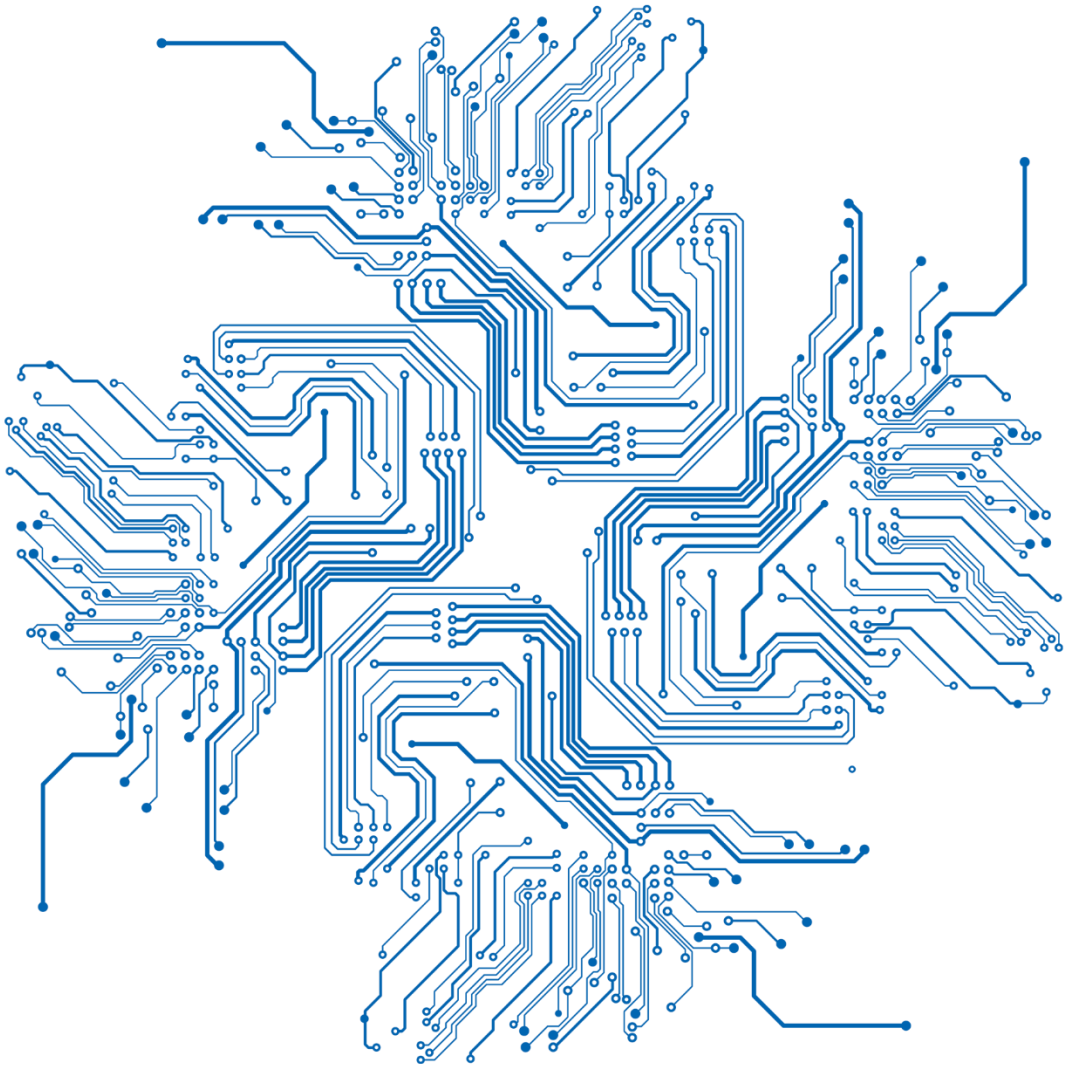
NRisc Pipeline Processor

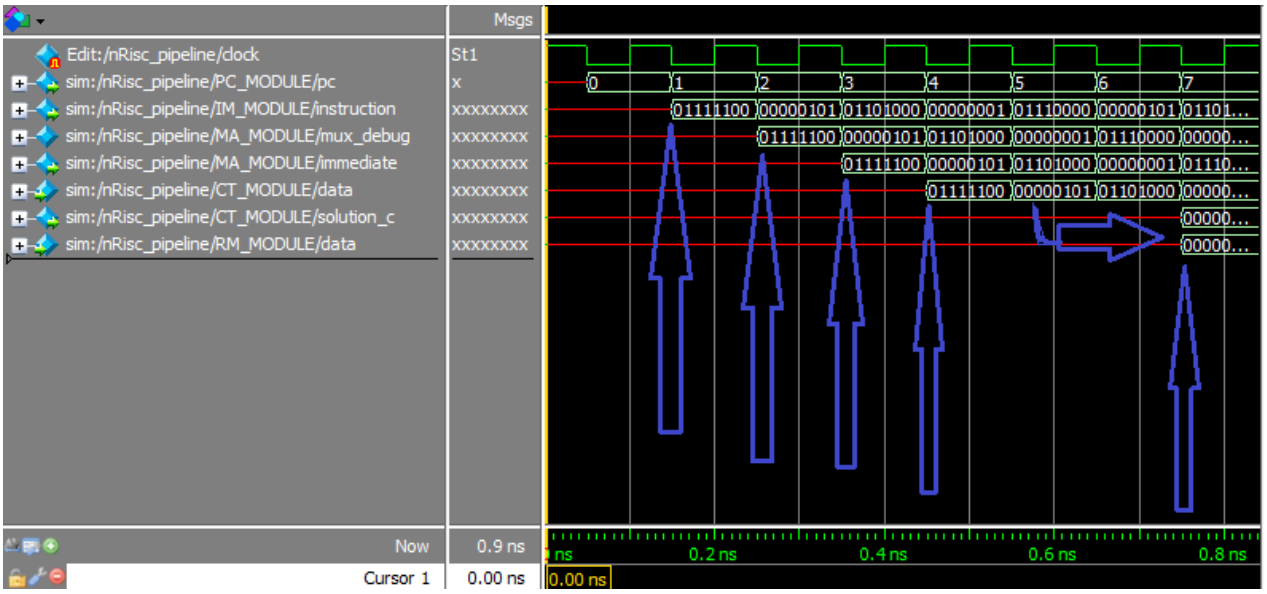
Aluno: Tarcísio Batista Prates



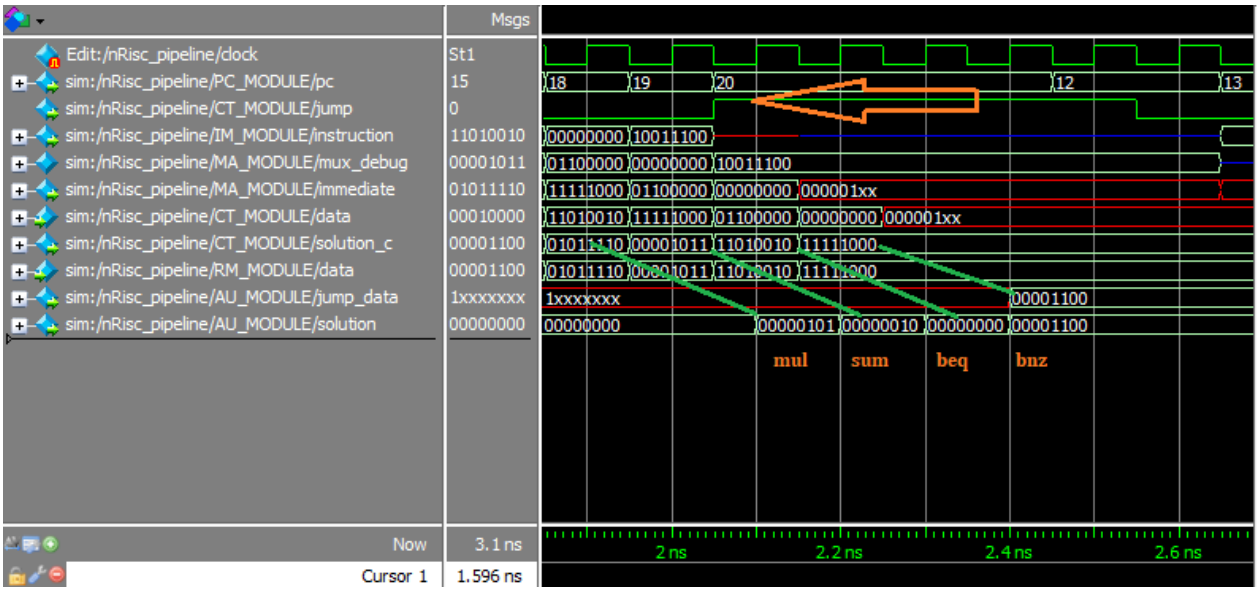
nRisc Pipeline

Para projetar o nRisc pipeline, foi adotado como plataforma de desenvolvimento o nRisc uniciclo apresentado na última aula. As semelhanças se estendem para o formato e tipo de instruções executadas, preservando até mesmo o formato multiciclo da instrução load_immedite.

O primeiro passo foi reajustar a sincronia dos módulos, de tal forma que operassem todos a partir do mesmo sinal de clock, feito isso, foi adicionado nos módulos ***Mux, Controler e Register*** memórias de transição, que permitem que enquanto uma instrução esteja nos registradores de saída, a próxima instrução já esteja pronta para ser executada no módulo, aguardando apenas o próximo clock para tal, isso permite que as instruções sejam encadeais, e os módulos nunca esteja ociosos, melhorando a performance do processador. Para a instrução de jump, sem sombra de dúvidas a mais complexa, já que até que a instrução de jump seja identificada no módulo der controle, outras instruções já ocupam os módulos anteriores, então a partir do Controlador, um sinal de controle Jump é enviado para ***Program_counter***, parando a contagem, ***Instruction_memory e Mux*** de forma a invalidar as suas saídas, assim apesar de o fluxo de instruções já carregadas continuar, as saídas inválidas geradas não executam quaisquer instrução nos módulos seguintes, assim quando a ALU executar a comparação que resulta na resposta para o Jump, o contador volta a operar normalmente a partir do valor recebido da ALU, além disso, informa para o **Controler,** através de um sinal nominado Jumped, que o jump foi realizado, então o controle libera os módulos bloqueados para receberem novas instruções.
Usualmente, as memórias intermediárias não ficam dentro dos módulos, são estruturas a parte, no entanto, devido ao curto período de prazo e complexidade do projeto, não foi possível aplicar esse modelo. **Ainda assim, o processador se comporta como um modelo pipeline e a execução ocorre sem erros, apresentando os resultados esperados.**



Na imagem acima, é possível verificar que à medida que o contador avança, os módulos são preenchidos com instruções e as mesmas também avançam no caminho de decodificação da instrução e execução. Nas duas últimas linhas, há um espaço de 3 ciclos de clock, isso ocorre, pois, a instrução que está sendo executada é do tipo load_immedite, que consome 2 ciclos para receber os dados e há um ciclo adicional que é executado no controlador, para organizar os dados e disponibilizá-los no momento certo para o banco de registradores.

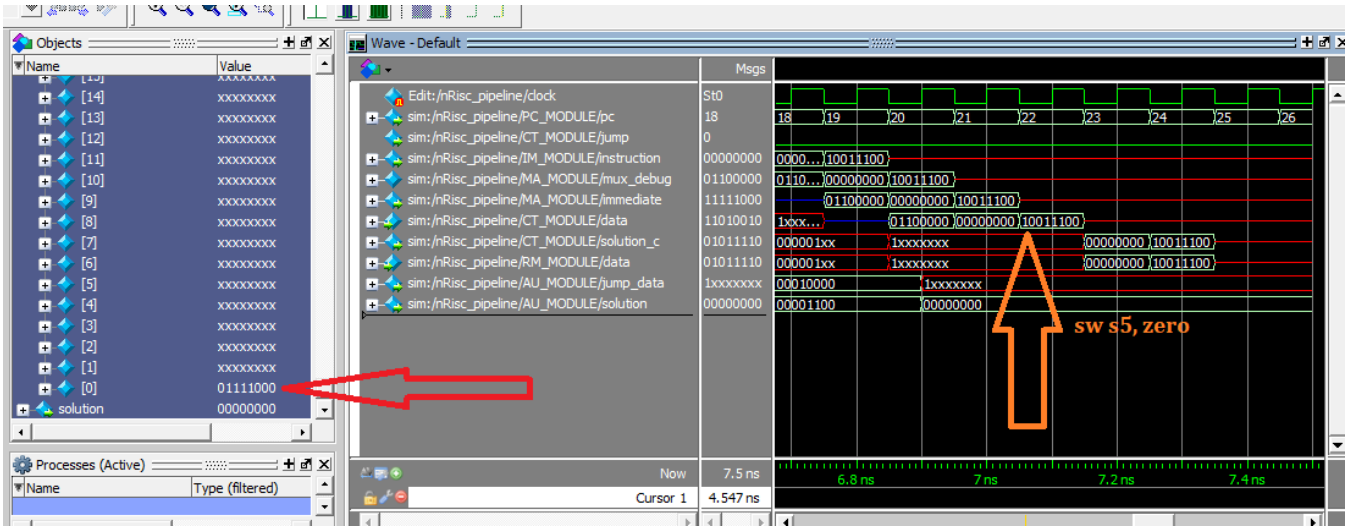


Nessa imagem, mostrada acima, a seta em laranja indica o momento em que o controlador identifica uma operação de jump prestes a ser executada. Observe que em seguida a instrução de número 20 não entra na fila, pois o sinal de controle jump invalida todas as saídas da memória de instrução e mux , fazendo com que entrem na fila instruções com valores que não estimulam o processador.

Na última linha, verifica-se as operações aritméticas sendo executadas e por último o valor do jump sendo decidido, assim que disponível o contador já está apto a executar o jump, e assim o faz.

É importante ressaltar que o valor do contador não reflete a instrução que está sendo executada, já que o formato pipeline faz com que as próximas instruções após a leitura do jump já estejam dentro do processador, assim, independentemente da instrução BEQ ser verdadeira ou não, as saídas serão anuladas e será preciso recarregar as instruções que foram descartadas, tornando a operação de Jump extremamente custosa para a execução.

O processo se repete até que a instrução de comparação seja verdadeira e o programa seja finalizado.



O programa executado foi o fatorial de 5, que é igual a 120, e binário a representação é **01111000**, exatamente o que foi gravado na memória. O programa executa 6/8 das (75%) operações possíveis, no entanto há um arquivo na raiz do projeto, contendo um teste com todas as operações.
Fim da execução. :)

nRisc Instruction

000	000	00
OPERATION	**RESGISTER_0**	**REGISTER_1**

A instrução adotada para a execução de tarefas no processador ***nRisc***, possui 8 bits de largura, sendo os três primeiros dedicados à operação a ser executada, os três seguintes são dedicados ao primeiro registrador da operação e os dois últimos ao segundo registrador. Sendo assim, o usuário dispõe de oito tipos operações diferentes e oito registradores para serem usados nas operações. De modo geral a arquitetura das operações segue o modelo descrito acima, no entanto, a operação de imediato, utiliza duas instruções para executar a operação, mais detalhes na descrição completa abaixo. Além disso, como os registradores possuem três bits, o segundo registrador recebe um bit zero à esquerda, proporcionado por um extensor de sinal.

Code operations table

Nº	CODE			OP	DESCRIPTION
0	**0**	**0**	**0**	ADD	Soma dois registradores
1	**0**	**0**	**1**	SUB	Subtrai dois registradores
2	**0**	**1**	**0**	MUL	Multiplica dois registradores
3	**0**	**1**	**1**	LI	Carrega um imediato em um registrador
4	**1**	**0**	**0**	SW	Armazena o valor de um registrador na memória
5	**1**	**0**	**1**	LW	Carrega um valor da memória em um registrador
6	**1**	**1**	**0**	BEQ	Compara igualdade entre dois registradores
7	**1**	**1**	**1**	BNZ	Faz um jump com base no resultado de BEQ

Register code table

Nº	CODE			REGISTER
0	**0**	**0**	**0**	ZERO
1	**0**	**0**	**1**	R_BEQ
2	**0**	**1**	**0**	S0
3	**0**	**1**	**1**	S1
4	**1**	**0**	**0**	S2
5	**1**	**0**	**1**	S3
6	**1**	**1**	**0**	S4

7	**1**	**1**	**1**	S5

Instructions examples

Operação de soma [ADD]

```
add s3, s1    //Estrutura
s3 = s3 + s1 //Interpretação no processador
000 101 11    //Representação em binário
```

Operação de subtração [SUB]

```
sub s3, s1    //Estrutura
s3 = s3 - s1 //Interpretação no processador
001 101 11    //Representação em binário
```

Operação de multiplicação [MUL]

```
mul s3, s1    //Estrutura
s3 = s3 * s1 //Interpretação no processador
010 101 11    //Representação em binário
```

Operação carregamento de imediato [LI]

```
li s3          //Primeiro o reg. de destino
10             //Valor a ser gravado
MEMORY[s3] = 10       //Interpretação no
processador
011 101 00       //Representação em binário
00001010      //valor
//A instrução é composta de duas partes, assim
//é possível gravar valores mais altos no reg.
```

Operação de store word [SW]

```
sw s3, s1         //Estrutura
```

```
MEMORY[s1] = s3 //Interpretação no processador
100 101 11      //Representação em binário
//Grava o valor de um registrador na memória
```

Operação de load word [LW]

```
lw s3, s1       //Estrutura
S3 = MEMORY[s1] //Interpretação no processador
101 101 11      //Representação em binário
//Carrega um valor da memória no registrador
//informado
```

Operação de comparação de registradores [BEQ]

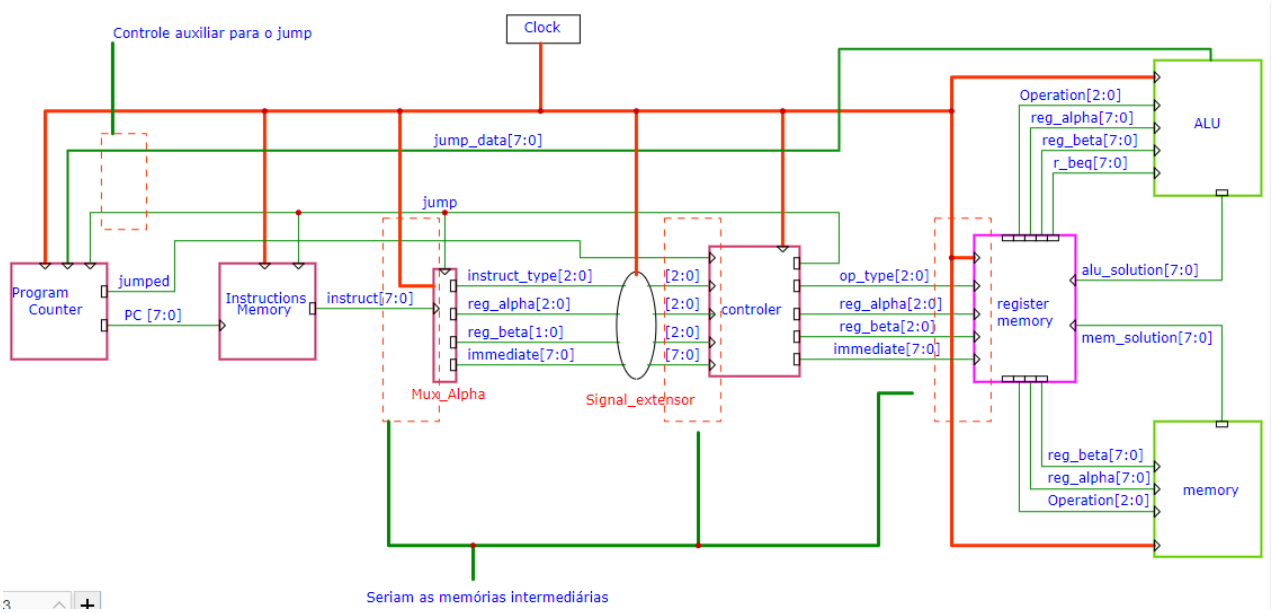
```
beq s3, s1          //Estrutura
R_BEQ = (s3 == s1) //Interpretação
110 101 11          //Representação em binário
//Compara os dois registradores informados,
//sendo o valor gravado no registrador R_BEQ,
//onde 1 para verdadeiro e 0 para falso.
```

Operação de comparação de registradores [BNZ]

```
bnz s3, s1                  //Estrutura
pc = (R_BEQ == 0)? S3 : s1 //Interpretação
111 101 11      //Representação em binário
//Compara o resultado da instrução BEQ e com
//base nele faz um jump
```

Datapath nRisc

Arquivo com detalhes na pasta raiz



Os blocos pontilhados representam os módulos que estavam no projeto, mas não foi possível implementá-los devido ao curto prazo para a complexidade do problema. Os mais alongados representam as memórias intermediárias, que por ocasião estão inseridas dentro dos módulos ao longo do caminho de dados, já o menor, logo acima do program_counter, é um bloco que assume a responsabilidade de receber os dados de jump antes de executá-lo, tirando a responsabilidade que atualmente se encontra dentro do módulo program_counter.

Control signals:

N°	SIGNAL	REG NAME	FUNCTION
0	1	Li_instruction	Habilita operação multiciclo para receber um imediato e posteriormente gravá-lo no registrador
1	1	await_solutions	Sempre que uma operação for executada e que o resultado precisa ser gravado no banco de registradores, o sinal é habilitado, colocado o módulo dos registradores em função de gravação, pois assim que o resultado, seja aritmético da ULA ou de leitura de memória retornar o dado, é imediatamente gravado na posição anteriormente definida pela instrução.
2	0 ou 1	clock	Os módulos Counter, Instruction Memory, Mux, Control e Register memory são sensíveis à borda de descida do clock, já os módulos ALU e Memory são sensíveis à borda de subida, isso garante que a resposta das operações feitas na ALU e Memória, estejam prontamente disponíveis antes da leitura de dados da próxima instrução.
3	1	jump	Habilita a inserção de um valor arbitrário na contagem de PC
4	[2:0]	operation	Recebe a instrução da word
5	1	jumped	Assim que uma operação de jump é executada, o sinal informa ao controle que imediatamente libera os demais módulos para receber novas instruções.