registers
	subtract	sub $s1,$s2,$s3	$s1 = $s2 – $s3	three operands; data in registers
Data transfer	load word	lw $s1,100($s2)	$s1 = Memory[$s2 + 100]	Data from memory to register
	store word	sw $s1,100($s2)	Memory[$s2 + 100] = $s1	Data from register to memory

FIGURE 3.4 MIPS architecture revealed through section 3.3. Highlighted portions show MIPS assembly langua structures introduced in section 3.3.

MIPS operands

Name	Example	Comments
32 registers	$s0, $s1, . . . , $s7 $t0, $t1, . . . , $t7	Fast locations for data. In MIPS, data must be in registers to perform arithmetic. Registers $s0–$s7 map to 16–23 and $t0–$t7 map to 8–15.
2^{30} memory words	Memory[0], Memory[4], . . . , Memory[4294967292]	Accessed only by data transfer instructions in MIPS. MIPS uses byte addresses, so sequential words differ by 4. Memory holds data structures, such as arrays, and spilled registers.

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add $s1,$s2,$s3	$s1 = $s2 + $s3	Three operands; data in registers
	subtract	sub $s1,$s2,$s3	$s1 = $s2 - $s3	Three operands; data in registers
Data transfer	load word	lw $s1,100($s2)	$s1 = Memory[$s2 + 100]	Data from memory to register
	store word	sw $s1,100($s2)	Memory[$s2 + 100] = $s1	Data from register to memory

MIPS machine language

Name	Format	Example						Comments
add	R	0	18	19	17	0	32	add $s1,$s2,$s3
sub	R	0	18	19	17	0	34	sub $s1,$s2,$s3
lw	I	35	18	17	100			lw $s1,100($s2)
sw	I	43	18	17	100			sw $s1,100($s2)
Field size		6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions 32 bits
R-format	R	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	I	op	rs	rt	address			Data transfer format

FIGURE 3.6 MIPS architecture revealed through section 3.4. Highlighted portions show MIPS machine language structures introduced in section 3.4. The two MIPS instruction formats so far are R and I. The first 16 bits are the same: both contain an *op* field, giving the base operation; an *rs* field, giving one of the sources; and the *rt* field, which specifies the other source operand, except for load word, where it specifies the destination register. R-format divides the last 16 bits into an *rd* field, specifying the destination register; *shamt* field, which is unused in Chapter 3 and hence always is 0; and the *funct* field, which specifies the specific operation of R-format instructions. I-format keeps the last 16 bits as a single *address* field.

Tabela 3 - Exemplos da codificação Assembly MIPS

Avaliação

1) Implementação da arquitetura geral = 50 pontos
2) Implementação da arquitetura geral + respostas corretas = 70 pontos
3) Implementação da arquitetura geral com o multiplicador = 80 pontos
4) Implementação da arquitetura geral com o multiplicador + respostas corretas = 100 pontos