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 **0500h**;

-A memória de dados tem 1kWord alocados a partir de ***Número do grupo * 300h*** (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 inicial da memória de programa;
- Para essa versão, o *instruction set* não contemplará instruções de Branch/Jump;

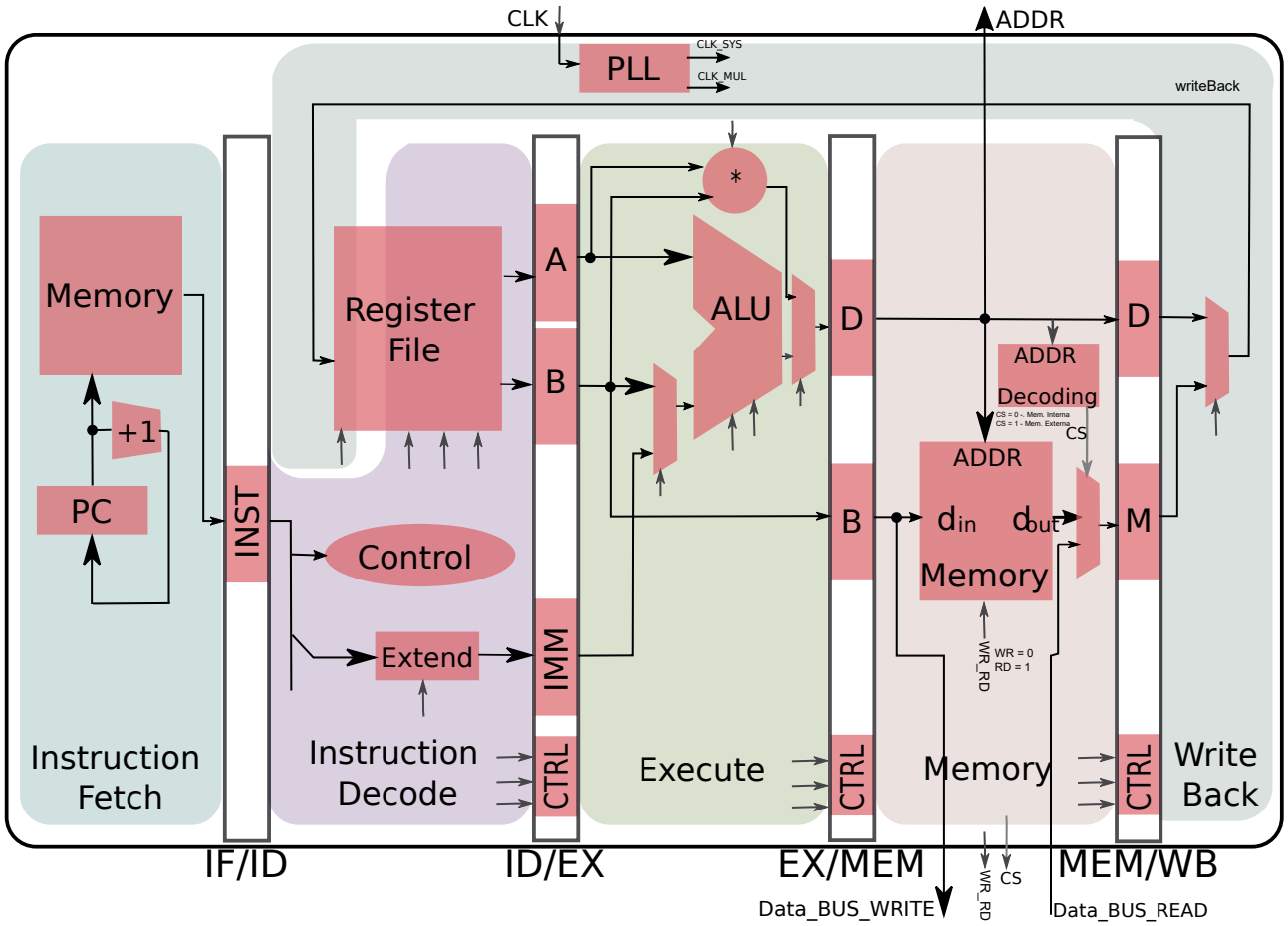

Fig. 1 – Arquitetura MIPS a ser Implementada

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]
8. MemDados [última posição] recebe [R6]

Código 1 – Exemplo

Obs.: Traduza o portugol acima para MIPS Assembly e, após, gere o código binário a ser salvo na memória de programa (utilize como base as Tabela 2 e 3). 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).

Ambos os códigos (com pipeline hazzard e sem pipeline hazzard deverão ser carregados na memória de programa ao mesmo tempo).

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 (***CLK_SYS***) e clock do multiplicador (***CLK_MUL***) 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 (Apenas os arquivos indicados. 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)
IMPORTANTE: Antes de enviar o arquivo compactado, extraia-o em outra pasta, resimule-o e garanta que o arquivo contenha a hierarquia funcional e pedida. NÃO IREI CORRIGIR TRABALHOS QUE NÃO ATENDAM O QUE ESTA PEDIDO.
TestBench mostrando saídas/entradas e sinais internos indicados por nomes na fig. 1 (*exatamente na forma como escrito na fig.1 – utilizar $init_signal_spy e (*keep=1*) para sinais internos*), rodando 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
- PLL
  - Manter todos os arquivos da pasta raiz do projeto deletando apenas as pastas.

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

Dicas:

1) Todos os módulos que contiverem *Reset* devem *resetar* da mesma forma (parece brincadeira mas, quando vocês dividem as tarefas e cada um faz um módulo, acaba cada um fazendo de um jeito). E lembre-se, o tipo de *reset* tem suas implicações. É possível fazer funcionar de qualquer forma sem que haja interferência na arquitetura proposta.

2) Todos os módulos devem operar com o mesmo tipo de clock. É claro que o multiplicador e sistema terão os seus respectivos. O que quero dizer é, todos devem operar na mesma borda. É possível fazer módulos operarem em bordas diferentes? Sim, mas a análise do *timing* no TimeQuest Timing Analyzer será diferente.

O único módulo que pode dificultar a implementação é o *Register file*. Se você estiver com problemas em fazer uma descrição apropriada, utilize uma BRAM funcionando como *true dual port* RAM. No entanto, a descrição vista na aula sobre memórias e bancos de registradores deveria ser o suficiente.

3) Tenha certeza que o programa gravado na BRAM que funciona como InstructionMemory contém o programa que gera pipeline hazzard e o que não gera.

4) Tenha certeza que seu sistema funciona no *testbench* rodado em GateLevel. Para isso, a estimativa de máxima frequência de operação deve estar correta e deve ser respeitada na definição do *clock* no TestBench.

5) Tenha certeza que todos os sinais monitorados no *testbench* são os pedidos no escopo do trabalho, inclusive respeitando o *case* dos caracteres.

FAQ

1 Tamanho do .zip maior que 10MB

1.a Por favor, siga exatamente as instruções do guia do trabalho. Há exatamente a estruturas das pastas, com os arquivos que devem estar nelas. Todo o restante deve ser deletado. Seguindo isso, vai caber com folga em 10Mb

2 Dúvida simulação em *Gate Level*: Estamos finalizando o projeto e surgiu uma dúvida referente a simulação em Gate Level, já que na aula do Lab08 vi que essa simulação é feita quando não é possível usar o *timing* por ele não reconhecer o tempo dos multiplicadores, a simulação em gate level seria apenas dos blocos que não possuem registradores?

Aconselho a fazer/assistir a aula sobre *timing* e o lab05.

A simulação *Gate Level* deverá ser feita sempre que disponível pois a mesma considera os atrasos dos circuitos (*Gates*) e por isso se aproxima muito da realidade. A simulação RTL não se aproxima em nada da realidade, nunca poderá ser considerada como parâmetro para funcionamento no mundo real. A simulação RTL é apenas prova de conceito já que a mesma não considera os atrasos intrínsecos do hardware (*Gates*) e, por isso, é chamada de RTL (*register transfer level*). Ou seja, ela não considera o *timing* (que nós estudamos). Se quiser fazer um teste, qualquer frequência de *clock* que você colocar no *testbench* funcionará a simulação em RTL.

No trabalho eu pedi para vocês utilizarem como base a FPGA Cyclone IV GX (qualquer uma da família que caiba o circuito) pois a mesma, por se tratar de uma FPGA mais antiga, tem disponibilizado os arquivos de tecnologia que tornam possível a simulação em *Gate Level*. Com a família MAX10 isso não é possível pois não existem esses arquivos.

Além disso, em simulações RTL, para vários modelos de FPGA, dados inicializados em memória são desprezados, tornando impossível a simulação do processador já que, tanto o programa quanto os dados estarão inicializados nas memórias de programa e de dados, respectivamente.

Nada a ver com o descrito acima é o TimeQuest Timing Analyser. Ele funcionará sempre, e deverá ser feito sempre para determinação dos parâmetros de operação do circuito (frequência de *clock* , etc).

3 Duvida entre questões c) e d): Professor, a diferença entre as perguntas relativas à máxima frequência operacional e a máxima frequência de operação seria com relação à que no primeiro seria considerado as piores condições de operação do ambiente (alta temperatura) e a segunda seria considerado as condições ótimas do ambiente (baixas temperaturas) ? Ou seria o contrário?

Quando você fizer a análise no TimeQuest, o que será reportado é a máxima frequência, baseada no timing do caminho crítico (lembrando que um caminho é SEMPRE referenciado da ENTRADA DE UM FLIP-FLOP À ENTRADA DO PRÓXIMO FLIP-FLOP). Essa análise será feita para o multiplicador e para o sistema (MIPS) e os valores encontrados serão as respostas ao item c.

No entanto, quando vocês acoplarem o multiplicador ao sistema (MIPS), as coisas mudam de figura. O timeQuest não sabe que a arquitetura proposta compreende o multiplicador (que necessita de vários pulsos de clock para entregar o resultado - latência grande) dentro do estágio de execução do sistema (MIPS). Assim sendo, um período de clock do sistema (MIPS) deve ser capaz de acomodar diversos períodos de clock do multiplicador já que o multiplicador deve receber os operandos, processá-los e entregar a produto dentro do estágio de execução do MIPS. Assim, basicamente, a máxima frequência de operação do sistema MIPS total, basicamente, será a máxima frequência reportada pelo TimeQuest para o multiplicador dividido pela quantidade de pulsos de clock necessária para que o multiplicador processe os operando e entregue o produto (resposta a letra d).

Exemplo: Digamos que no item c você encontrou 340MHz para máxima frequência operacional do multiplicador e 80MHz para máxima frequência operacional do sistema (MIPS) sem o multiplicador.

Se você interligar o sistema total, com estas frequências, o mesmo não funcionará para a instrução da multiplicação pois a freq. do multiplicador é apenas 4,25x maior que a do sistema sem ele. E seriam necessários, baseado no trabalho do multiplicador, 2N+2 (onde N é o número de bits do multiplicando ou do multiplicador, considerando que ambos tenham o mesmo tamanho em bits). Assim, 2N+2 = 2*16+2 = 34. A frequência de clock do multiplicador deve ser 34x maior que a frequência de clock do sistema para que o estágio de execução acomode toda a latência do multiplicador. Como

a máxima frequência reportada pra o multiplicador pelo timeQuest é 340MHz, se fizermos 80MHz (máxima freq. reportada para o sistema sem o multiplicador) * 34 = 2720MHz. Ou seja, não podemos fazer assim, pois ultrapassamos a máxima frequencia reportada pelo timeQuest para o multiplicador.

Dessa forma, o que temos que fazer é utilizar como frequência do MIPS:

freq. Sistema MIPS= freq. Multiplicador / 34 = 340MHz / 34 = 10 MHz

Como essa freq. encontrada (10MHz é menor que 80MHz, a mesma pode ser utilizada como freq. do sistema total.

Caso fosse maior que a reportada pelo timequest para o sistema sem o multiplicador, utilizaríamos a máxima freq. reportada.

Essa análise já é, basicamente a resposta do item g. Perceba como a utilização da arquitetura proposta para este multiplicador não casa bem à arquitetura do MIPS. Devido sua latência muito grande (2N + 2), o clock do sistema total fica comprometido

Obs.:

1) Latência do multiplicador é 2N + 2 se implementado exatamente igual ao diagrama de estados do Laboratório 4. Se for omitido o sinal de done e, consequentemente o estado S3, será 2N + 1. Se também for omitido o estado S0 e o sinal de start, 2N.

2) Como o multiplicador e o sistema MIPS sem o multiplicador funcionam com clocks distintos, o timequest conseguirá forneceres as máximas frequências pedidas no item c mesmo sendo rodado pro sistema total (ele analisará o caminho crítico para ambos os domínios de clock). No entanto, é mais fácil fazer separado. Daí a importância de termos hierarquias separadas para cada componente, individualmente.

4 Memória de Programa :Professor, para o TestBench do teste do MIPS a memória de programa deverá já estar gravada ou será necessário que o Testbench realize o boot das instruções na memória e posteriormente as execute?

Sintetize, ambas as memórias, já com informação. Memória de programa com as instruções codificadas e a memória de dados já com os dados A, B, C , D. Lembre-se que a memória de programa já conterá as duas sequências de instruções (a primeira apresentará o pipeline hazzard e a segunda , não), como pedido no guia do trabalho.

5 Sinais de Controle do módulo control: Professor, na figura da arquitetura MIPS a ser implementada existem sinais de controle em alguns blocos, minha dúvida é, o numero de sinais de controle representados em cada bloco será o número de sinais de controle real que cada bloco irá receber? Por exemplo, o módulo Register File Receberá 4 sinais de controle, a ALU 2 sinais de controle e assim suscetivamente?

Entendendo o funcionamento vocês terão condições de terminar quantos e quais serão os sinais de controle. O diagrama não é uma representação fidedigna.

6 Memoria de dados e addr decoding: Professor, não entendi muito bem como vai funcionar a memória de dados. A memoria interna começa em 0000h, e a memoria externa começa em nº * 500h? Então meu grupo sendo 1, a memoria externa começa em 500h? ou 1500h?

Número do Grupo multiplicado por 500h. Exemplo: Se seu grupo é o 1, o resultado será 0500h.

7 Como consigo manter o nome de um sinal para usar a diretiva *$init_signal_spy* para monitoramento de sinais internos em simulações GateLevel?

8 Em simulações gate *level*, a *netlist* do circuito é modificado para poder acomodar os parasitas dos *gates*. Desta forma, o motor de síntese altera os nomes das ligações do circuito que descrevemos pois, na verdade, um novo circuito é feito, compreendendo os parasitas. Esses nomes são gerados por software e se tornam praticamente indistinguíveis de outros, dada a complexidade dos mesmos.

9 Para forçar a barra e dizer à ferramenta de síntese que uma determinada ligação/nó de nossa descrição deve ter o nome mantido, podemos usar a diretiva *keep*. Exemplo:

*(*keep=1*) reg nomeDoNó;*

*(*keep=1*) wire nomeDoNó;*