Características do Projeto do µP: Descrição e Explicações (versão beta)

18 de setembro de 2025

Sumário

Aqui estão as regras para interpretar o sorteio dos itens do seu microprocessador. Note que cada seção deste documento pode se relacionar com Labs VHDL diferentes (tabela 1).

A entrega da ULA, em especial, não precisa de *todo o conteúdo* das seções: as condicionais servem só para determinar quais as flags devem ser

Seção / Labs	ULA	Regs	UnCtrle	Calc	Conds	RAM	Valid
Obrigatório	X	X		X	X	X	X
Acumulador		X		X	X	X	X
Carga de Ctes		X		X			X
Ops de ULA	X				X		X
Saltos e Cond	X			X	X	X	X
Custom Valid	X		X	X	X	X	X
Checklist							

Tabela 1: Mapa cruzado das seções deste documento com os laboratórios VHDL

implementadas, e a customização da validação é apenas para adiantar serviço (recomendado mas não cobrado).

1 Obrigatório

O acumulador, caso exista, será identificado por A; os i registradores do banco de registradores são identificados por $R0, R1, ..., Ri$, sendo especificados como Rn ou Rm em uma instrução. Se houver um segundo acumulador, ele será B.

O processador não deverá executar *opcodes fora do que foi explicitamente pedido* (a não ser que os alunos peçam autorização ao professor). Isto significa que não pode existir um código binário que faça uma instrução "extra" executar.

Para ser bastante claro: se o seu processador só faz somas com dois registradores (eg., `ADD R1,R2` que realiza a operação $R1 \leftarrow R1 + R2$), então não pode existir nele um código binário que executa uma instrução de soma com três registradores (algo como `ADD R1,R2,R3` que faria $R1 \leftarrow R1 + R2$).

Tanto os opcodes quanto a estrutura interna dos campos de bits são livres para escolha de vocês.

2 Acumulador

Há duas possibilidades de sorteio:

- **ULA com instruções ortogonais:** segue o modelo RISC da ULA, com as operações possíveis para qualquer um dos registradores;
- **ULA com acumulador:** neste caso, obrigatoriamente uma das fontes de dados para a ULA deverá ser o acumulador e o destino do resultado também deve ser o acumulador;

- **ULA com dois acumuladores:** como no anterior, obrigatoriamente uma das fontes de dados deverá ser um dos acumuladores, que também é o destino do resultado.

Para implementação, o circuito com acumulador está descrito no *laboratório #3* (banco de registradores e ULA). O acumulador (ou os acumuladores) deverá obrigatoriamente ser um registrador instanciado à parte do banco de registradores.

No caso de uso de acumulador, instruções `MOV A,Rn` (que faz $A \leftarrow Rn$ para algum Rn do banco de registradores) e `MOV Rn,A` (análoga) deverão ser implementadas, mas instruções `MOV Rn,Rm` (direto entre dois registradores) estão proibidas. Se houver dois acumuladores, também serão necessários `MOV B,Rn` e `MOV Rn,B`, obviamente.

No caso ortogonal, deverá ser implementada uma instrução `MOV Rn,Rm`, que faz $Rn \leftarrow Rm$ para Rn e Rm pertencentes ao banco.

3 Carga de constantes

As constantes deverão ser sinalizadas e estar em complemento de 2, exigindo portanto extensão de sinal no circuito. As instruções deverão ser especificadas e implementadas no *laboratório #5* (calculadora programável), com um dos casos seguintes, de acordo com o sorteio:

- Instrução LD exclusiva de carga (por ex., `LD R5,131`), caso em que a constante deverá ser conduzida diretamente da instrução para o banco de registradores ou acumulador;
- Constantes gravadas por soma com registrador zero e instrução ADDI com três operandos (por ex., `ADDI R6,R0,131`), caso em que o registrador R0 deve ser fixo com o valor zero;
- Constantes gravadas por soma com o próprio registrador (zerado) usando ADDI com apenas dois operandos (por ex., `CLR R6; ADDI R6,131`), caso em que uma instrução `CLR Rn`, que zera o registrador especificado, deve ser implementada também.

4 Operações de ULA

Note que, além das operações abaixo, a ULA deverá realizar as instruções exigidas pelo seu sorteio das complicações da validação e determinação do final de loop, na seção 6.

4.1 Acumulador / Ortogonal

Para **ULA com acumulador,** as operações de ULA (soma, subtração, comparação e demais) são obrigatoriamente com apenas dois operandos, como p.ex. `SUB A,R5` que realiza a operação $A \leftarrow A - R5$), e o primeiro operando é obrigatoriamente o acumulador.

Para **ULA ortogonal,** podem ser sorteados 2 ou 3 operandos. No caso de 2, o primeiro operando é tanto fonte quanto destino, e o segundo operando é uma outra fonte, como em `SUB R3,R6` que realiza $R3 \leftarrow R3 - R6$); no caso de 3, a fonte e os dois destinos são independentes, como em `ADD R7,R2,R4` que faz $R7 \leftarrow R2 + R4$.

4.2 Constante

Caso uma ou mais operações da ULA permitam **constantes** (p.ex., `CMPI R3,25`, que compara $R3$ com o número 25), estas constantes imediatas devem estar *especificadas dentro dos bits da instrução* e obrigatoriamente devem ser expressas em complemento de 2 e utilizar extensão de sinal antes da operação em si.

4.3 Subtração e Comparacão

A subtração pode ser feita com ou sem **borrow** (em português, "empresta um"), ou seja, com um bit adicional (que é o Cf, *Carry Flag*) que vai ser subtraído junto. Por exemplo, `SUBB R1,R2` irá realizar $R1 \leftarrow R1 - R2 - Cf$. O propósito disto é fazer operações com mais bits do que um registrador tem disponível, cascateando subtrações.

Caso haja uma instrução de **comparação** presente, ela realiza uma comparação subtraindo os dois operandos e alterando as flags de acordo, sem gravar o resultado da subtração em nenhum lugar. Se a instrução for `CMP` ela compara dois registradores; se a instrução for `CMPI`), a comparação é feita com uma constante imediata.

4.4 Flags

Flags são sinalizadores que indicam que algo de interesse ocorreu. O exemplo mais simples é a flag de carry, que indica o "vai-um" de soma ou subtração. Iremos usar apenas flags de operações de ULA. O processador RISC-V é uma das poucas exceções que *não utiliza* flags, tratando estas situações de outra forma.

As flags mais comuns e úteis, que existem em quase todos os processadores, são:

- **Carry,** que indica se houve estouro em uma operação não sinalizada na ULA (ou seja, só com números positivos e zero);

- **Overflow,** que indica se houve estouro em uma operação sinalizada na ULA (ou seja, que inclui número negativos);
- **Zero,** que indica que o resultado da operação mais recente da ULA foi zero;
- **Sinal (ou Negativo),** que indica o sinal do resultado da operação mais recente da ULA (ou o sinal do acumulador, depende!), sendo apenas uma cópia do MSB.

As flags que devem ser obrigatoriamente implementadas são apenas aquelas usadas pelas instruções condicionais sorteadas para o processador, vistas na tabela 2 na seção 5.

Explicação do funcionamento

Suponha que temos um processador de 8 bits e vamos realizar uma instrução `ADD R1,R2,R3`. Os valores são $R2 = 11001000_2$ e $R3 = 01100100_2$; eles estão em binário porque *o processador em si não sabe se R2 e R3 são signed ou unsigned,* é o programador quem determina essa interpretação.

A operação é $200_{10} + 100_{10}$, ou em binário:

```
   11001000
 + 01100100
 ----------
(1) 00101100
```

O (1) entre parênteses indica o vai um na saída do circuito somador (lembre dos somadores completos cascateados); *este é o valor da carry flag.* Ele indica o estouro no caso em que consideramos R2 e R3 como *unsigned.* Dito de outra forma, o resultado exigiria um nono bit em 1 ; como os registradores têm 8 bits, ou seja, são feitos de 8 flip-flops, o resultado não cabe nele e Cf=1. O número deveria ser $(1)00101100_2 = 300_{10}$ mas o resultado gerado é $00101100_2 = 44_{10}$.

Se, por outro lado, formos considerar R2 e R3 como *signed,* percebemos que R2 é negativo, R3 é positivo e o resultado é positivo (basta olhar o bit MSB: se b7 é 1, o número é negativo em complemento de 2). Portanto não há estouro e *o valor da flag de overflow é 0.* Só há estouro de números *signed* se somarmos dois números que têm um mesmo sinal e se o resultado tiver o sinal contrário.

Nos casos de subtração o raciocínio é análogo.

A *zero flag* estará valendo zero depois da execução da instrução, pois o resultado da operação é diferente de zero.

A *flag de sinal ou a flag de negativo* vale zero, pois o resultado é positivo quando interpretado em complemento de 2. É apenas a cópia do bit MSB, o b7.

Mais explicações sobre carry e overflow podem ser vistas no PDF de Complemento de 2 no Moodle.

Armazenamento das flags

Obrigatório: o valor das flags deve ser armazenado em flip-flops individuais sempre que uma instrução aritmética for executada. A implementação correta destes flip-flops é parte apenas do *laboratório #6*, mas pode ser adiantada a critério da equipe.

Estes flip-flops *não são atualizados* nas instruções que não são de ULA. Alternativamente, as flags podem ser guardadas juntas em um registrador especial chamado PSW (*Processor Status Word*), vocês é que mandam.

Dito de outra forma, quando houver ADD, SUB,CMP ou similares, o valor indicado pela ULA será guardado, mas quando houver branches, MOV, NOP, LW, SW, o valor dos flip-flops fica inalterado. Portanto eles efetivamente *guardam o status da instrução de ULA mais recente.* [1]

Estes flip-flops ficam no top-level ou na UC, nunca dentro da ULA.

Como se usa no processador?

Veja abaixo um exemplo em assembly similar ao RISC-V (mas com flags) de como comparar um registrador de nome R3 com o valor 51:

O pseudocódigo é:

```
se R3>=51
  então R4 = 7
  senão R4 = -3
```

A implementação fica:

```
   cmpi R3,51        # ULA: R3-51
   bcc  menor        # Salta para "menor" se Cf=1
maior_ou_igual:
   mov  R4,7
   jmp  continua     # Salto incondicional
menor:
   mov  R4,-3
continua:
```

A parte principal é a comparação. A instrução `CMPI` vai mandar a ULA fazer a subtração `R3-51` (mas não vai gravar este resultado em lugar nenhum!). Se R3 estiver com um valor maior ou igual a 51, esta subtração

[1] Claro, em processadores reais há exceções, documentadas nos manuais. Por exemplo, incremento e decremento normalmente não atualizam as flags.

resulta em algo positivo ou zero, portanto não há estouro de operação *unsigned* e a flag de carry vai para zero.

No entanto, se R3 estiver com um valor menor do que 51, a conta gera um bit de "empresta-um," ou seja, um indicador de que o resultado foi negativo. Isso indica que houve um estouro *unsigned* e portanto neste caso a flag de carry vai ser ativada, ou seja, vai para um.

A instrução BCC vai simplesmente consultar o valor atual da flag de carry e saltar caso Cf=1. Se Cf=0, a execução avança para a instrução da linha seguinte. Isto foi projetado para ser usado em conjunto na implementação de condicionais.

Perceba também que ao final da execução do programa a Cf ainda vai estar com o valor determinado pela instrução `CMPI R3,51` do início, já que as outras instruções não são de ULA e não afetam o valor das flags.

5 Saltos e Condicionais

A implementação correta do valor das flags é parte do *laboratório #2 – ULA*. A implementação correta das instruções de desvio condicional é parte do *laboratório #6 – Condicionais*. Mais detalhes podem ser vistos na seção 4.4.

5.1 Cálculo do Destino

Observe que, no caso de desvios relativos, a constante do "delta" deverá estar **obrigatoriamente em complemento de 2** – não pode estar em sinal-magnitude.

A convenção que deve ser utilizada é: instruções `JMP` devem saltar para um endereço absoluto (ou seja, a constante especificada na instrução vai ser escrita no PC) e instruções `JR` ou branches `B` ou `Bxx` (onde xx é a condição, como BGT ou BMI) utilizam um endereço relativo (a constante vai ser somada ao PC, especificando um "delta" de endereços que devem ser saltados).

5.2 Saltos Condicionais

Os saltos condicionais possuem convenções diferentes em cada ISA de processador existente. Iremos utilizar o padrão das instruções dos processadores ARM, que trabalham com *flags,* ao invés do padrão do RISC-V, que faz a comparação na própria instrução de desvio. Consulte o material de flags para entender isso.

As únicas duas instruções de desvio condicional sorteadas para a equipe estão identificadas na tabela 2; a equipe não pode implementar outras instruções da tabela; o processador *não pode executar outras instruções da tabela.*

Sign	Suffix	Meaning	Flags
	EQ	Equal	Z = 1
	NE	Not equal	Z = 0
	CS	Carry set (identical to HS)	C = 1
	CC	Carry clear (identical to LO)	C = 0
	MI	Minus or negative result	N = 1
	PL	Positive or zero result	N = 0
	VS	Overflow	V = 1
	VC	Now overflow	V = 0
	AL	Always. This is the default	-
Unsigned	HI	Higher	C = 1 and Z = 0
	HS	Higher or same	C = 1
	LS	Lower or same	C = 0 or Z = 1
	LO	Lower (identical to CC)	C = 0
Signed	GT	Greater than	Z = 0 and N = V
	GE	Greater than or equal	N = V
	LE	Less than or equal	Z = 1 or N != V
	LT	Less than	N != V

Tabela 2: Instruções *assembly* de desvio condicional – padrão da ISA dos µP ARM 32 bits

Todos os programas em *assembly* do semestre deverão ser executados com condicionais implementadas apenas com as duas instruções sorteadas. Saltos incondicionais podem ser usados na lógica de programação à vontade.

Em outros processadores, outras nomenclaturas são comuns; por exemplo, é comum `BEQ dest` ter o mesmo significado que `BZ dest` (salte se o resultado mais recente da ULA for zero) ou `JR Z,dest` ou `JZ dest` ou `JMP Z,dest`.

6 Customização da validação

A validação consiste em executar no seu processador um programa para cálculo de números primos utilizando o crivo de Eratóstenes. Há dois itens sorteados na validação: o *final do loop* e alguma *complicação.* A equipe deve escolher um destes dois para implementar, obrigatoriamente, podendo ignorar o outro e sendo livre nas demais escolhas.

6.1 Conselho importante de VHDL!

Faça estas instruções adicionais do jeito mais idiota possível! Gaste linhas como se não houvesse amanhã, o compilador normalmente otimiza internamente estas construções, até certo ponto.

Quando a internet tenta ajudar, ela complica o código usando VHDL diferente do nosso feijão-com-arroz; do ChatGPT nem se fala, ele não é muito bem versado na linguagem. Tome cuidado e não encha o saco do professor com código que não foi você mesmo que digitou.

6.2 Final do Loop

Existem várias formas diferentes de se detectar o final do loop do cálculo dos número primos; dependendo da implementação, pode haver um loop, dois loops encaixados ou alguma outra variação (provavelmente com um "early exit"). Abaixo temos a descrição dos sorteados, supondo que devemos detectar os primos abaixo de 32 (é fácil adaptar para qualquer potência de 2 neste limite; perceba que $32_{10} = 10\ 0000_2$, ou seja, o bit b5 está setado). Pode adaptar também para o uso de acumulador ou ortogonal, conforme o caso.

- **Detecção do MSB setado usando SHIFT RIGHT:** basta "shiftar" 5 bits à direita (instrução `SLR A,Rn,5`) e comparar pra ver se deu zero ou não;
- **Detecção do MSB setado usando AND:** se fizer um AND com 100000_2 e der zero, não terminou ainda;
- **Detecção do MSB setado usando OR:** se fizer um OR com 11111_2 e der 31, não terminou ainda;
- **Detecção do MSB setado usando SHIFT LEFT com carry:** basta "shiftar" 11 bits à esquerda (instrução `SLL Rn,11`) e comparar pra ver se deu carry ou não; [2]
- **Contagem regressiva até zero usando DEC:** no loop, decrementa-se a variável de controle (um registrador) com uma instrução `DEC Rn` e encerra o loop se ela chegar no zero;
- **Contador com INC:** no loop, incrementa-se a variável de controle (um registrador) com uma instrução `INC Rn` e o final do loop deve ser verificado com este registrador Rn;
- **JB direto no bit b5:** o teste para ver se Rn==32 é simplesmente `JB Rn.b,destino` (ex.: `JB R2.5,bit5_setado`): a codificação da instrução deve especificar o n para identificar o registrador e o b para identificar qual o bit a examinar (se o bit estiver setado, abandone o loop);

[2] Essa instrução coloca o bit MSB do operando na flag de carry, shifta todos os outros à esquerda e insere um novo bit em 0 no bit LSB.

- **JNB direto no bit b5:** análogo ao anterior – enquanto não estiver setado, repita;
- **CTZ igual a 5 (count trailing zeroes):** uma instrução `CTZ R1,R2` irá contar quantos bits em zero são contíguos no LSB de R2 e guardar isso em R1 (por ex., se R2=1011000_2, R1 ficará com 3, pois há 3 zeros à direita); [3]
- **CLZ igual a 11 (count leading zeroes):** uma instrução `CLZ R1,R2` irá contar quantos bits em zero são contíguos no MSB de R2 e guardar isso em R1 (por ex., se R2=0000000001011000_2, R1 ficará com 9, pois há 9 zeros à esquerda); [4]
- **Loop com DJNZ:** a instrução `DJNZ Rn,destino` decrementa Rn e salta para o destino se o resultado não for zero;
- **Loop com CJNE:** a instrução `CJNE Rn,Rm,destino` compara Rn com Rm e salta para o destino se eles não forem iguais.

Se você quer fazer de outro jeito, consulte previamente o professor para autorização.

6.3 Complicação

Alguns dos itens abaixo incluem uma constante *cte* única para a equipe, derivada de um *hash* dos nomes dos integrantes. Os sinais "bus debug," "bit debug," "exception" e "halted" são pinos de saída que devem estar presentes no top level do projeto caso sejam usados.

- **Exceção opcode inválido:** se o microprocessador encontrar um código binário que não representa uma instrução válida, ele deverá paralisar a execução e indicar isso levantando para 1 o sinal em um pino de saída chamado 'exception', no top level;
- **Instrução Halt ao final:** todo programa deverá terminar com uma instrução HALT; quando ela for executada, o microprocessador deverá paralisar a execução e indicar isso levantando para 1 o sinal em um pino de saída chamado 'halted', no top level;
- **Primos $<$ 1024:** todos os primos menores do que 1024 deverão ser exibidos ao final;

[3] Implemente isso em VHDL do jeito mais estúpido do mundo: um `when-else` com 16 cláusulas.

[4] Implemente isso em VHDL do jeito mais estúpido do mundo: um `when-else` com 16 cláusulas.

- **Exceção endereço inválido ROM:** caso o programa tente executar uma instrução em um endereço de ROM em que não há memória (ou seja, o microprocessador tenta escrever no PC um valor maior do que o tamanho da ROM), ele deverá paralisar a execução e indicar isso levantando para 1 o sinal em um pino de saída chamado 'exception', no top level;
- **Exceção endereço inválido RAM:** caso o programa tente ler ou escrever dados em um endereço de RAM em que não há memória (ou seja, o microprocessador tenta ler ou escrever num endereço maior do que o tamanho da RAM), ele deverá paralisar a execução e indicar isso levantando para 1 o sinal em um pino de saída chamado 'exception', no top level;
- **Tabela do crivo começa no endereço *cte* da RAM:** normalmente a tabela começa exatamente no endereço zero, o que facilita a implementação – aqui deve-se iniciar a tabela com um deslocamento inicial (p. ex., se a constante for 64, o endereço 64+7=71 será correspondente ao número primo 7);
- **A constante *cte* é número primo?** Depois que a construção da tabela for finalizada, o programa deverá usá-la para determinar se a constante especificada é um número primo ou não;
- **Colocar o enésimo primo no bus debug, sendo n a *cte*:** ou seja, o programa deverá montar a tabela inteira e depois contar os primos um a um até encontrar o enésimo;
- **NOP é opcode *cte*, o restante é exceção:** é idêntico ao item "Exceção opcode inválido," exceto pelo fato de que NOP não é mais o opcode zero (portanto uma instrução com código 0x0 gera exceção);
- **Colocar no bus debug um divisor de *cte*:** usando a tabela calculada dos primos, testar a divisibilidade da constante por cada um deles até achar um divisor;
- **Indicar, ao final, em debug bit, se *cte* é primo ou não:** usando a tabela calculada dos primos, testar a divisibilidade da constante por cada um deles até determinar primalidade, indicando 1 no bit debug caso seja primo.

7 Checklist das instruções

- Instrução MOV para cópia de valor de um registrador para o outro;
- Instrução ADD que soma dois valores e grava em um registrador (e ADDI com constante, se for o caso);

- Instrução SUB que subtrai dois valores e grava em um registrador (ou SUBB, e SUBI com constante, se for o caso);
- Carga de uma constante para um registrador (com LD ou ADDI com registrador de constante zero);
- Escrita de um registrador na memória com ponteiro e leitura da memória com ponteiro;
- Instrução NOP (No OPeration, não faz nada) é o código de máquina 0x0 exceto se especificado outro;
- Saltos incondicionais e condicionais baseados em flags;
- Demais instruções sorteadas para a equipe.