Universidade Federal de Pernambuco
UFPE

Centro de Informática – CIn

Relatório do Projeto

Infra Estrutura de Hardware – if674

Recife, Outubro de 2011

Infra Estrutura de Hardware – if674

Especificação de Projeto

Desenvolvido por:

- Antônio Victor Palmeira Leite de Lima (avpll)
- Gisely Cristiane da Costa Melo (gccm)
- Hugo Leonardo Nascimento Almeida (hlna)
- José Araújo da Silva Neto (jasn)
- José de Arimatea Rocha Neto (jarn)

Monitor orientador:

- Leonardo da Silva Leandro (lsl2)

Índice

Introdução

Nosso projeto tem como base os conceitos aprendidos na disciplina de Infra Estrutura de Hardware, que serviram de máximo auxilio para a implementação de um processador do tipo multiciclo. As aulas da disciplina, assim como as técnicas, analises, conhecimento e ferramentas foram ministradas e disponibilizadas pelo professor durante o curso da disciplina, com a ajuda de sua equipe de monitores.

As instruções propostas pela especificação do projeto são encontradas no nosso processador. Para sua implementação, dividimos o projeto em três etapas conforme sugerido: Projetar o diagrama de blocos do Processador – utilizando uma cartolina para definir a sua arquitetura; Desenvolver uma Máquina de Estados do Processador – utilizando mais uma cartolina, observando os controles presentes na arquitetura; Implementação do projeto do processador, utilizando uma linguagem de descrição de hardware bastante conhecida e introduzida na disciplina de Sistemas Digitais: Verilog.

Este relatório contém informações essenciais de todas as entidades que compõem a CPU projetada, fazendo uma descrição especificada de cada entidade, onde é mostrada as funcionalidades da CPU de modo detalhado. O relatório também apresenta as “waveforms” obtidas nos testes assertivos do desenvolvimento do trabalho, além da Máquina de Estados do controle.

Descrição das Entidades

- **Extends1_32**

Entradas

LT(1 bit): sinal que será estendido.

Saídas

OutE3 (32 bits): sinal de entrada concatenado a 31 bits 0 colocados à esquerda dele

Objetivo

Transformar um sinal de um bit em uma palavra de 32 bits, estendido simplesmente com zeros. No nosso projeto isso só é feito com o sinal *LT* da *ALU*, por causa da instrução *slt*, que deve armazenar em um registrador especificado 31 zeros concatenados ao sinal *LT* resultante da operação da *ALU*.

Algoritmo

O *Extends1_32* concatena o sinal de entrada com 31 bits zero à esquerda dele.

- **Extends8_32**

Entradas

OutMem (32 bits): sinal que terá seu byte menos significativo estendido.

Saídas

OutE1(32 bits): sinal de entrada concatenado a 24 bits 0 à esquerda dele.

Objetivo

Transformar um sinal de 8 bits em uma palavra de 32 bits, estendido somente com bits 0. No nosso projeto isso só é feito com o byte menos significativo da palavra lida da memória no endereço enviado pelo PC quando ocorre uma exceção.

Algoritmo

O *Extends8_32* concatena o byte menos significativo do sinal de entrada com 24 bits zero à esquerda.

- **Extends16_32**

Entradas

SignSel(1 bit): Seleciona se a extensão é com ou sem sinal.

OutIr7 (16 bits): valor que será estendido.

Saídas

OutExtends16_32 (32 bits): valor de entrada concatenado a 16 bits 0 ou 1 (dependendo do bit de sinal) à esquerda dele.

Objetivo

Transformar um valor de 16 bits em uma palavra de 32 bits com ou sem sinal estendido.

Algoritmo

Caso seja com sinal, o *Extend16_32* concatena o sinal de entrada com 16 bits do mesmo valor do bit mais significativo do sinal de entrada. Caso contrario, o Extends16_32 concatena o sinal de entrada com 16 zeros.

- **Extends26_32**

Entradas

OutPc (32 bit): Sinal que vem do PC.

Jump(26 bits): Sinal que será deslocado

Saídas

OutE2(32 bits): sinal de entrada concatenado a dois zeros à direita.

Objetivo

Estende de 26 pra 28 bits e concatena com os 4 bits mais significativos do PC. Por fim, está com 32 bits.

Algoritmo

O *Extends26_32* estende o Jump e depois concatena com os quatro bits mais significativos do PC.

- **SLL16**

Entradas

OutExtends16_32(16 bits): sinal que será shiftado à esquerda 16 vezes.

Saídas

OutSLL16(32 bits): sinal de entrada concatenado a 16 bits 0 à direita dele, ficando o valor de entrada nos 16 bits mais significativos.

Objetivo

Dar um shift de 16 na entrada.

Algoritmo

O *SLL16* desloca o valor binário 16 posições para a esquerda resultando em um valor igual ao original multiplicado por 2^16.

- **SLL2**

Entradas

OutE3(32 bits): sinal que será deslocado.

Saídas

OutSLL2(32 bits): sinal de entrada deslocado duas vezes para a esquerda (os espaços são completados com 0).

Objetivo

Deslocar um valor binário n posições para a esquerda resulta em um valor igual ao original multiplicado por 2^n. O *SLL2* desloca o valor em duas posições, efetuando, assim, uma multiplicação por $2^2=4$, de uma maneira muito mais rápida do que seria feita através de operações na ALU.

Essa operação precisa ser feita rapidamente porque é feita com frequência, especialmente no cálculo de endereços de destino em instruções de desvio (neste caso *branchs*). O corpo desse tipo de instrução em geral contém a quantidade de posições entre o endereço atual e o de desvio, e cada posição é composta por 4 bytes. Como o cálculo do endereço de destino exige que se obtenha essa quantidade de posições em bytes, esse valor deve ser multiplicado por 4.

Algoritmo

O *SLL2* desloca o valor binário 2 posições para a esquerda resultando em um valor igual ao original multiplicado por 2^2.

- **StoreBox**

Entradas

Clk(1 bit): representa o clock do sistema.

StoreSel: Seleciona o tipo do Store (byte, half ou word).

out_MemReg(32 bits): valor originalmente contido na posição *offset+rs* da memória.

out_B(32 bits): valor do registrador *rt*.

Saídas

out_Store(32 bits): valor a ser armazenado na memória.

Objetivo

Esta entidade trata os casos peculiares de armazenamento *Store Half* (armazena apenas a metade menos significativa do valor de *rt* na posição de memória *offset+rs*) e *Store Byte* (armazena apenas o byte menos significativo de *rt* na posição *offset+rs*), garantindo que os outros 16 bits, no primeiro caso, ou 24 bits, no segundo caso, contidos originalmente na posição *offset+rs* sejam preservados.

Algoritmo

O *StoreBox* verifica o sinal StoreSel a fim de determinar se é uma operação *sb* (StoreSel =0), *sh* (StoreSel =1) ou sw (StoreSel = 2). No primeiro caso, concatena os 24 primeiros bits do conteúdo da posição da memória *offset+rs* com os 8 últimos bits do conteúdo de *rt*, no segundo caso são concatenados os 16 bits do conteúdo de *offset+rs* com os 16 últimos bits do conteúdo de *rt* e no ultimo não há concatenação, a saida conterá os 32 bits do conteúdo de RT.

- **LoadBox**

Entradas

clk(1 bit): representa o clock do sistema.

out_MemReg(32 bits): valor contido na posição especificada da memória.

LoadSel(2 bits): determina o tipo de carregamento (*load byte, load half* ou *load word*).

Saídas

out_Load(32 bits): valor lido da memória.

Objetivo

Este componente trata os casos de carregamento *Load Half* (carrega apenas a metade menos significativa do valor contido na posição de memória *offset+rs* no registrador *rt*) e *Load Byte* (carrega apenas o byte menos significativo do valor da posição *offset+rs* em *rt*).

Algoritmo

O *Load* verifica o sinal *LoadSel* a fim de determinar se é uma operação *lb* (*LoadSel*=00) ou *lh* (*LoadSel*=01) ou *lw* (*LoadSel*=10). No primeiro caso, carrega no registrador *rt* os 8 bits menos significativos do endereço de memória, completando 32 bits através da concatenação de 24 zeros à esquerda. No segundo caso, carrega em *rt* os 16 bits menos significativos do conteúdo da memória e completa 32 bits concatenando 16 zeros à esquerda. No terceiro caso, carrega em *rt* exatamente o que sai da memória.

- **Div**

Entradas

clk(1 bit): representa o clock do sistema.

rst(1 bit): representa o reset do sistema.

DivStart(1 bit): sinal de controle que informa o início da operação.

DivSel(1bit): Sinal que escolherá entre o registrador hi e o registrador lo para sair nas instruções mflo (quociente) e mfhi (resto).

OutA(32 bits): valor que será usado como dividendo na operação.

OutB(32 bits): valor que será usado como divisor na operação.

Saídas

paraDiv(1 bit): sinal enviado ao controle que indica o fim da operação.

DivZero(1 bit): sinal enviado ao controle que indica a exceção de divisão por 0.

OutDiv(32 bits): Resultado escolhido pelo DivSel

Objetivo

Este componente realiza a operação de divisão entre dois inteiros de 32 bits com sinal, enviando o quociente resultante ao registrador lo e o resto ao registrador hi, de modo a otimizar o tempo de execução dessa operação em relação ao que seria gasto se fossem utilizadas as operações de subtração, comparação, e deslocamento dos outros componentes da CPU.

Algoritmo

O algoritmo de divisão de inteiros utilizado é baseado em operações de Deslocamento e Subtração, usando internamente dois registradores, um de 64 bits e um de 32 bits (divisor e resto).

Inicialmente, é verificado se o divisor é igual a zero. Caso seja, é emitido o sinal DivZero = 1 e o controle interpretará isso como uma exceção.

Caso contrário, a operação prossegue. O valor do *divisor* é guardado nos 32 bits do divisor, e o valor *do dividendo* nos 32 bits mais à direita do resto. Depois é feita uma verificação dos sinais iniciais do *registrador A* e o *registrador B* e o armazenamento desses sinais para uso futuro.

Após a verificação de sinais, os valores negativos são alterados para positivos, pois o algoritmo só trata divisão de números positivos. O algoritmo em si consta de 33 iterações, com cada iteração contendo 3 etapas:

1. Subtrai o divisor do resto e guarda o resultado no resto.
2. Verifica se o resultado da subtração foi positivo ou negativo. Caso tenha sido positivo, o bit menos significativo do quociente é setado para 1 e é efetuado um *Shift Left* de 1 bit. Já se o resultado for negativo, o bit menos significativo do Quociente é setado para 0, é efetuado um *Shift Left* de 1 bit, e finalmente, o valor do Resto é restaurado, somando-se ele ao Divisor (Resto – Divisor + Divisor = Resto).

3. É efetuado um *Shift Right* de 1 bit no Divisor é verificado o número de iterações do algoritmo, se for igual a 33, o procedimento é encerrado. Caso contrário, ele volta para a primeira etapa.

Após o fim do algoritmo, o valor do quociente da divisão se encontra no registrador lo, e o valor do resto nos 32 bits menos significativos do registrador hi.

- **Controle**

Entradas

clock(1 bit): representa o clock do sistema.

reset(1 bit): representa o reset do sistema.

Opcode(6 bits): campo *opcode* da instrução lida da memória.

Funct(6 bits): campo *function* da instrução lida da memória.

shamt(5 bits): campo *shift amount* da instrução lida da memória.

O(1 bit): sinal da ALU que indica que o resultado de uma operação gerou *overflow*.

LT(1 bit): sinal da ALU que indica que o primeiro operando é menor que o segundo em uma operação de comparação.

GT(1 bit): sinal da ALU que indica que o primeiro operando é maior que o segundo em uma operação de comparação.

ET(1 bit): ALU que indica que o primeiro operando é igual ao segundo em uma operação de comparação.

Zero(1 bit): sinal da ALU que indica que o resultado de uma operação foi zero.

DivZero(1 bit): sinal do componente Div que indica uma tentativa de efetuar divisão por zero.

ParaDiv(1 bit): sinal do componente Div que indica o fim da operação de divisão.

N(1 bit): Indica que o resultado da operação é negativo.

Z(1 bit): Indica que o resultado da operação é zero.

Saídas

PcWrite(1 bit): sinal que habilita/desabilita escrita no registrador PC.

StoreSel(1 bit): sinal que determina a operação que será efetuada pelo componente *Store*.

Iord(3 bits): sinal que vai para o multiplexador que determina o endereço da Memória que será acessado.

SrcData(2 bits): sinal que vai para o multiplexador que determina o dado que será escrito em WriteData.

MemStatus(1 bit): sinal que determina se o valor contido na posição especificada da memória será lido (0) ou se será escrito (1) um dado nessa posição.

LoadSel(2 bits): sinal que determina a operação que será efetuada pelo componente *Load*.

IrWrite(1 bit): sinal que habilita/desabilita escrita no registrador de instruções.

SrcReg(2 bits): sinal que vai para o multiplexador que determina o valor a ser escrito em um registrador no Banco de Registradores.

RegWrite(1 bit): sinal que habilita/desabilita escrita no Banco de Registradores.

SrcA(3 bits): sinal que vai para o multiplexador que determina o primeiro operando a ser usado na ALU.

SrcB(2 bits): sinal que vai para o multiplexador que determina o segundo operando a ser usado na ALU.

AluOp(3 bits): sinal que determina a operação que será efetuada pela ALU.

DivStart(1 bits): sinal que solicita ao componente *Div* que inicie a operação de divisão.

PCSel(3 bits): sinal que vai para o multiplexador que determina o valor que irá para o registrador PC.

EpcWrite(1 bit): sinal que habilita/desabilita escrita no registrador EPC.

DivSel(1 bit): sinal que habilita/desabilita escrita nos registradores hi e lo.

ShiftSel(3 bits): Sinal que entra no Registrador de deslocamento selecionando qual o tipo do deslocamento.

SrcShift(1 bit): Sinal que vai para os multiplexadores que selecionam as entradas do Registrador de deslocamento.

LoadReg(1 bit): sinal que habilita sempre a escrita em um registrador. (Exemplo: A, B, AluOut, MemReg)

rstDiv(1 bit): Reseta o div.

SignSel(1 bit): Indica se haverá extensão com sinal ou sem sinal na extensão de 16 para 32 bits.

Objetivo

O controle tem que existir para que o processador funcione, pois sem ele as instruções não seriam diferenciadas e não existiriam meios para guiar o fluxo de execução de cada instrução. Dentre outros ele ativa e desativa multiplexadores, permite ou não escrita em unidades sequenciais além de enviar sinais de controle para outros componentes.

Algoritmo

O Controle funciona como uma máquina de estados. Inicialmente ele verifica o sinal *reset*. Caso não esteja acionado, ele verifica as entradas *opcode* e *funct,* segue para o estado correspondente à instrução por eles indicada, de onde segue para estados posteriores necessários à execução dela, enviando sinais aos componentes da CPU, instruindo-os sobre o que fazer em cada estado. Caso seja detectada uma exceção em algum desses estados, o que seria indicado pelas entradas de *Overflow* e *DivZero* ou se o *opcode/funct* forem inválidos, o que é obtido a partir de um valor guardado numa variável tipo “reg”, segue-se para o estado de tratamento da respectiva exceção. Em seguida, o controle retorna ao estado inicial. Caso o *reset* esteja acionado, todos os sinais importantes são zerados, e o controle retorna ao início.

- **CPU**

Entradas

clock(1 bit): representa o clock do sistema.

reset(1 bit): representa o reset do sistema.

Saídas

Nenhuma

Objetivo

Este componente é a junção de todas as entidades aqui citadas interligadas, de modo a funcionar perfeitamente da forma que foi implementada, seguindo a especificação para a execução das instruções listadas. É análoga à cartolina apresentada na primeira etapa.

Algoritmo

A CPU recebe os sinais de *clock* e *reset* e os envia a seus elementos internos para que funcionem em sincronia.

Descrição das Operações

- **Instruções tipo R**

add $rd, $rs, $rt

Após identificar a instrução *add* no estágio de decodificação, os valores dos registradores rs e rt são carregados nos registradores A e B, vindo do banco de registradores, e a operação de *Soma*, determinada pelo sinal *AluOp,* vindo da Unidade de Controle, é realizada na ALU. Posteriormente, o resultado é escrito no registrador de destino rd e uma nova instrução é buscada na memória.

and $rd, $rs, $rt

Após identificar a instrução *and* no estágio de decodificação, os valores dos registradores rs e rt são carregados nos registradores A e B e a operação de *And Lógico*, determinada pelo sinal *AluOp,* vindo da Unidade de Controle, é realizada na ALU. Posteriormente, o resultado é escrito no registrador de destino rd e uma nova instrução é buscada na memória.

div $rs, $rt

Após identificar a instrução *div* no estágio de decodificação, os valores dos registradores rs e rt são carregados nos registradores A e B e são, então, levados às entradas *A e B*, respectivamente, da entidade *Div*. O resto da divisão é escrito no registrador *hi*, e o quociente é escrito no registrador *lo* e uma nova instrução é buscada na memória. Vale salientar que esses dois registradores são internos do módulo DIV, não havendo, portanto, nenhuma relação com os registradores do Banco de Registradores.

jr $rs

Após identificar a instrução *jr* no estágio de decodificação, o valor do registrador rs é carregado no registrador A e em seguida é selecionado como novo valor do registrador PC e uma nova instrução é buscada na memória.

mfhi $rd

Após identificar a instrução *mfhi* no estágio de decodificação, o valor do registrador hi (interno do DIV - resto) é carregado no registrador rd e uma nova instrução é buscada na memória.

mflo $rd

Após identificar a instrução *mflo* no estágio de decodificação, o valor do registrador lo (interno do DIV - quociente) é carregado no registrador rd e uma nova instrução é buscada na memória.

sll $rd, $rt, shamt

Após identificar a instrução *sll* no estágio de decodificação, o valor do registrador rt é carregado no registrador B e então levado à entrada *In* do Registrador de Deslocamento. O shamt representa a quantidade de shifts a serem realizados. A operação de *Shift à Esquerda*, determinada pelo sinal *ShiftSel* vindo da Unidade de Controle, é então realizada no Registrador de Deslocamento. O sinal do controle, SrsShift, determina que entrarão no *In* e *N,* B e shamt, respectivamente. Posteriormente, o resultado é escrito no registrador de destino rd e uma nova instrução é buscada na memória.

sllv $rd, $rs, $rt

Após identificar a instrução *sllv* no estágio de decodificação, os valores dos registradores rs e rt são carregados nos registradores A e B, e então levados, respectivamente, aos multiplexadores que dão acesso às entradas *In* e *N* do Registrador de Deslocamento. A quantidade de shifts a serem realizados é determinado pelos 5 bits menos significativos do valor registrador B. A operação de *Shift à Esquerda*, determinada pelo sinal *ShiftSel,* vindo da Unidade de Controle, é então realizada no Registrador de Deslocamento. O sinal do

controle, SrsShift, determina que entrarão no *In* e *N,* A e B[4:0], respectivamente. Posteriormente, o resultado é escrito no registrador de destino rd e uma nova instrução é buscada na memória.

slt $rd, $rs, $rt

Após identificar a instrução *slt* no estágio de decodificação, os valores dos registradores rs e rt são carregados nos registradores A e B e a operação de *Comparação*, determinada pelo sinal Alu*Op* vindo da Unidade de Controle, é realizada na ALU. Posteriormente o sinal LT, que sai da ALU, é estendido e escrito no registrador de destino rd e uma nova instrução é buscada na memória.

sra $rd, $rt, shamt

Após identificar a instruçãos *sra* no estágio de decodificação, o valor do registrador rt é carregado no registrador B e então levado à entrada *In* do Registrador de Deslocamento. O shamt representa a quantidade de shifts a serem realizados. A operação de *Shift à Direito Aritmético*, determinada pelo sinal *ShiftSel* vindo da Unidade de Controle, é então realizada no Registrador de Deslocamento. O sinal do controle, SrsShift, determina que entrarão no *In* e *N,* B e shamt, respectivamente. Posteriormente, o resultado é escrito no registrador de destino rd e uma nova instrução é buscada na memória.

srav $rd, $rs, $rt

Após identificar a instrução *srav* no estágio de decodificação, os valores dos registradores rs e rt são carregados nos registradores A e B, e então levados, respectivamente, aos multiplexadores que dão acesso às entradas *In* e *N* do Registrador de Deslocamento. A quantidade de shifts a serem realizados é determinado pelos 5 bits menos significativos do valor registrador B. A operação de *Shift à Direita Aritmético*, determinada pelo sinal *ShiftSel,* vindo da Unidade de Controle, é então realizada no Registrador de Deslocamento. O sinal do controle, SrsShift, determina que entrarão no *In* e *N,* A e B[4:0], respectivamente. Posteriormente, o resultado é escrito no registrador de destino rd e uma nova instrução é buscada na memória.

srl $rd, $rs, shamt

Após identificar a instruçãos *srl* no estágio de decodificação, o valor do registrador rt é carregado no registrador B e então levado à entrada *In* do Registrador de Deslocamento. O shamt representa a quantidade de shifts a serem realizados. A operação de *Shift à Direito Lógico*, determinada pelo sinal *ShiftSel* vindo da Unidade de Controle, é então realizada no Registrador de Deslocamento. O sinal do controle, SrsShift, determina que entrarão no *In* e *N,* B e shamt, respectivamente. Posteriormente, o resultado é escrito no registrador de destino rd e uma nova instrução é buscada na memória.

sub $rd, $rs, $rt

Após identificar a instrução *sub* no estágio de decodificação, os valores dos registradores rs e rt são carregados nos registradores A e B e a operação de *Subtração*, determinada pelo sinal *AluOp,* vindo da Unidade de Controle, é realizada na ALU. Posteriormente, o resultado é escrito no registrador de destino rd e uma nova instrução é buscada na memória.

break

Após identificar a instrução *break* no estágio de decodificação, o sinal de PcWrite é setado para 0 e o controle não pede para buscar a próxima instrução.

rte

Após identificar a instrução *rte* no estágio de decodificação, o valor do registrador EPC é carregado no registrador PC e uma nova instrução é buscada na memória.

- **Instruções tipo I**

addi $rt, $rs, imediato

Após identificar a instrução *addi* no estágio de decodificação, o valor do registrador rs é carregado no registrador A e o dado imediato é selecionado como segundo operando. A operação de *Soma*, determinada pelo sinal Alu*Op* vindo da Unidade de Controle, é realizada na ALU. Posteriormente o resultado é escrito no registrador de destino rt e uma nova instrução é buscada na memória.

addiu $rt, $rs, imediato

Após identificar a instrução *addiu* no estágio de decodificação, o valor do registrador rs é carregado no registrador A e o dado imediato é selecionado como segundo operando. A operação de *Soma*, determinada pelo sinal Alu*Op* vindo da Unidade de Controle, é realizada na ALU, sem possibilidade de ocorrer a exceção *overflow*. Posteriormente o resultado é escrito no registrador de destino rt e uma nova instrução é buscada na memória.

beq $rs, $rt, offset

Na etapa de decodificação da instrução, o valor de PC sempre é somado ao valor contido no campo da instrução correspondente ao offset, multiplicado por 4. Após identificar a instrução *beq* no estágio de decodificação, os valores dos registradores rs e rt são carregados nos registradores A e B e a operação de *Comparação*, determinada pelo sinal Alu*Op,* vindo da Unidade de Controle, é realizada na ALU. Se o sinal ET da ALU for igual a 1, o resultado da operação de PC + (offset x 4) é escrito no registrador de destino PC. Em seguida uma nova instrução é buscada na memória.

bne $rs, $rt, offset

Após identificar a instrução *bne* no estágio de decodificação, é feito um procedimento similar ao da instrução anterior, mas quando o sinal ET da ALU é igual a 0. Em seguida uma nova instrução é buscada na memória.

blt $rs, $rt, offset

Após identificar a instrução *blt* no estágio de decodificação, é feito um procedimento similar ao da instrução anterior, mas quando o sinal LT da ALU é igual a 1. Em seguida uma nova instrução é buscada na memória.

bgt $rs, $rt, offset

Após identificar a instrução *bgt* no estágio de decodificação, é feito um procedimento similar ao da instrução anterior, mas quando o sinal GT da ALU é igual a 1. Em seguida uma nova instrução é buscada na memória.

bgez $rs, offset

Após identificar a instrução *bgez* no estágio de decodificação, é feita uma comparação entre o valor do registrador rs com o valor zero, que está no registrador zero. O registrador zero é identificado através do campo do registrador rt que vem zerado. A operação de *Comparação*, determinada pelo sinal Alu*Op,* vindo da Unidade de Controle, é realizada na ALU. Se o sinal LT da ALU for igual a 0, o resultado da operação de PC + (offset x 4) é escrito no registrador de destino PC. Em seguida uma nova instrução é buscada na memória.

beqm $rs, $rt, offset

Da mesma forma que ocorre com a instrução *beq* fazemos com a instrução *beqm,* com a diferença de que nessa instrução agora especificada compara-se o valor da memória com o valor do registrador em questão.

lb $rt, offset($rs)

Após identificar a instrução *lb* no estágio de decodificação, o valor do registrador rs é carregado no registrador A e o *offset* é selecionado como segundo operando. A operação de *Soma*, determinada pelo sinal Alu*Op* vindo da Unidade de Controle, é realizada na ALU. O resultado da operação é selecionado como posição da memória a ser acessada. O valor lido na posição especificada segue para a entidade Load, onde é realizada a operação de *Load Byte*, determinada pelo sinal *LoadSel* vindo da Unidade de Controle.

Posteriormente o resultado é escrito no registrador de destino rt e uma nova instrução é buscada na memória.

lh $rt, offset($rs)

Após identificar a instrução *lh* no estágio de decodificação, é feito um procedimento similar ao da instrução anterior, mas usando a operação *Load Half*. Em seguida uma nova instrução é buscada na memória.

lui $rt, imediato

Após identificar a instrução *lui* no estágio de decodificação, o dado imediato é deslocado de 16 bits para a esquerda e carregado no registrador de destino rt e uma nova instrução é buscada na memória.

lw $rt, offset($rs)

Após identificar a instrução *lw* no estágio de decodificação, é feito um procedimento similar ao das instruções *lh* e *lb*, mas usando a operação *Load Word*. Em seguida uma nova instrução é buscada na memória.

sb $rt, offset($rs)

Após identificar a instrução *sb* no estágio de decodificação, os valores dos registradores rt e rs são carregados nos registradores B e A e o *offset* é selecionado como segundo operando, e não B. A operação de *Soma*, determinada pelo sinal *OpALU* vindo da Unidade de Controle, é realizada na ALU. O resultado da operação é selecionado como posição da memória a ser acessada. O valor lido na posição especificada segue para a entidade Store, bem como o conteúdo do registrador B, onde é realizada a operação de *Store Byte*, determinada pelo sinal *StoreSEL* vindo da Unidade de Controle. Posteriormente o resultado é escrito na mesma posição de memória já especificada no AluOut e uma nova instrução é buscada na memória.

sh $rt, offset($rs)

Após identificar a instrução *sh* no estágio de decodificação, é feito um procedimento similar ao da instrução anterior, mas usando a operação *Store Half*. Em seguida uma nova instrução é buscada na memória.

slti $rt, $rs, imediato

Após identificar a instrução *slti* no estágio de decodificação, o valor do registrador rs é carregado no registrador A e o dado imediato é selecionado como segundo operando. A operação de *Comparação*, determinada pelo sinal Alu*Op* vindo da Unidade de Controle, é então realizada na ALU. Posteriormente o sinal LT, que sai da ALU, é estendido e escrito no registrador de destino rt e uma nova instrução é buscada na memória.

sw $rt, offset($rs)

Após identificar a instrução *sw* no estágio de decodificação, é feito um procedimento similar ao das instruções *sh* e *sb*, mas usando a operação *Store Word*. Em seguida uma nova instrução é buscada na memória.

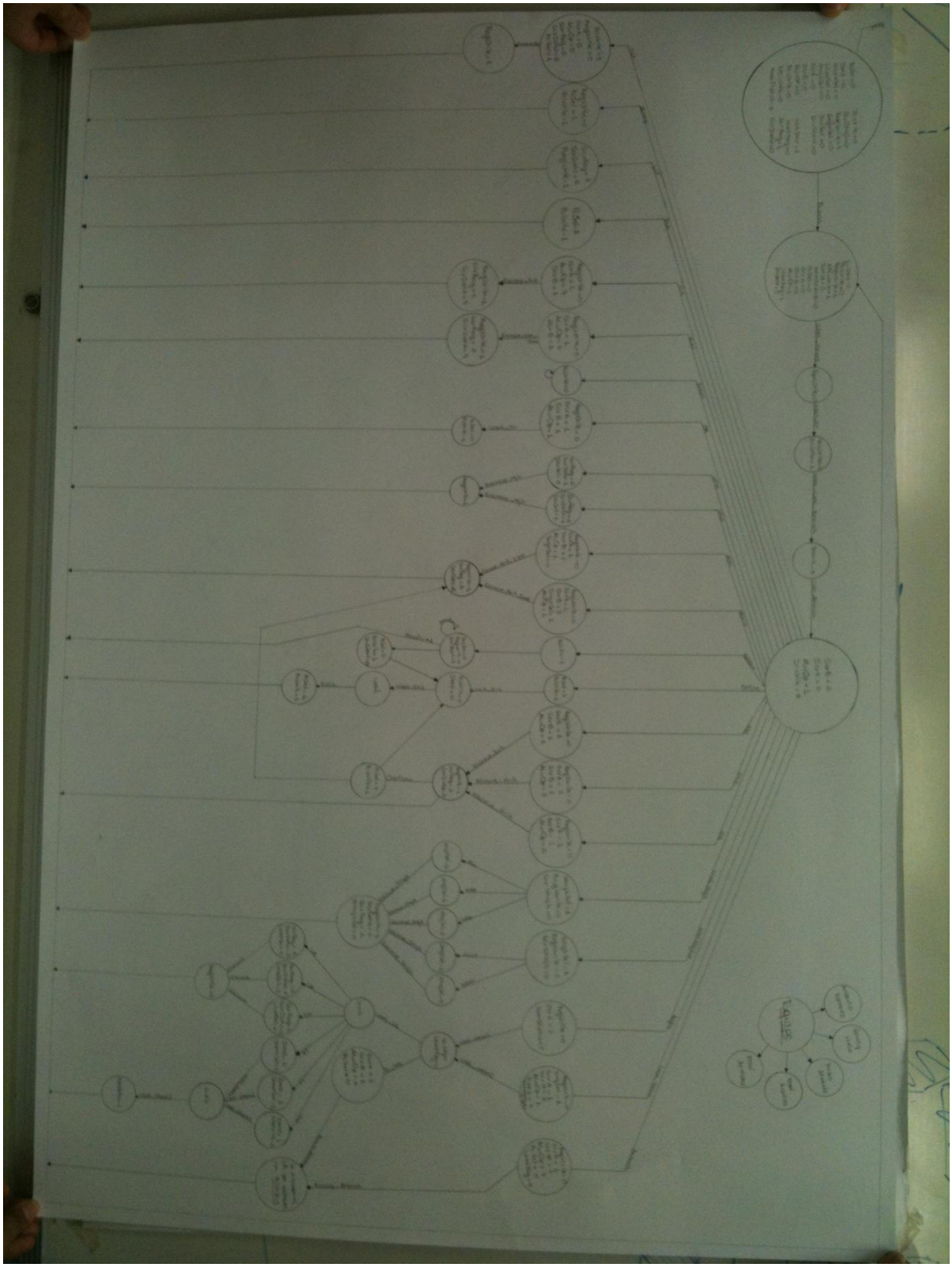
- **Instruções tipo J**

j offset

Após identificar a instrução *j* no estágio de decodificação, os 26 bits do campo offset da instrução são concatenados a dois bits 0 à direita e aos 4 bits mais significativos de PC à esquerda. Esse valor é escrito no registrador de destino PC. Em seguida uma nova instrução é buscada na memória.

jal offset

Após identificar a instrução *jal* no estágio de decodificação, o valor do registrador PC é carregado no registrador de destino r31. Em seguida é feito o mesmo procedimento da instrução *j* e uma nova instrução é buscada na memória.



Descrição dos Estados de Controle

1 - Reset: Neste estado, os principais sinais são zerados, com exceção RegWrite, auxDiv, rstDiv, ScrReg, que são setados como 1.

2 - Busca: Neste estado, o controle lê a instrução da posição de memória apontada por PC e escreve no registrador de instruções. PC recebe PC+4. O próximo estado será o *Wait_Mem1*.

3 - Wait_ir: Neste estado, o PC recebe PC+4 e o IrWrite é setado como 1, para podermos destrinchar a operação que está na memória(endereço do PC). O próximo estado será o Wait_Calc_Branch.

4 - Calc_Branch: Neste estado, o valor relativo ao possível branch será calculado. Setamos o IrWrite como zero para que não seja decodificada a instrução do PC+4 precipitadamente. O proximo estado será escolhido de acordo com o Opcode e com o Funct.

5 - Rte: Escreve no PC o valor de EPC e segue para a *Busca*.

6 - Jump: Seleciona para entrar no PC o valor do jump extendido e concatenado com 4 bits do PC. Segue para a *Busca*.

7 - Branch: Neste estado setamos o IrWrite como 0, para que não haja a decodificação da instrução indesejada, também haverá uma comparação na ULA entre os valores contidos nos registradores representados pelos campos rs e rt. O PcSel seleciona o valor do branch na entrada do PC, e o LoadReg é setado como 0 para podermos segurar o valor do cálculo do branch no registrado AluOut. O proximo estado será o Escreve_Branch.

8 - Lui: Neste estado setamos o ScrReg como 3, que escolhe o valor do campo rt como o Write Register e setamos o ScrData como 5, que escolhe o valor do imediato, que é estendido com sinal de 16 para 32 e depois e depois sofre um shift left de 16, e por fim setamos RegWrite como 1 para escrever o valor do imediato no registrador adequado.

9 - Jr: Soma o valor que sai do rs com o valor que sai de rt (sempre zero) e vai para o estado *Wait_Jr*.

10 – OpIne: Se ocorreu Opcode inexistente, o valor 253 é setado para entrar no PC e prossegue para o estado *Wait_Ex1.*

11 - Div: O sinal DivStart é setado para 1, assim a divisão começa. Ele continua nessa estado até que a divisão acabe (ParaDiv = 1), e ele segue para o estado de *Busca,* ou ocorra uma exceção (DivZero = 1), que segue para o estado *Ex_DivZero*.

12 - Sll: Seleciona o Shift Left no registrador de deslocamentos e segue para o *Escreve_Shift.*

13 - Sra: Seleciona o Shift Right Aritmético no registrador de deslocamentos e segue para o *Escreve_Shift.*

14 - Srl: Seleciona o Shift Right Lógico no registrador de deslocamentos e segue para o *Escreve_Shift.*

15 - Sllv: Seleciona o Shift Left no registrador de deslocamentos e segue para o *Escreve_Shift.*

16 - Srav: Seleciona o Shift Right Aritmético no registrador de deslocamentos e segue para o *Escreve_Shift.*

17 - Break: Seta PcWrite = 0 e continua nessa estado em um loop infinito.

18 - Mfhi: Escolhe para sair do módulo DIV o valor do registrador hi (resto) e escolhe o rd como registrador a ter o valor vindo do DIV salvo. Prossegue para *Escreve_Mfh*.

19 - Mflo: Escolhe para sair do módulo DIV o valor do registrador lo (quociente) e escolhe o rd como registrador a ter o valor vindo do DIV salvo. Prossegue para *Escreve_Mfh*.

20 - Slt: Manda a ULA comparar os valores de rs e rt e segue para *Escreve_Slt.*

21 - Slti: Manda a ULA comparar os valores de rs e imediato estendido e segue para *Escreve_Slt_Ime.*

22 - Jal: Carrega o valor do PC (PC+4, pois já está atualizado) para a ULA. Seleciona, para ser escrito no próximo clock, o registrador 31 e o valor da ULA(PC+4). Seleciona para entrar no PC o valor do jump extendido e concatenado com 4 bits do PC. Segue para a *Escreve_jal*.

23 – Escreve_Jal: Escreve, no registrador 31, o valor que sai da ULA (PC+4) e segue para *Busca*.

24 – Addi: Neste estado fazemos uma soma do valor contido no registrado indicado pelo rs com o valor contido no campo imeditado estendido de 16 para 32 bits com sinal e gravamos o resultado no registrador indicado pelo campo rt. O proximo estado é o Escreve_Arit_Ime.

25 - Addiu: Assim como no Addi, neste estado será feita uma soma entre o valor contido no registrado contido no registrado identificado pelo campo RS e o valor contido no imediato, que será estendido de 16 para 32 bits. A diferença aqui é que a extensão será feita ignorando o sinal do número. O próximo estado será o Escreve_Arit_ime.

26 - Add: Neste estado, o rs e o rt são escolhidos para operação de soma, na ULA, essa operação é feita, o resultado guardado na ALUout e em seguida o controle prossegue para o estado *Escreve_Arit*.

27 - Sub: Neste estado, o rs e o rt são escolhidos para operação de subtração, na ULA, essa operação é feita, o resultado guardado na ALUout e em seguida o controle prossegue para o estado *Escreve_Arit*.

28 - And: Neste estado, o rs e o rt são escolhidos para operação de and, na ULA, essa operação é feita, o resultado guardado na ALUout e em seguida o controle prossegue para o estado *Escreve_Arit*.

29 – LoadStore: Neste estado o RegWrite é setado como 0 para não decodificar uma instrução desejada e o valor contido no campo offset é estendido sem sinal e somado ao valor do campo rs, calculando o valor do endereço de memória que será usado posteriormente. Setamos o Iord como 1 para acessarmos o valor desejado na memória e o MemStatus é setado como 0 o que faz esse acesso ser de leitura. O próximo estado é Wait_Mem2.

30 - Escreve_Arit_Ime: Este estado funciona auxiliando a escrita do valor calculado pelo Addi ou pelo Addiu. Aqui há uma comparação entre o valor da flag de Overflow com 1. Se a comparação for verdadeira o próximo estado será o Overflow, se não, a escrita ocorre normalmente com os valores adequado e o próximo estado se a Busca.

31 - Beqm: Como no beqm a comparação será feita entre o valor contigo no resgitrado equivalente ao campo rt com o valor contido na posição de memória contida no rs, neste estado temos com setar o RegWrite como 0, para que uma instrução indesejada seja destrinchada. Depois disso o valor contido na posição do valor contido no rs da memória é carregado e enviado diretamente para a ULA para que possa ser feita a comparação. O próximo estado será Wait_Mem2.

32 - Wait_Mem2: Estado auxiliar para esperar o valor ser carregado da memória. Nele também é feita uma comparação do Opcode com 0x2c, se for verdadeira quer dizer que viemos do beqm, então mudamos o LoadReg para 0, segurando o valor do branch no AluOut e o próximo estado se Pc_Set, se for falsa é porque viemos do LoadStore, logo, o próximo estado será Wait_LS.

33 - Wait_Ex1: Escolhe para entrar na Memória o valor vindo do PC e segue para o estado *Wait_Ex2*.

34 - Exc2: Escolhe para entrar no PC o valor vindo da Memória. Prossegue para a *Busca.*

35 - Escreve_Shift: Escreve no rd o valor que sai do registrador de deslocamentos e seta esse para não fazer mais nada. Prossegue para a *Busca*.

36 - Escreve_Slt: Escreve em rd o valor estendido de LT e segue para a *Busca*.

37 - Overflow: Se ocorreu overflow, o valor 254 é setado para entrar no PC e prossegue para o estado *Wait_Ex1.*

38 - Escreve_Arit: Se ocorreu overflow em algum dos estados anteriores, ele segue para o estado *Overflow,* se não, ele escreve no registrador rd o valor vindo da ULA e segue para a *Busca*.

39 - Lb: Setamos o ScrReg para 3, que seleciona o valor do rt como entrada do valor para o Write Register e ScrData seleciona o valor do LoadBox como entrada do Write Data e o LoadSel seleciona o tipo de load que vai ser feito na LoadBox. O próximo estado será Escreve.

40 - Lh: Funciona do mesmo modo que o Lb, mudando apenas o valor do LoadSel que irá selecionar o load half. O próximo estado será o Escreve.

41 - Lw: Funciona do mesmo modo que o Lb, mudando apenas o valor do LoadSel que irá selecionar o load word. O próximo estado será o Escreve.

42 - Sb: Neste estado setamos Iord como 1, que seleciona o valor da saída da ULA como entrada na memória e selecionamos o tipo como byte de load na LoadBox. O próximo estado será Wait_Store1.

43 - Sh: Funciona do mesmo modo que o Sb, mudando apenas a seleção na LoadBox, que será para um load half. O Próximo estado será Wait_Store1.

44 - S2: Funciona do mesmo modo que o Sb, mudando apenas a seleção na LoadBox, que será para um load word. O Próximo estado será Wait_Store1.

45 - PcSet: Neste estado a comparação do beqm (valor carregado da memória com o valor contido no campo rt) é feita, o PcSel é setado para receber a saída da ULA e o próximo estado é Escreve_Branch.

46 - Wait_Store1: Neste estado apenas damos um wait, esperando a memória carregar o valor onde será dado o store. O próximo estado será Wait_Store2.

47 - Wait_Calc_Branch: Neste estado, o PcWrite é mudado para 0, para não sobrescrevermos o valor PC+4. O próximo estado será o Calc_Branch.

48 - Escreve_Slt_Ime: Escreve em rt o valor extendido de LT e segue para a *Busca*.

49 - Escreve_Mfh: Seta RegWrite = 1 para poder escrever no registrador rd o valor da saída do Div. Prossegue para a *Busca*.

50 - Load_Shift: O SrcShift é escolhido para se ter como entrada no registrador de deslocamento os valores de rt e shamt em IN e N, respectivamente. O valor ShifSel é setado para carregar esses valores para seu interior. Daí, dependendo do Opcode, ele pode seguir para *Sll*, *Sra* e *Srl*.

51 - Load_Shift_V: O SrcShift é escolhido para se ter como entrada no registrador de deslocamento os valores de rs e os 5 bits menos significativos de rt em IN e N, respectivamente. O valor ShifSel é setado para carregar esses valores para seu interior. Daí, dependendo do Opcode, ele pode seguir para *Sllv* e *Srav*.

52 - Wait_LS: Neste estado damos um wait para esperar o valor lido da memória ser carregado no MemReg e um case do Opcode para vermos qual é a função desejada(Lb, Lh, Lw, Sb, Sh, Sw). O próximo estado será escolhido de acordo com a função.

53 - Escreve: Neste estado apenas setamos o RegWrite como 1 para que seja escrito o valor lido da memória no registrador adequado. O próximo estado é a Busca.

54 - Wait_Store2: Neste estado setamos o MemStatus como 1 para escrever o valor que vem da LoadBox na memória. O próximo estado é a Busca.

55 - Wait_Jr: Escreve no PC o valor que veio da ULA e vai para o estado *Busca*.

56 - Escreve_Branch: Neste estado será feita uma comparação entre a flag desejada com o valor correto, dependendo do Opcode. Para cada tipo de branch será feita uma comparação diferente, como por exemplo, para o beq, que compara se o valor da flag ET é igual a 1, se for verdade o PcWrite é mudado para 1 e o valor do brach é escrito no PC, se não, o PC não será sobrescrevido. Dependendo do Opcode também será feita neste estado a comparação para todos os outros branches. O próximo estado será a Busca.

57 - Ex_DivZero: Se ocorreu uma divisão por zero, o valor 255 é setado para entrar no PC e prossegue para o estado *Wait_Ex1.*

58 - Wait_Mem1: Neste estado, o controle espera um ciclo de clock antes de seguir para o estado *Wait_Ir*. O EpcWrite é setado como 0, o que significa que o Epc não receberá o valor de PC+4.

59 - Wait_Ex2: Nesse estado espera um ciclo de clock para prosseguir para o estado *Exc2.*

60 - RstDiv: Nesse estado é mandado um comando para o módulo DIV resetar tudo que se encontra no seu interior (sinais e registradores). Prossegue para o *Div.*

Conjunto de Entidades

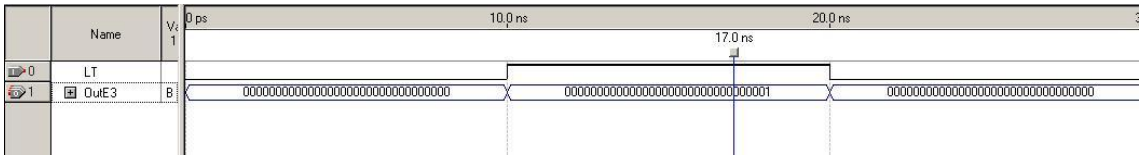
Entidade: Extends1_32

Descrição das Portas:

LT: sinal de 1 bit que será estendido.

OutE3 : sinal de entrada concatenado a 31 bits 0 à esquerda.

Descrição da Simulação: O sinal de entrada é sempre estendido para 32 bits, concatenando-se 31 bits 0 à esquerda, e enviado para a saída.



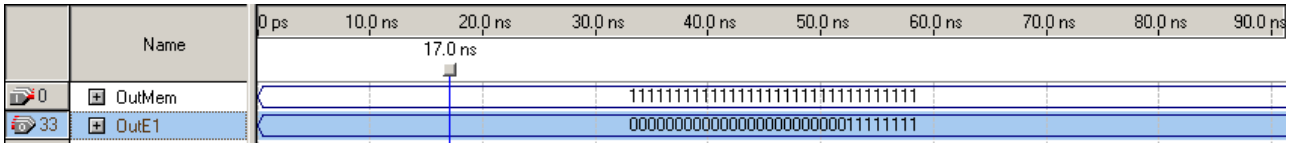
Entidade: Extends8_32

Descrição das Portas:

OutMem[7:0]: sinal de 32 bits que terá os 8 bits menos significativos concatenados a 24 zeros .

OutE1 : 8 bits menos significativos do sinal de entrada concatenado a 24 bits 0 à esquerda.

Descrição da Simulação: O sinal de entrada é sempre de 32 bits, concatenando-se 24 bits 0 à esquerda dos 8 bits menos significativos da entrada, e enviado para a saída.



Entidade: Extends16_32

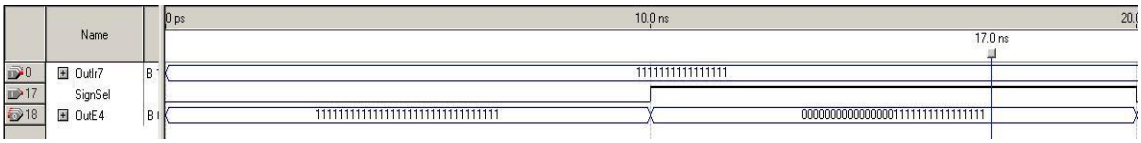
Descrição das Portas:

OutIr7: sinal que de 16 bits será estendido.

SignSel: sinal que seleciona o tipo da extensão(com ou sem sinal).

OutE4 : sinal de entrada(OutIr7) concatenado a 16 bits 0 ou 1 à esquerda.

Descrição da Simulação: O sinal de entrada sempre é estendido para 32 bits, repetindo-se à esquerda o bit mais significativo do sinal de entrada, se for com sinal e adicionado-se 16 zeros à esquerda do sinal de entrada se for sem signal, após isso os 32 bits são enviados para a saída.



Entidade: Extends26_32

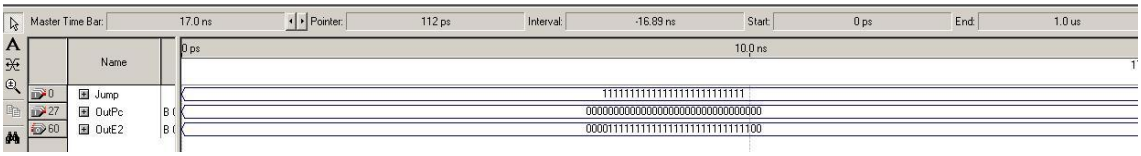
Descrição das Portas:

Jump: Sinal de 26 bits que será concatenado com 4 mais significativos de PC à esquerda e com dois zeros à direita.

OutPC: Valor do qual será retirado os 4 bits mais significativos para ser concatenado com Jump.

OutE2: Sinal de entrada concatenado com PC[31:28] à esquerda e com 2 zeros à direita.

Descrição da Simulação: A entrada sempre é concatenada com PC[31:28] à esquerda e com 2 zeros à direita, o resultado é enviado para a saída.



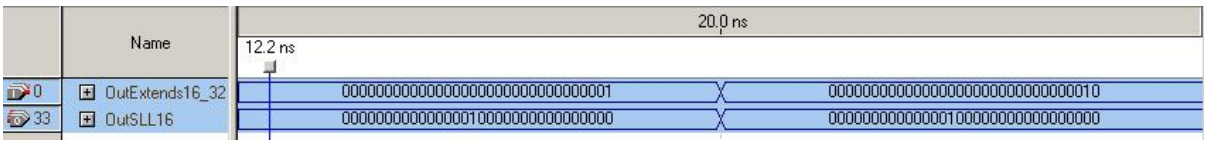
Entidade: SLL16

Descrição das Portas:

OutExtends16_32: sinal de 32 bits que sofrerá o shift.

OutSLL16: sinal de entrada shiftado 16 vezes.

Descrição da Simulação: O sinal de entrada sempre sofre 16 shifts a esquerda e é enviado para a saída.



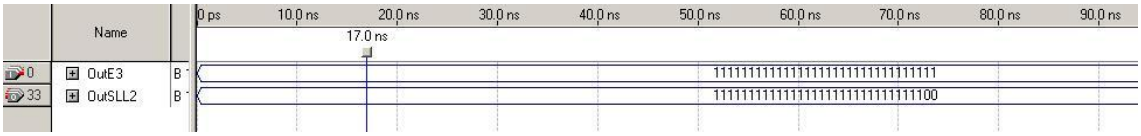
Entidade: SLL2

Descrição das Portas:

OutE3: sinal de 32 bits que receberá dois shifts a esquerda.

OutSLL2 : sinal de entrada deslocado duas posições para a esquerda.

Descrição da Simulação: Os 30 bits menos significativos do sinal de entrada são concatenados a 2 bits 0 à direita, e o resultado é enviado para a saída.



Entidade: StoreBox

Descrição das Portas:

StoreSel: determina o tipo de armazenamento (store word, *store half* ou *store byte*).

clk: representa o clock do sistema.

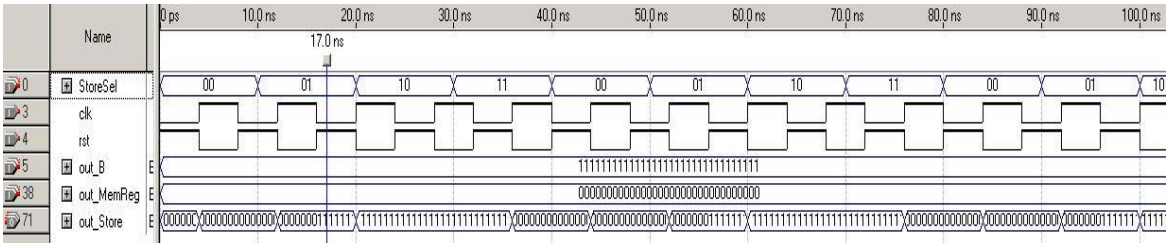
rst: representa o reset do sistema.

Out_MemReg: valor originalmente contido na memória.

Out_B: valor do registrador *rt*.

OutStore: valor a ser armazenado na memória.

Descrição da Simulação: A cada subida de clock, se o sinal StoreSel for 0, concatena-se os 24 bits mais significativos de Out_MemReg com os 8 menos significativos de Out_B. Se StoreSel for 1, concatena-se os 16 bits mais significativos de Out_MemReg com os 16 menos significativos de Out_B, se StoreSel for 2 o sinal de saída recebe os 32 bits de Out_B. O sinal resultante é enviado à saída.



Entidade: LoadBox

Descrição das Portas:

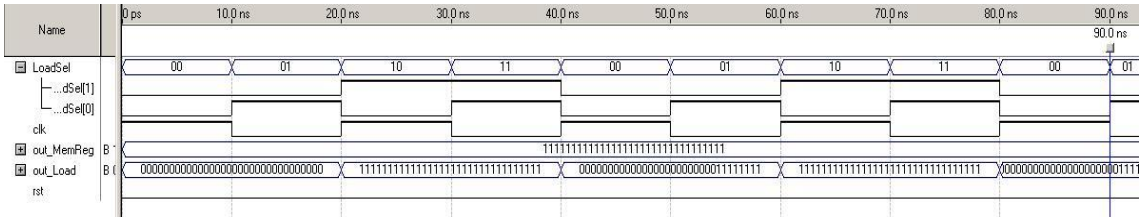
LoadSel: determina o tipo de carregamento (*load word, load half* ou *load byte*).

Out_MemReg: valor contido em uma posição da memória.

Out_Load: valor lido da memória.

rst: representa o rst do sistema.

Descrição da Simulação: Se o sinal LoadSel for 0, concatena-se 24 bits zero à esquerda dos 8 bits menos significativos de Out_B. Se LoadSel for 11, concatena-se 16 bits zero à esquerda dos 16 bits menos significativos de Out_B. Se LoadSel for 2,a saída será o próprio Out_B. O sinal resultante é enviado à saída.



Entidade: Div

Descrição das Portas:

DivSel: sinal que escolherá a saída(cociente ou resto).

clk: representa o clock do sistema.

rst: representa o reset do sistema.

DivStart: sinal de controle que indica o início da operação.

paraDiv: sinal para representar o fim da divisão.

OutA: valor que será usado como dividendo na operação.

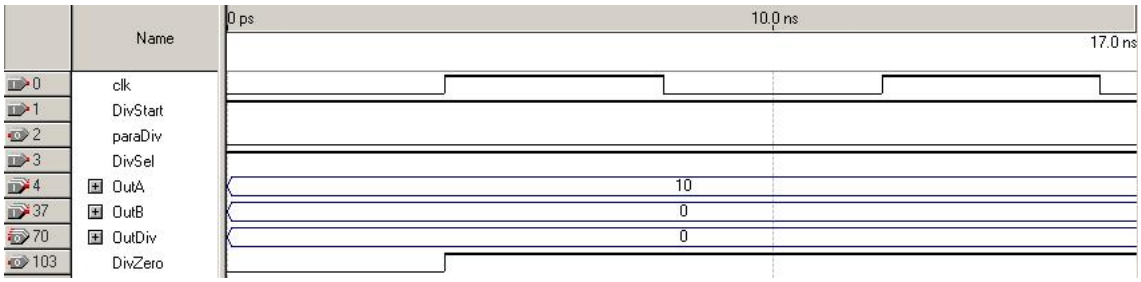
OutB: valor que será usado como divisor na operação.

divZero: sinal de saída que indica a exceção de divisão por zero.

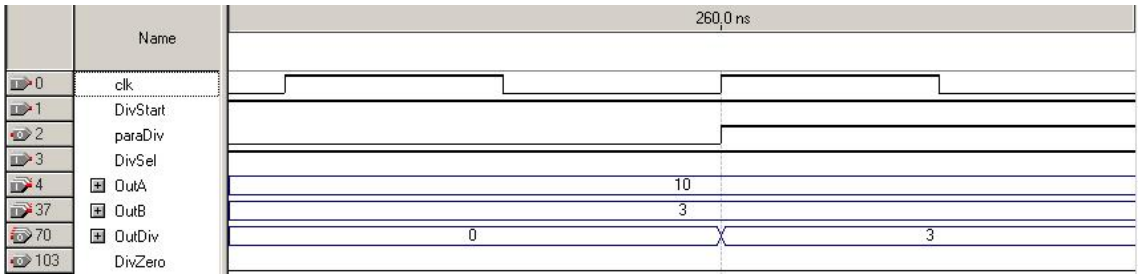
OutDiv: conterá a saída da divisão.

Descrição da Simulação: O algoritmo usado sempre leva 33 ciclos de clock até dar a operação de divisão como encerrada, a menos que seja detectada uma tentativa de divisão por zero, quando levará 2,5 ciclos para que seja interrompida e o sinal divZero acionado. O DivSel escolherá o saída da divisão(usado nas instruções Mfhi e Mflo),

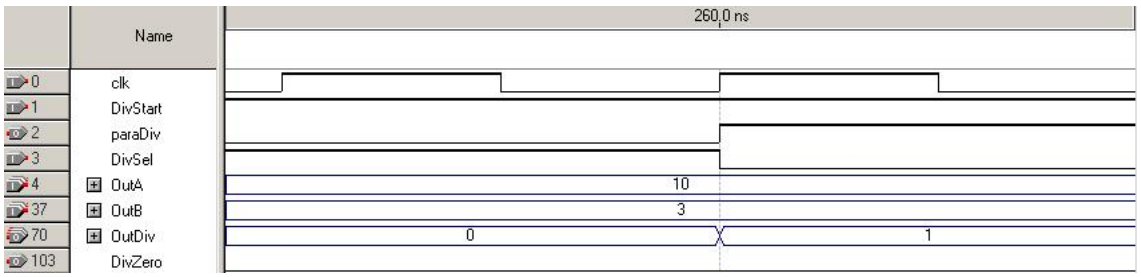
quando o DivSel é 0 a saída conterá o valor do registrador auxiliar hi, que contém o resto da divisão. Já quando o DivSel é 1 a saída conterá o valor do resgistrador auxiliar lo, que contém o cociente da divisão.



DIV – DivZero



DIV – Saída Quociente



DIV – Saída Resto

Entidade: IR(Registrador de Instruções)

Descrição das Portas:

clk: representa o clock do sistema.

rst: representa o reset do sistema.

IrWrite: sinal que controla a atualização dos valores contidos no IR.

OutMem: entrada do IR(instrução que vem da memória).

OutIr1 : saída quem contém OutMem[31:26].

OutIr2 : saída quem contém OutMem[25:21].

OutIr3 : saída quem contém OutMem[20:16].

OutIr4 : saída quem contém OutMem[15:11].

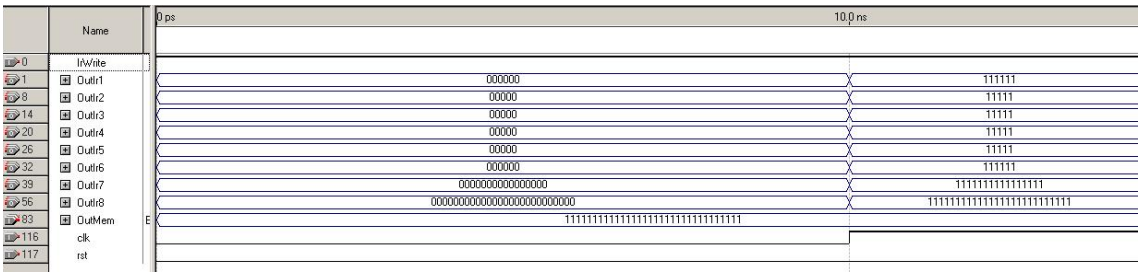
OutIr5 : saída quem contém OutMem[10:6].

OutIr6: saída quem contém OutMem[5:0].

OutIr7 : saída quem contém OutMem[15:0].

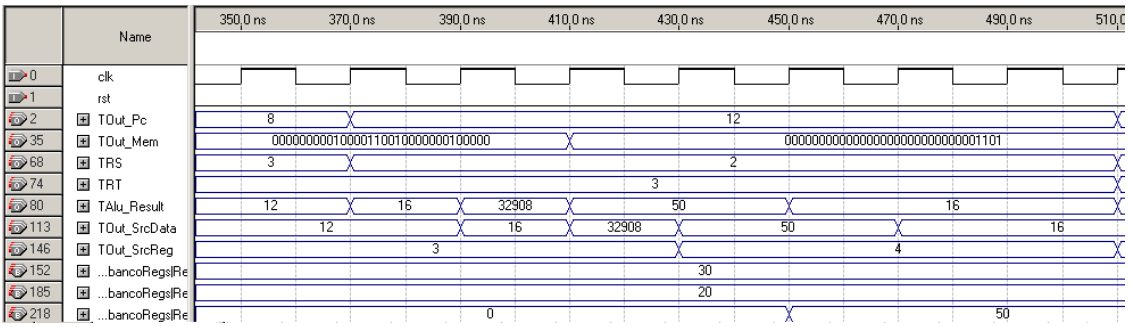
OutIr8 : saída quem contém OutMem[25:0].

Descrição da Simulação: Sempre que houver uma subida de clock e que o IrWrite for 1 haverá uma atualização do OutMem, pois uma nova instrução deve ser decodificada, ou seja todos os valores de saída serão alterados de acordo com a nova instrução.

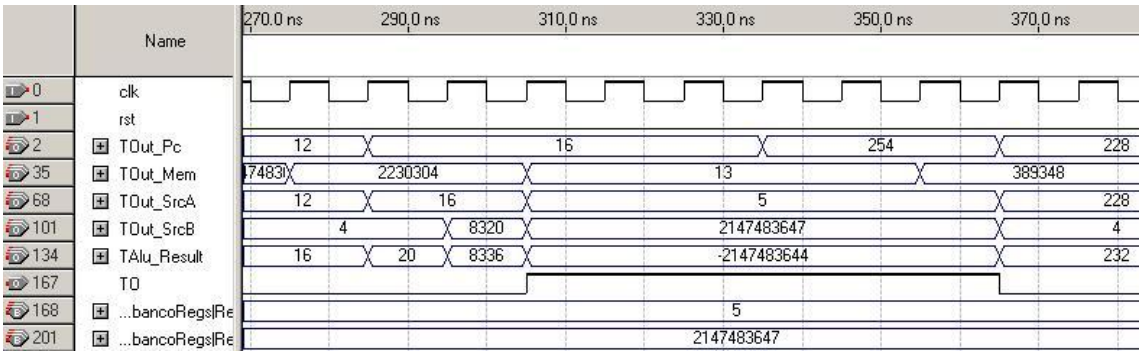


Simulações de Instruções

Instrução add



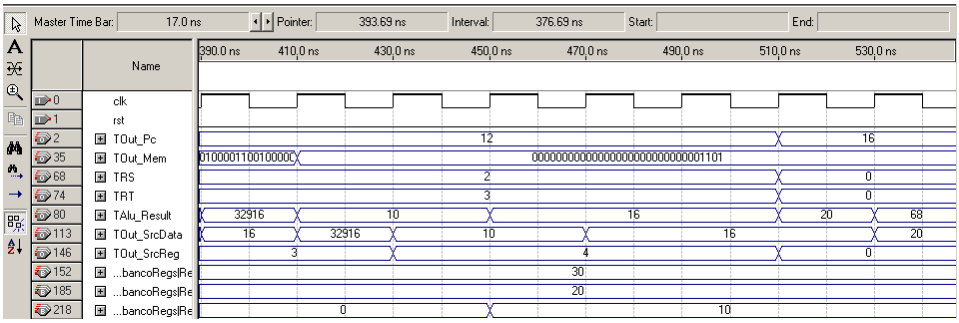
Adição sem Overflow



Adição com Overflow, vemos o valor 254 saindo do PC.

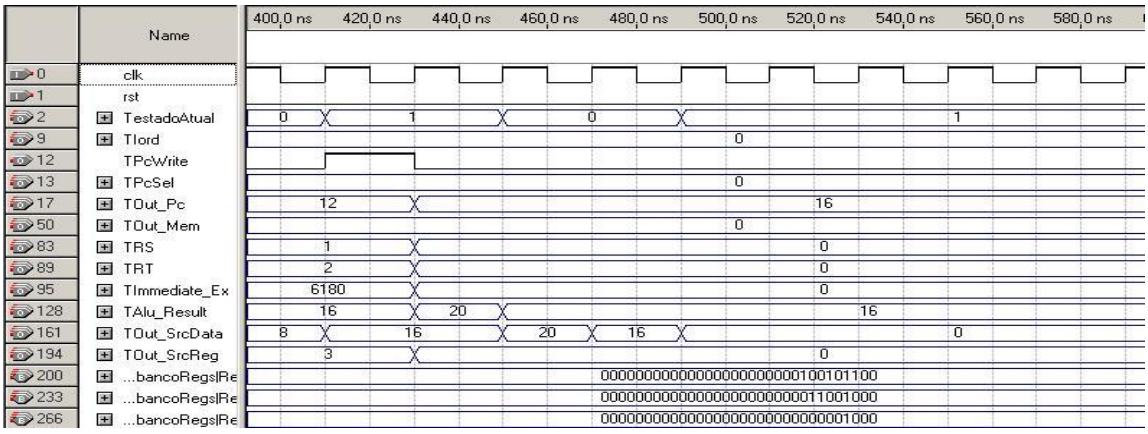
Descrição da simulação: Na simulação acima, é carregado para o registrador 1 o valor 20 e para o registrador 2 o valor 30. No estado add os registradores A e B já contêm os valores que estão contidos nos registradores 1 e 2. Então, neste estágio é feita a soma e seu resultado sai no Alu_Result. No próximo estado, o Escreve_Arit (00001001), o valor de Alu_Result é salvo no registrador 3, ou se ocorreu exceção ele prossegue com a rotina de tratamento.

Instrução sub



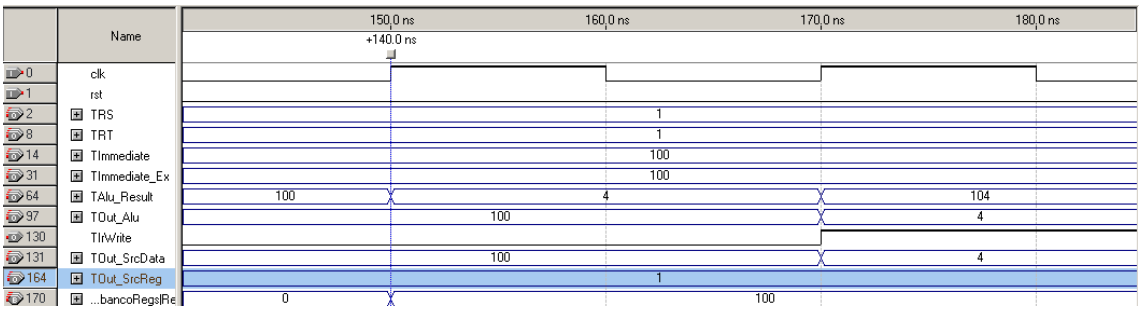
Descrição da simulação: Na simulação acima, é carregado para o registrador 1 o valor 20 e para o registrador 2 o valor 30. No estado sub os registradores A e B já contêm os valores que estão contidos nos registradores 1 e 2. Então, neste estágio é feita a subtração e seu resultado sai no Alu_Result. No próximo estado, o Escreve_Arit (00001001), o valor de Alu_Result é salvo no registrador 3. Como não foi gerado overflow, retorna-se ao estado Busca, para a próxima instrução (com PC + 4).

Instrução and



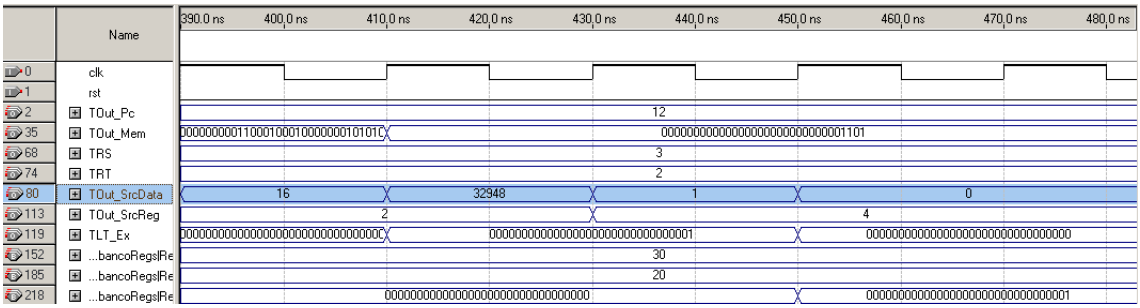
Descrição da simulação: Na simulação acima, é carregado para o registrador 1 o valor 00000000000000000000000100101100 e para o registrador 2 o valor 00000000000000000000000011001000. No estado and os registradores A e B já contêm os valores que estão contidos nos registradores 1 e 2. Então, neste estágio é feita o and bit a bit e seu resultado sai no Alu_Result. No próximo estado, o Escreve_Arit (00001001), o valor de Alu_Result é salvo no registrador3 e retorna ao estado Busca, para a próxima instrução (com PC + 4).

Instrução addi e addiu



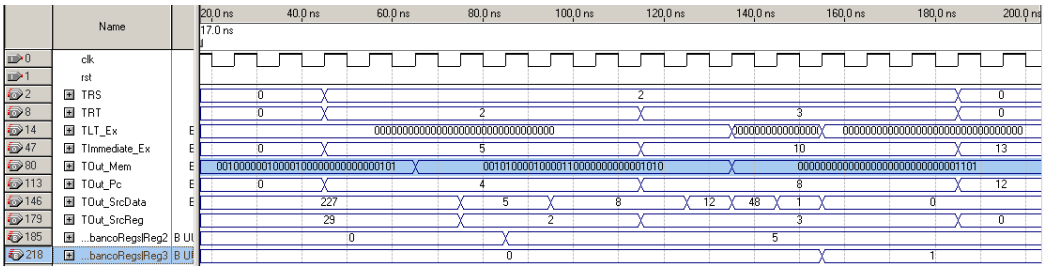
Descrição da simulação: Na instrução acima, nós estamos salvando no registrador 1, o valor do registrador 1 (zero) mais cem. O valor do imediato sai estendido para a ULA, onde é feita a soma e o resultado fica em Out_Alu. Posteriormente, esse valor é salvo no registrador 1. Após isso, o controle volta à Busca, com PC + 4. Os waveforms para as duas instruções, nesse caso, foram iguais.

Instrução slt



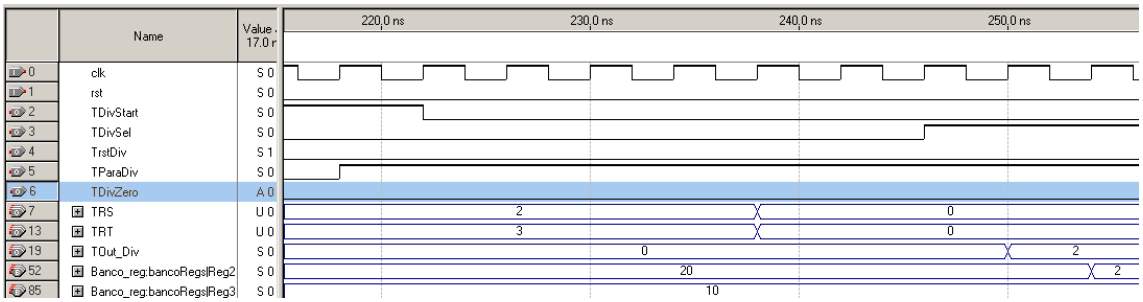
Descrição da simulação: No estado SLT, os valores de A e B já foram carregados com os valores do registrador 1 (trinta) e do registrador 2 (vinte), respectivamente. Na ALU, a comparação B < A é feita. Caso B seja menor que A, o sinal LT é ativado e estendido (de 1 para 32 bits), se não, o LT será zero, e esse valor também será estendido. O valor é escrito no registrador 3.

Instrução slti

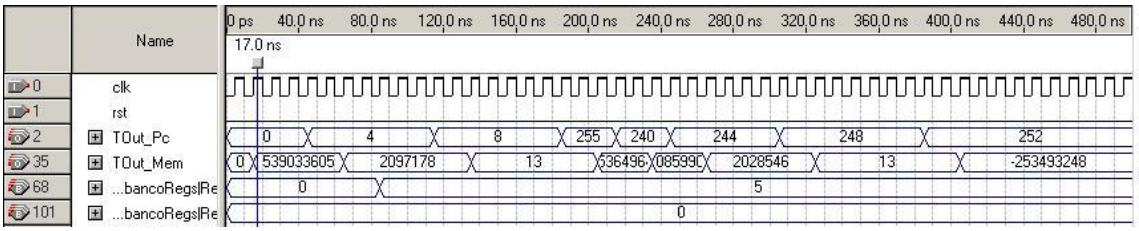


Descrição da simulação: O valor de A e B são carregados com os valores do registrador 2 (cinco) e o imediato estendido (dez), respectivamente. Na ALU, a comparação A < B é feita. Caso reg1 seja menor que o imediato, o sinal LT é ativado e estendido (de 1 para 32 bits). Caso A seja menor que B, o sinal LT é ativado e estendido (de 1 para 32 bits), se não, o LT será zero, e esse valor também será estendido. O valor é escrito no registrador 3.

Instrução div



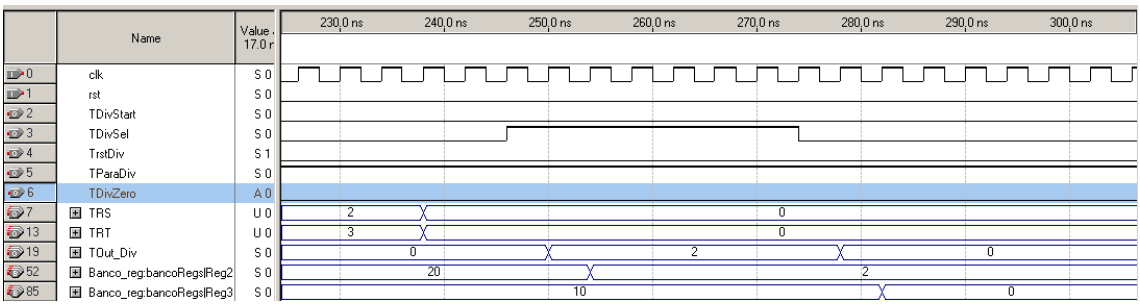
Divisão sem exceção



Como houve a divisão por zero, vemos o valor 255 saindo do PC.

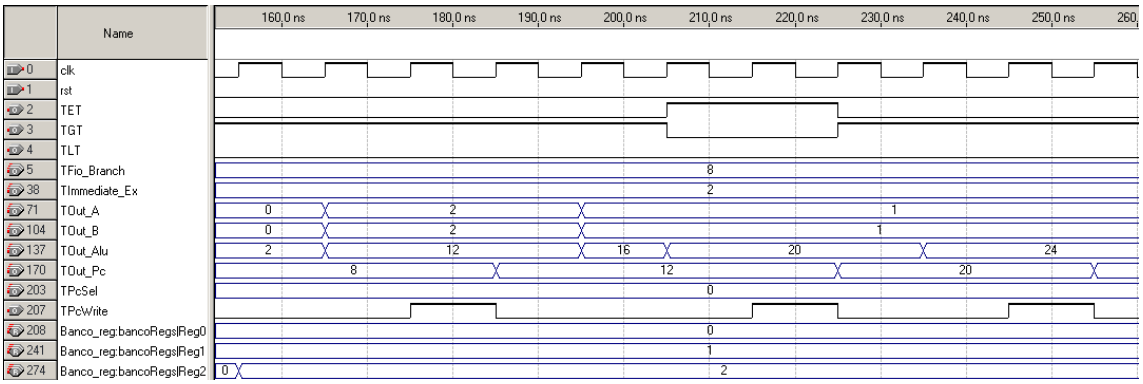
Descrição da simulação: No estado Div o controle aciona o sinal DivStart para que o componente Div inicie a operação de divisão com os operandos carregados nos registradores A e B (o que já foi feito no estado de decodificação). O algoritmo de divisão é executado durante 32 ciclos de clock, e ao fim desse tempo o componente Div ativa o sinal ParaDiv, avisando ao controle que a operação foi finalizada. O resto fica no registrador hi (interno do DIV)e o quociente no registrador lo (interno do DIV). Quando DivSel = 0, no Out_Div sai o resto (hi) e quando DivSel = 1, no Out_Div sai o quociente (lo). Note que 20 / 10 tem quociente 2 e resto 0. No caso da exceção, se ocorreu, ele prossegue com a rotina de tratamento.

Instrução mfhi e mflo



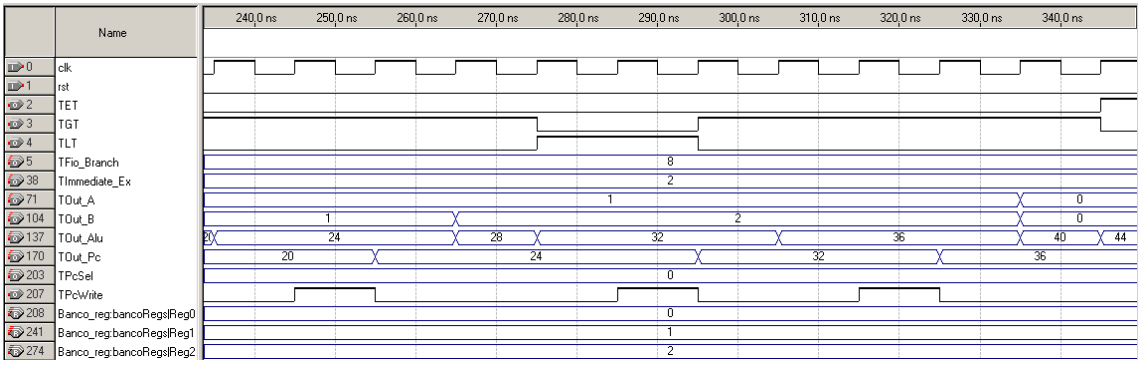
Descrição da simulação: As instruções ocorrem em dois estados, cada. No primeiro estado, é setado o DivSel (zero para resto e um para quociente, hi e lo, respectivamente). No próximo estado nós escrevemos no registrador (reg2 = quociente e reg3 = resto). Após isso, volta para o início com PC+4.

Instrução beq



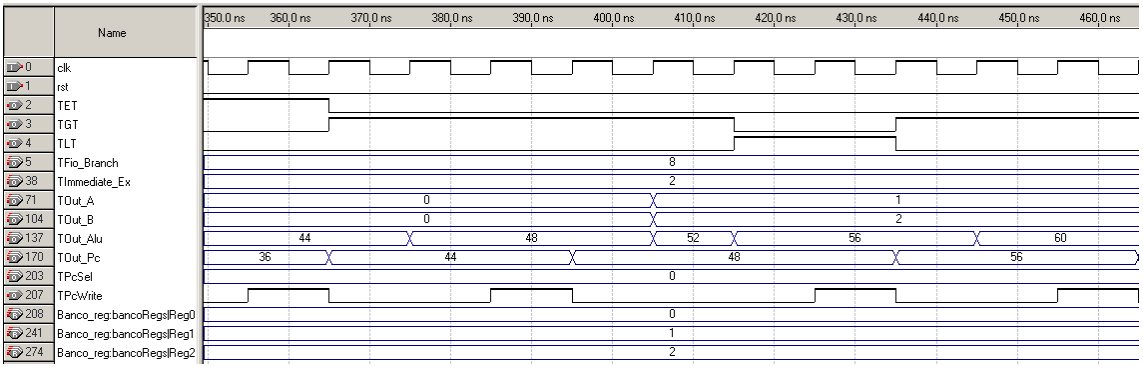
Descrição da simulação: Inicialmente, é carregado o valor 1 para o registrador 1. Após isso, esse mesmo valor é carregado tanto para o registrador A quanto para o registrador B. Na ALU é feita a comparação. Como A e B têm o mesmo valor, o sinal ET é ativado e é dado um PcWrite, para que o PC seja atualizado para o endereço do branch, anteriormente calculado. Se a condição não fosse satisfeita, não seria dado o PcWrite, e o programa continaria a execução normalmente, com o valor de PC+4 já no interior do PC.

Instrução bne



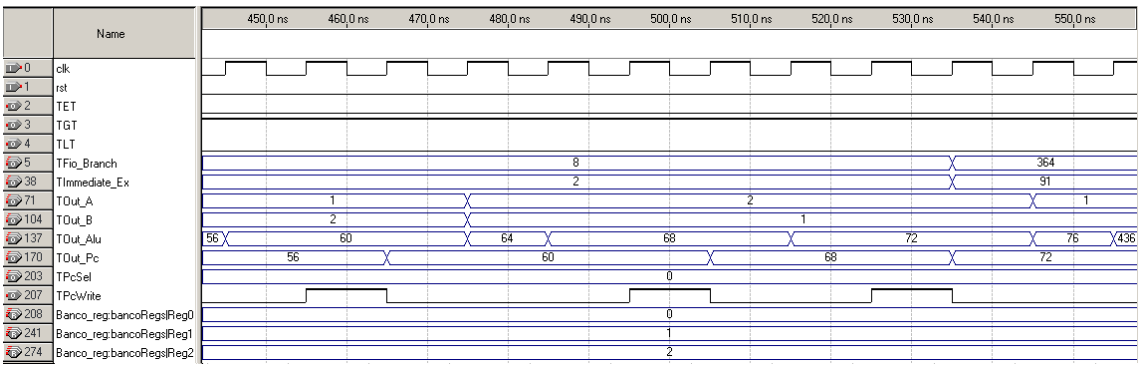
Descrição da simulação: Inicialmente, é carregado o valor 1 para o registrador 1 e 2 para o registrador 2. Esses valores são passados para os registradores A e B, respectivamente. Na ALU é feita a comparação. Como A e B têm valores diferentes, o sinal ET não é ativado, então é dado um PcWrite, para que o PC seja atualizado para o endereço do branch, anteriormente calculado. Se a condição não fosse satisfeita, não seria dado o PcWrite, e o programa continaria a execução normalmente, com o valor de PC+4 já no interior do PC.

Instrução ble



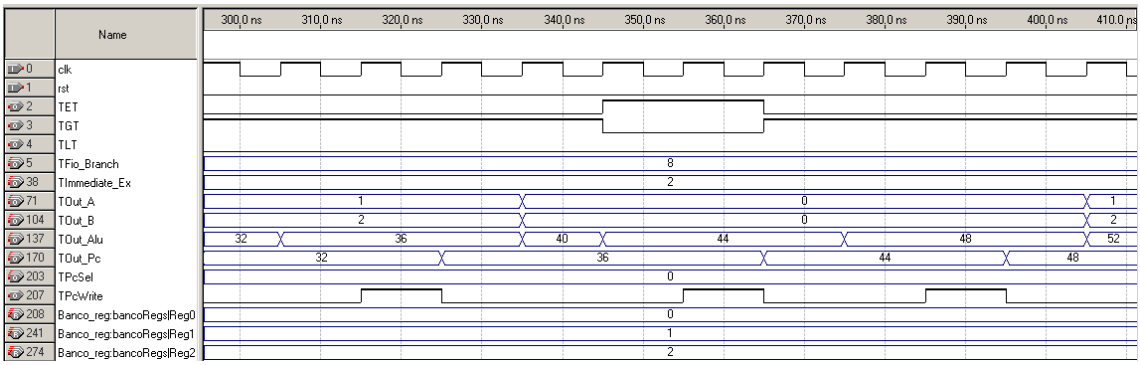
Descrição da simulação: Inicialmente, é carregado o valor 1 para o registrador 1 e 2 para o registrador 2. Esses valores são passados para os registradores A e B, respectivamente. Na ALU é feita a comparação. Como A é menor que B, o sinal LT é ativado, então é dado um PcWrite, para que o PC seja atualizado para o endereço do branch, anteriormente calculado. Se a condição não fosse satisfeita, não seria dado o PcWrite, e o programa continaria a execução normalmente, com o valor de PC+4 já no interior do PC.

Instrução bgt



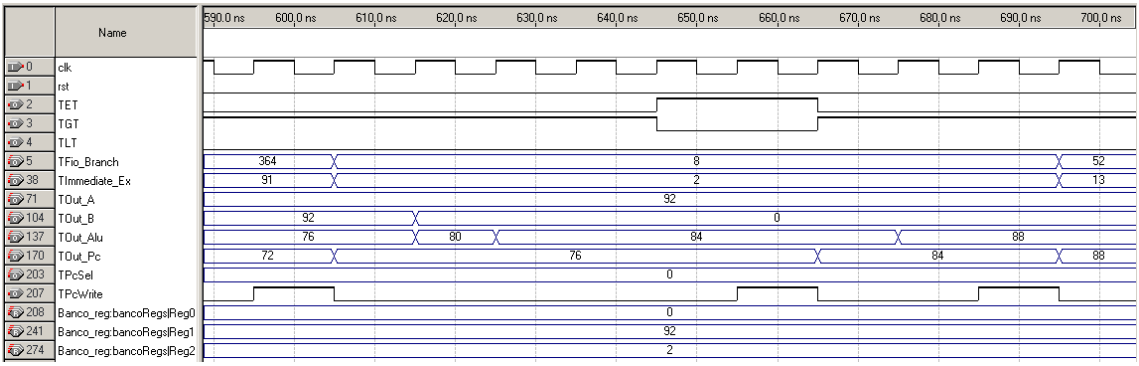
Descrição da simulação: Inicialmente, é carregado o valor 1 para o registrador 1 e 2 para o registrador 2. Esses valores são passados para os registradores B e A, respectivamente. Na ALU é feita a comparação. Como B é maior que A, o sinal GT é ativado, então é dado um PcWrite, para que o PC seja atualizado para o endereço do branch, anteriormente calculado. Se a condição não fosse satisfeita, não seria dado o PcWrite, e o programa continaria a execução normalmente, com o valor de PC+4 já no interior do PC.

Instrução bgez



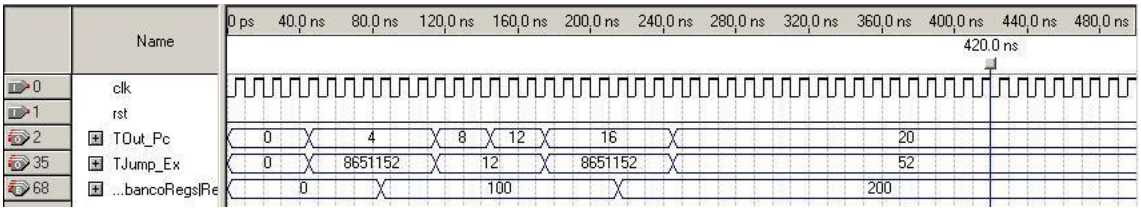
Descrição da simulação: No primeiro estado, o Branch, os valores de A e B já foram carregados do registrador 1 e do registrador 0 (seu valor é sempre zero). Na ALU, é feita uma comparação. Como Out_A não é menor que Out_B (A >= 0), o sinal LT (less than) fica zerado, e o próximo estado é o Escreve_Branch. Lembrando que o valor do pc do branch já foi calculado anteriormente, no estado Decodifica. No Escreve_Branch, PC é atualizado com o valor calculado do branch, e o estado Inicio (com o PC novo) é acionado.

Instrução beqm



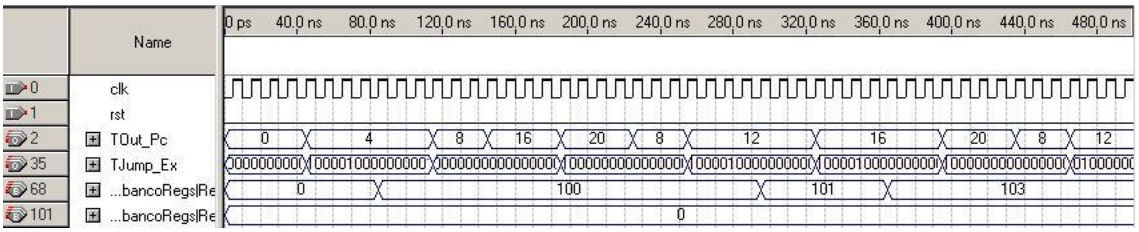
Descrição da simulação: Inicialmente, é carregado o valor 92 (endereço de memória que contém o valor zero) para o registrador 1. O endereço que está contido no registrador 1 é enviado à memória e espera-se 1 ciclo de clock para sair o valor. O valor vindo da memória e o valor do registrador 0 (sempre zero) são carregados na ALU, onde é feita a comparação. Como eles são iguais (os dois são 0), o sinal ET é ativado, então é dado um PcWrite, para que o PC seja atualizado para o endereço do branch, anteriormente calculado. Se a condição não fosse satisfeita, não seria dado o PcWrite, e o programa continaria a execução normalmente, com o valor de PC+4 já no interior do PC.

Instrução jump



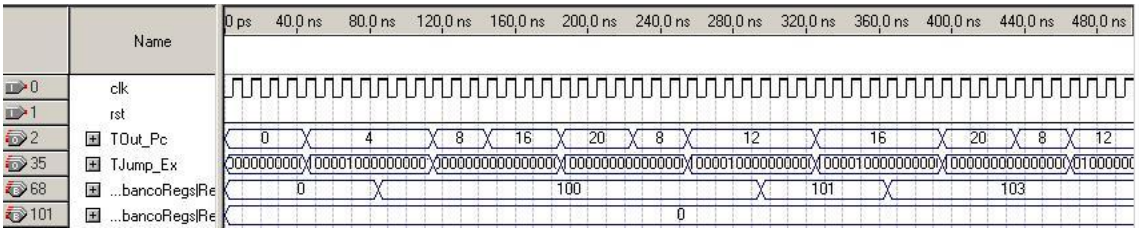
Descrição da simulação: No único estado, o Jump (00011111), o Jump_Ex (que já está estendido, multiplicado por 4 e concatenado com PC) passa do mux do PC para o PC, que é atualizado com o novo valor do desvio, no caso 16*4 = 64.

Instrução jal



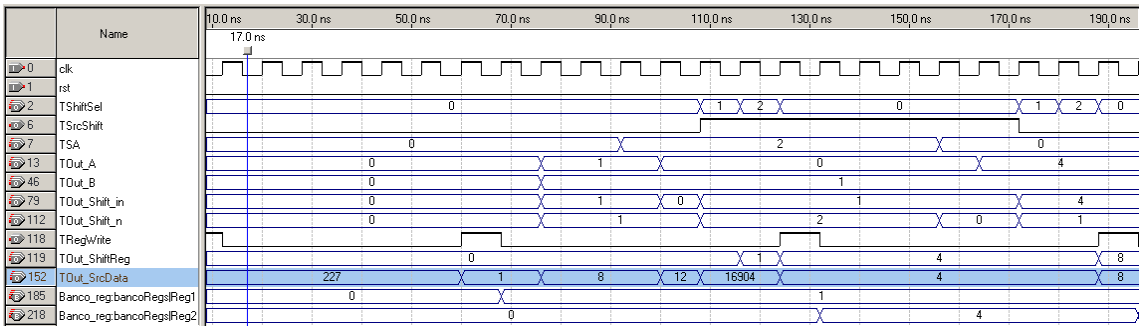
Descrição da simulação: No único estado, o Jal (00011111), o Jump_Ex (que já está estendido, multiplicado por 4 e concatenado com PC) passa do mux do PC para o PC, que é atualizado com o novo valor do desvio, no caso 4*4 = 16.

Instrução jr



Descrição da simulação: No único estado, o Jump register (00100000), o valor de reg0 já foi carregado em Out_A. Logo, o Out_A é passado como novo valor de PC.

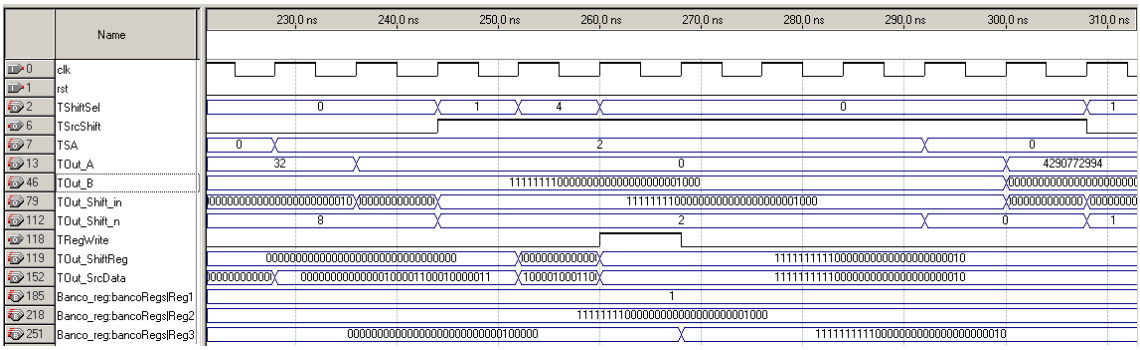
Instrução sll



Descrição da simulação: Na simulação acima, nós estamos guardando no registrador 2, o valor do registrador 1 << 2. A quantidade de bits a serem deslocados está no SA (shamt). Quando o Out_ShiftReg muda, o Out_SrcData também muda, e esse será o

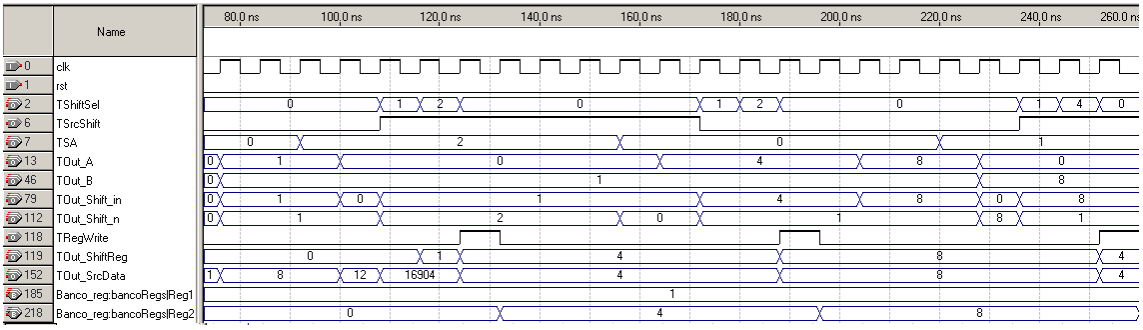
valor a ser salvo no registrador 2 quando RegWrite = 1. O SrcShift = 1 significa que será algumas das instruções de shift que usam o B e o shamt no IN e N, respectivamente.

Instrução sra



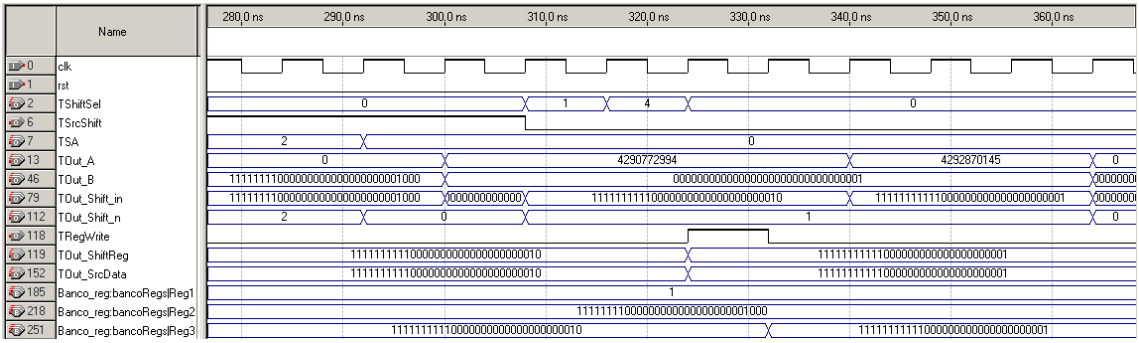
Descrição da simulação: Na simulação acima, nós estamos guardando no registrador 3, o valor do registrador 2 >>> 2 (shift right aritmético de 2 posições). A quantidade de bits a serem deslocados está no SA (shamt). Quando o Out_ShiftReg muda, o Out_SrcData também muda, e esse será o valor a ser salvo no registrador 3 quando RegWrite = 1. O SrcShift = 1 significa que será algumas das instruções de shift que usam o B e o shamt no IN e N, respectivamente. Perceba que o shift, no caso do sra, preserva o sinal.

Instrução sllv



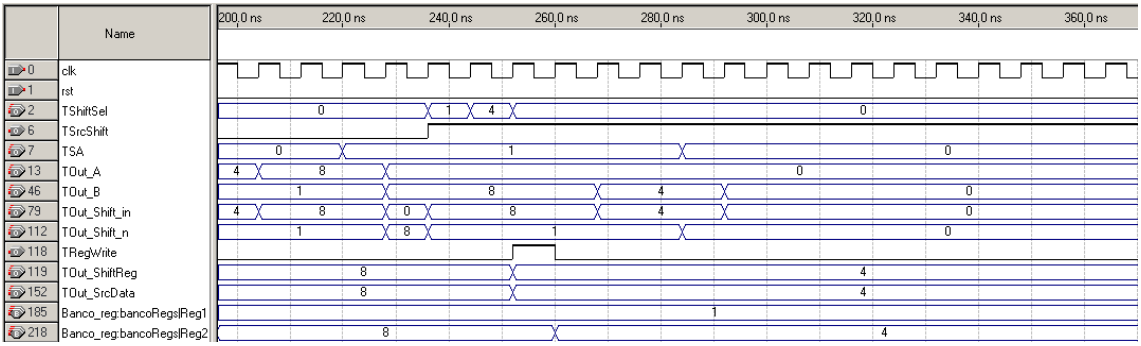
Descrição da simulação: Na simulação acima, nós estamos guardando no registrador 2, o valor do registrador 2 << 1. A quantidade de bits a serem deslocados está nos 5 bits menos significativos de Out_B. Quando o Out_ShiftReg muda, o Out_SrcData também muda, e esse será o valor a ser salvo no registrador 2 quando RegWrite = 1. O SrcShift = 0 significa que será algumas das instruções de shift que usam o A e B[4:0] no IN e N, respectivamente.

Instrução srav



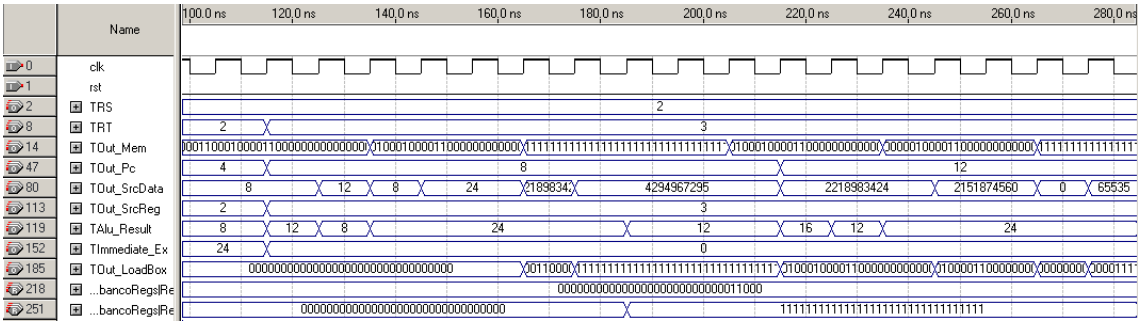
Descrição da simulação: Na simulação acima, nós estamos guardando no registrador 3, o valor do registrador 2 >>> 1 (shift right aritmético de 1 posição). A quantidade de bits a serem deslocados está nos 5 bits menos significativos de Out_B. Quando o Out_ShiftReg muda, o Out_SrcData também muda, e esse será o valor a ser salvo no registrador 3 quando RegWrite = 1. O SrcShift = 0 significa que será algumas das instruções de shift que usam o A e B[4:0] no IN e N, respectivamente.

Instrução srl



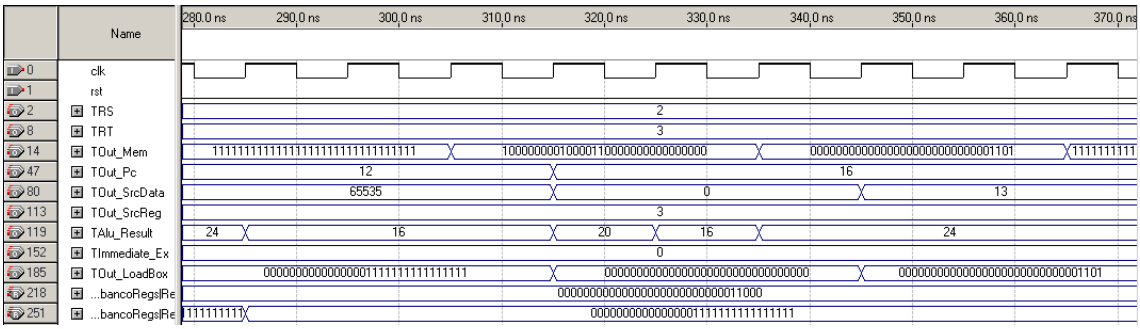
Descrição da simulação: Na simulação acima, nós estamos guardando no registrador 2, o valor do registrador 1 >> 2. A quantidade de bits a serem deslocados está no SA (shamt). Quando o Out_ShiftReg muda, o Out_SrcData também muda, e esse será o valor a ser salvo no registrador 2 quando RegWrite = 1. O SrcShift = 1 significa que será algumas das instruções de shift que usam o B e o shamt no IN e N, respectivamente.

Instrução lw



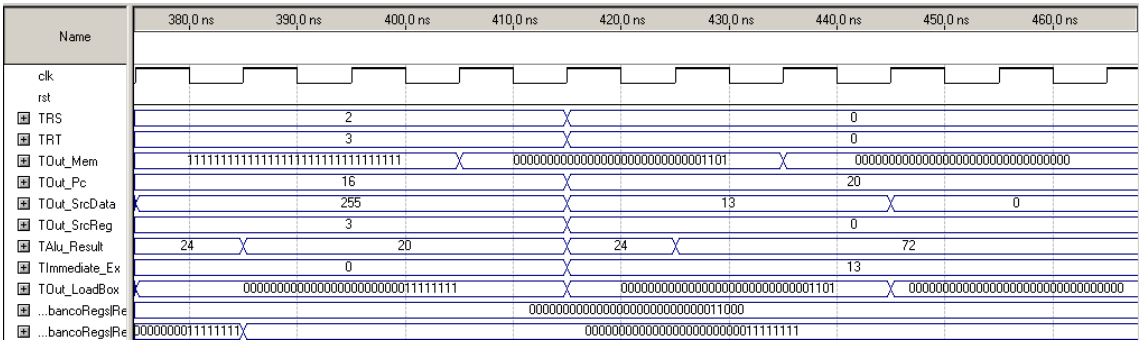
Descrição da simulação: No estado lw, a ALU soma o valor do registrador A, já carregado, com o offset, já estendido. Após isso o controle solicita a leitura do endereço de memória calculado, que se encontra no Alu_Result . Para a leitura, é necessário um estado de wait, em que o controle espera 1 ciclo de clock. Após isso, a saída da memória (Out_Mem) entra na caixa LoadBox, onde é reenviado para o banco de registradores e salvo no registrador escolhido(valor do rt). Então, o controle volta ao início com PC + 4.

Instrução lh



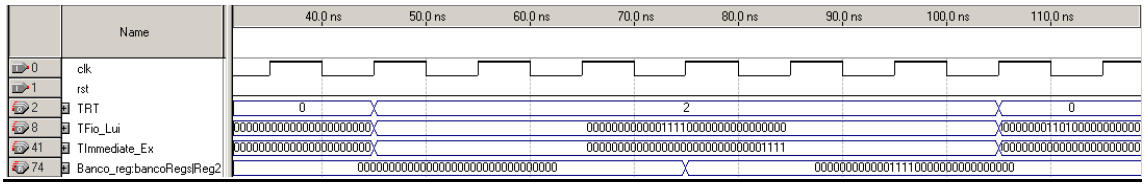
Descrição da simulação: No estado lh, a ALU soma o valor do registrador A, já carregado, com o offset, já estendido. Após isso o controle solicita a leitura do endereço de memória calculado, que se encontra no Alu_Result . Para a leitura, é necessário um estado de wait, em que o controle espera 1 ciclo de clock. Após isso, a saída da memória (Out_Mem) entra na caixa LoadBox, onde é processado (substitui os 16 bits mais significativos do endereço por 16 zeros) e enviado para o banco de registradores e salvo no registrador escolhido(valor do rt). Então, o controle volta ao início com PC + 4.

Instrução lb



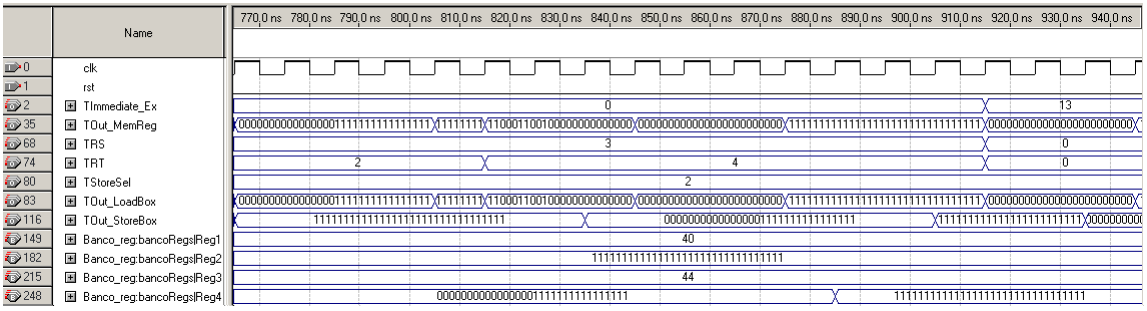
Descrição da simulação: No estado lb, a ALU soma o valor do registrador A, já carregado, com o offset, já estendido. Após isso o controle solicita a leitura do endereço de memória calculado, que se encontra no Alu_Result . Para a leitura, é necessário um estado de wait, em que o controle espera 1 ciclo de clock. Após isso, a saída da memória (Out_Mem) entra na caixa LoadBox, onde é processado (substitui os 24 bits mais significativos do endereço por 24 zeros) e enviado para o banco de registradores e salvo no registrador escolhido (valor do rt). Então, o controle volta ao início com PC + 4.

Instrução lui



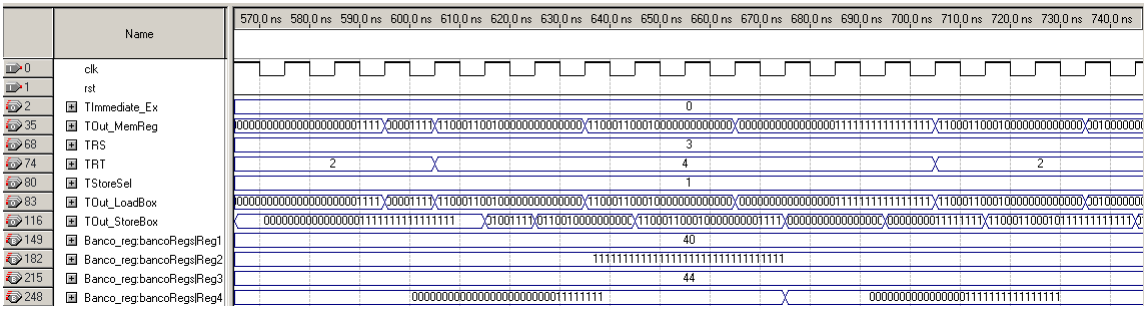
Descrição da Simulação: Depois de decodificar o opcode da instrução lui, o próximo estado é o LUI (00100011), onde o valor em do campo imediato sofre um shift left 16, como pode ser visto no Immediate_Ex, e esse valor é salvo no registrador escolhido. Após isso, o controle volta ao início com PC + 4.

Instrução sw



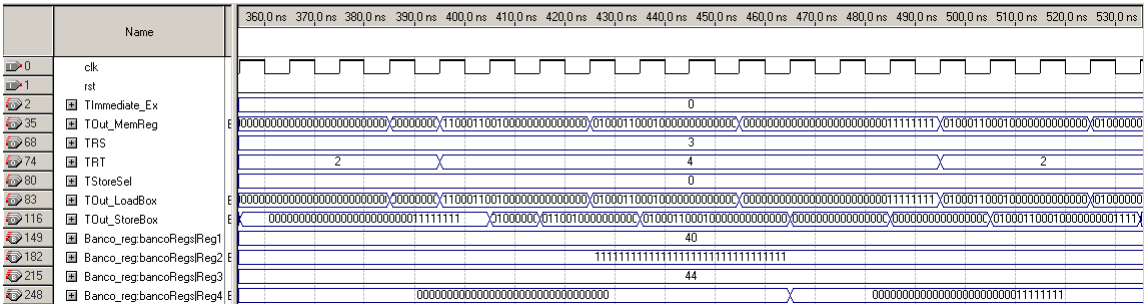
Descrição da simulação: Na instrução SW, o endereço de memória que será armazenado o valor do registrador é calculado somando o imediato ao endereço salvo no registrador RS passando o resultado para memória. A StoreBox então seleciona enviar todo o valor lido do registrador RT para escrever no endereço de memória calculado.

Instrução sh



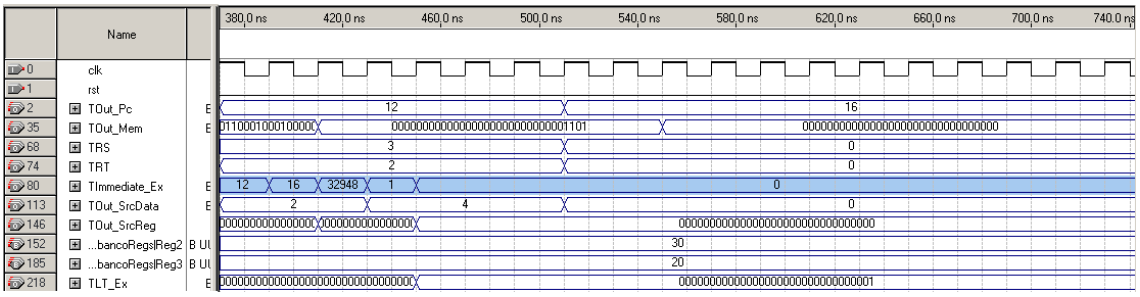
Descrição da simulação: Na instrução SH, o endereço de memória que será armazenado o valor do registrador é calculado somando o imediato ao endereço salvo no registrador RS passando o resultado para memória. Após a memória ler os dados fornecidos pelo endereço, estes dados vão para a StoreBox onde os seus dois bytes mais significativos são concatenados com os dois bytes menos significativos do valor lido do registrador RT e depois escrito no endereço de memória calculado.

Instrução sb



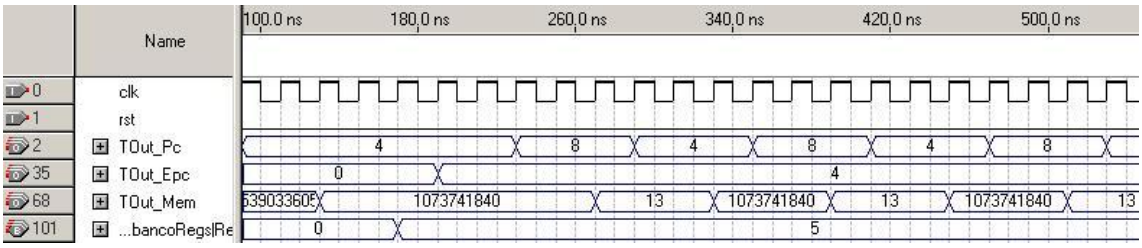
Descrição da simulação: Na instrução SB, o endereço de memória que será armazenado o valor do registrador é calculado somando o imediato ao endereço salvo no registrador RS passando o resultado para memória. Após a memória ler os dados fornecidos pelo endereço, estes dados vão para a StoreBox onde os seus três bytes mais significativos são concatenados com o byte menos significativo do valor lido do registrador RT e depois escrito no endereço de memória calculado.

Instrução break



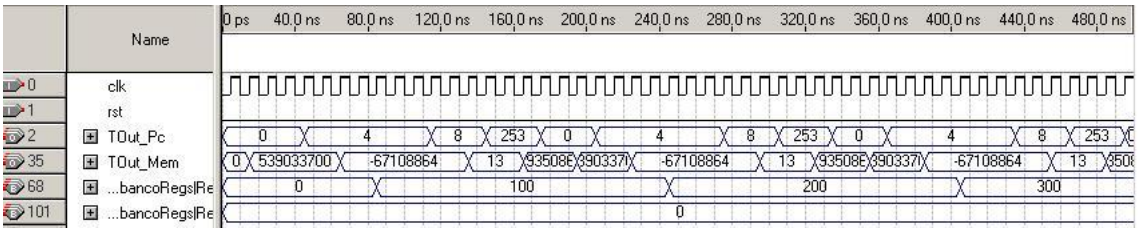
Descrição da simulação: No estado acima, Break, o controle mandará o sinal PcWrite = 0 e entrará num loop infinito em que o estadoAtual sempre voltará para o estado Break.

Instrução rte



Descrição da simulação: Na simulação acima, o EPC (que tem valor 4), é passado para o PC, que continua executando a operação do anterior (RTE), ficando em um loop infinito.

Opcode Inexistente



Descrição da Simulação: Quando o controle não conhece o Opcode/Funct, é enviado ao PC o endereço 253, a partir daí, a rotina de tratamento segue com a execução.

Conclusão

Por fim, podemos dizer que esse projeto nos ajudou muito a nível de conhecimento computacional por se tratar de um projeto desafiador que nos familiarizou um pouco com a arquitetura e organização de um processador e a sua implementação. Graças a essa disciplina conseguimos evoluir no curso ganhando mais experiência na execução de projetos e trabalho em grupo, em uma área que normalmente não iremos ver muito ao longo do curso.

Sem duvidas de que todo esse conhecimento adquirido foi massificado de uma forma construtiva podemos dizer que estamos mais preparados com as coisas que iremos enfrentar mais a frente. Com certeza estamos entendendo melhor o funcionamento de um processador multiciclo tendo uma visão mais abrangente no que tange a arquitetura de hardware.

Foi interessante a busca e a insistência do grupo em alcançar os objetivos do projeto anteriormente especificados, nos fazendo ficar horas pensando sobre a melhor forma de se arquitetar ou implementar o projeto. Acreditamos ter alcançado tais objetivos visando a conclusão da matéria com ganhos realmente significativos.