

OBJETIVOS

- Conhecer o mínimo do ambiente de trabalho com Verilog e FPGA
- Conhecer as técnicas de gerenciamento de complexidade em sistemas digitais
- Entender a relação entre bloco lógico (*blackbox*) e módulo em Verilog
- Aprender quando utilizar *wire* e *reg* em Verilog
- Usar *assign* para conectar fios e para gerar lógica básica
- Conhecer os tipos de porta básicas de Verilog: *input*, *output* e *inout*
- Aprender o método com ponto para instanciação em Verilog
- Rever vetores de bits, sua declaração e uso
- Rever concatenação de bits
- Conhecer o formato da linha de declaração de módulo para Verilog 2001 em diante
- Implementar um somador de 16 bits hierarquicamente a partir de módulos dados
- Conectar elementos pré-definidos seguindo um esquema (LCD, somador)
- Abrir um projeto pronto do Quartus II, obtendo as configurações de FPGA, pinagem, e nomes dos sinais da placa utilizada
- Apresentar 6 números em Hexadecimal de 16 bits no LCD
- Conhecer os principais erros de Verilog e suas soluções

1 INTRODUÇÃO

Durante o ensino de Laboratório de Arquitetura de Sistemas Digitais e Laboratório de Circuitos Lógicos detectamos uma deficiência dos alunos em construir hierarquicamente os sistemas em Verilog. Nos últimos anos esta tem sido uma causa frequente de diminuição da eficiência no ensino de outros conteúdos. Como se trata de uma matéria bastante simples, este experimento inicial foi criado, com o objetivo de introduzir o mais rápido possível estes conceitos, além de fornecer algumas dicas e outras informações que devem auxiliar em soluções de problemas recorrentes destes laboratórios. Desta forma, muitos tópicos simples serão abordados, no começo isoladamente e em seguida em conjunto para formar um sistema complexo, onde as partes (módulos) já são dadas.

2 AMBIENTE DE TRABALHO

Nos computadores do LTD, Laboratório de Técnicas Digitais, temos algumas contas de usuário, que são comuns para todos os computadores. A conta LASD (Laboratório de Arquitetura de Sistemas Digitais) deve ser usada. A senha é o mesmo nome do usuário (LASD). Ou seja, **Conta: LASD, Senha: LASD**.

Nesta conta os alunos têm acesso aos softwares que utilizamos nas três etapas do LASD: FPGA, microcontrolador e projeto. Trataremos nesta introdução apenas da parte de FPGA.

Cada aluno deve criar seu próprio diretório de trabalho, com seu nome e turma. Este diretório poderá ser revisado pelo professor para atribuir conceitos. Como os computadores podem não estar em rede, cada aluno deve, se possível, **usar sempre o mesmo computador**.

TAREFA: Criar diretório com seu nome e sobrenome, por exemplo, Andre_Marques, dentro do diretório C:\LASD\ANO, do seu computador.

Um projeto completo já foi criado e está disponível no diretório INSTANCIA no desktop de cada computador. A extensão dos arquivos de projeto é “.qpf”. Deve-se **sempre abrir os arquivos de projeto** e não os arquivos individuais Verilog no início dos trabalhos.

TAREFA: Copiar o diretório inteiro denominado INSTANCIA para dentro do seu diretório de trabalho. Após copiar abrir o arquivo instancia.qpf (duplo clique), do seu diretório.

2.1 LINGUAGEM VERILOG



A linguagem que utilizamos para descrever os hardwares que construiremos é Verilog. A linguagem Verilog sofreu várias revisões. Muitos livros apresentam o Verilog mais antigo, padronizado em 1995, sendo também conhecido como padrão IEEE 1364-1995. Uma versão revisada foi publicada em 2001: IEEE 1364-2001. Houve também outra revisão em 2005, com poucas mudanças.

Outro padrão também foi desenvolvido, SystemVerilog, no intuito de remover deficiências de Verilog no projeto em níveis mais altos de abstração e outros problemas de ordem prática.

A principal dificuldade encontrada no ensino de Verilog é a semelhança das construções com linguagens de programação. **Verilog é uma HDL** (Hardware Description Language), ou seja, linguagem de descrição de hardware. Com ela é possível construir sistemas digitais operando em diferentes níveis de abstração. Estes vários níveis de abstração são outra fonte comum de confusão. A mesma linguagem pode ser utilizada para descrever em alto nível (sem detalhes de implementação) e de forma comportamental (como um programa ou algoritmo) em um extremo até uma *netlist* (conjunto de dispositivos e suas conexões, sem preocupação funcional) formada por transistores no outro extremo. Para cada etapa de um projeto determinado nível de abstração é mais adequado.

No nosso laboratório apenas o nível RTL (Register Transfer Level) será utilizado. Este nível é intermediário em abstração, sendo o nível mais alto que conseguimos no qual a geração dos níveis mais baixos ainda seja completamente automática. Ou seja, embora seja mais fácil para um ser humano lidar com o nível comportamental, ainda não existem ferramentas que convertam automaticamente esta descrição para circuitos. O nível RTL ainda exige que sejam realizadas algumas tarefas comuns em níveis mais baixos, como interconexão.

Com a entrada em RTL é possível realizar a síntese, que é a criação de um *netlist* formado por portas lógicas, flip-flops e interconexões, capaz de realizar o que foi definido em Verilog. As portas lógicas e flip-flops por sua vez são minimizados e mapeados nos elementos internos do FPGA. Todo este processo é automático.

TAREFA: Sintetizar o circuito já existente no projeto clicando no símbolo . Programar o circuito na placa DE2 clicando em  e em seguida start, verificar seu funcionamento. A placa precisa estar ligada na fonte de 9V e na USB no conector mais próximo da entrada da fonte.

Normalmente o trabalho de desenvolvimento de um sistema digital em Verilog inclui a simulação para verificação do funcionamento antes de sua realização através da síntese. No entanto, deve-se sempre lembrar que a linguagem Verilog pode ser utilizada nestas diversas tarefas, mas as construções utilizadas são diferentes. Ou seja, subconjunto diferentes da mesma linguagem são utilizados na simulação e síntese. O subconjunto sintetizável é bem menor, sendo visto no LASD. Na simulação, a descrição em Verilog é a entrada para um software denominado simulador, que interpreta o Verilog e este pode ter construções similares a software, promovendo muita confusão. A principal diferença entre Verilog e software comum é a concorrência inerente nas descrições Verilog. O Verilog simulável possui construções como laços, funções, variáveis, inicialização, etc. Nada disso é sintetizável e gera circuito. SystemVerilog possui níveis ainda mais alto de abstração, incluindo classes e orientação a objeto. Mas novamente, esta parte da linguagem é utilizada para simulação e verificação funcional, não para produzir circuitos. O Quartus II não consegue lidar com nenhuma descrição que não seja puramente sintetizável. O fluxo de projeto incluindo simulação pode ser visto na Figura 1. Devido a esta ambiguidade, muitos dos **tutoriais e cursos na Internet são inadequados**, pois tratam principalmente de Verilog para simulação.

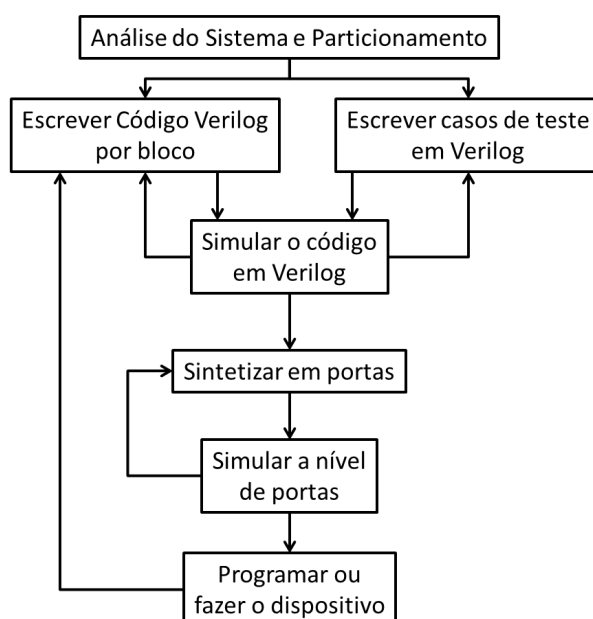


Figura 1. Fluxo de projeto simplificado em Verilog

2.2 SOFTWARE UTILIZADO

O software que utilizamos chama-se Quartus II (pronuncia-se “quorás”). O Quartus II é capaz de operar com todas as versões de Verilog. A extensão no final dos arquivos deve ser “.v”. Para usar SystemVerilog no Quartus II deve-se usar a extensão “.sv”. Deve-se observar que nem todas as características de SystemVerilog são suportadas pelo Quartus II.

Este software foi desenvolvido pelo próprio fabricante do FPGA que utilizamos, *Altera*. O software é bastante complexo, mas nós utilizamos apenas algumas funcionalidades. Especificamente utilizamos o sintetizador (alguns chamam de compilador, mas este nome remete-se a software, o que gostaríamos de evitar), que recebe como entrada um ou mais arquivos em Verilog e produz um arquivo de configuração (bitstream ou fluxo de bits), com extensão “.sof”. Neste arquivo estão as informações de que função lógica o FPGA vai realizar, como os pinos serão utilizados e os valores iniciais dos elementos de armazenamento. Ou seja, a programação do FPGA com este arquivo literalmente o converte no circuito que foi projetado em Verilog.

TAREFA: Visualizar o circuito gerado na etapa anterior clicando em *Tools -> Netlist Viewers -> RTL Viewer* e em seguida *Tools -> Netlist Viewers -> Technology Map Viewer (Post-Mapping)*. Um duplo clique em cada módulo entrará no nível inferior da hierarquia. Um clique fora voltará para o nível superior. Quais são os componentes de mais baixo nível em cada visualização?

3 GERENCIAMENTO DE COMPLEXIDADE

O ser humano possui uma capacidade limitada de lidar diretamente com sistemas complexos, necessitando de ferramentas que o ajudem nestas tarefas. Um sistema complexo é aquele que possui muitos componentes e interconexões. Um carro, por exemplo, pode ter de 15.000 a 30.000 componentes. Um avião moderno pode chegar a ter meio milhão de componentes. Atualmente um chip digital, como um processador, pode ter cerca de 5 bilhões de transistores, sendo de longe **o artefato mais complexo já concebido pela humanidade**.

Para gerenciar a complexidade em sistemas digitais os métodos usuais podem ser utilizados. Estes métodos são usados em diversas outras áreas do conhecimento que lidam com sistemas complexos, como ciência da computação, por exemplo. Destacamos a modularidade, regularidade, hierarquia, generalização (reuso), abstração e automação.

A **modularidade** é a separação de um sistema em componentes. Além disso, há uma diferenciação entre a *interface* usada para a ligação entre os componentes e a *funcionalidade*, ou seja, o que cada componente faz. Desta forma podemos conectar os componentes conhecendo apenas a sua interface e podemos projetar o sistema em alto nível de abstração sabendo apenas o que cada componente faz. Dizemos que cada componente é um módulo.

A **regularidade** é o uso de componentes e estruturas semelhantes, formando um arranjo em uma ou mais dimensões. Por exemplo, uma memória é normalmente um arranjo bidimensional de elementos de

memória capazes de guardar 1 bit. Um somador de 32 bits pode ser formado por 32 somadores completos (que recebem e geram o vai-um), etc.

Com a **hierarquia** cada módulo pode ser formado por outros módulos, formando um módulo mais complexo internamente. Através da interface deste novo módulo podemos acessá-lo sem nos preocuparmos com o que tem no seu interior.

O **reuso** devido à **generalização** é a capacidade de usar o projeto de um componente na solução de diversos problemas, similares ou não. O reuso implica na construção de componentes mais genéricos e parametrizáveis. Um exemplo seria o uso de um gerador de seno podendo gerar também o cosseno ao ter o módulo recebendo também uma fase inicial.

A **abstração**, como já vimos, é quando operamos com representações mais próximas dos seus significados reais, escondendo os detalhes de implementação. Estas representações de mais alto nível são mais naturais para o ser humano. Um exemplo é usar números decimais em uma descrição, quando sabemos que em qualquer circuito digital temos **apenas zeros e uns**, ou seja, números binários.

A **automação** é a utilização de equipamentos ou dispositivos para a realização de tarefas. No caso de projetos de hardware digital, são utilizados softwares cuja saída seja um dos artefatos do projeto. As entradas para estes softwares podem ser definições de alto nível de abstração, como no caso de um sintetizador, ou uma descrição de características, como tamanho, largura, velocidade, etc. como entradas de um gerador de memórias.

Estas técnicas podem ser desenvolvidas em um projeto de sistemas digitais realizado em Verilog, como veremos em seguida.

4 MODULARIDADE E INSTANCIÇÃO EM VERILOG

4.1 Módulos e blocos lógicos

Em Circuitos Lógicos aprendemos o conceito de bloco lógico. O bloco lógico é uma nomenclatura especial para o conceito de caixa preta, usada em todas as outras áreas. Em um bloco lógico apresentamos a sua interface, através de suas conexões de entrada e saída, como também a sua funcionalidade de alto nível, através da classe do bloco. Por exemplo, na Figura 2, temos alguns exemplos de blocos lógicos.

Quando representamos os blocos lógicos em Verilog utilizamos módulos. As palavras reservadas *module* e *endmodule*, delimitam a definição de um módulo. Uma vez um módulo definido ele pode ser usado uma ou mais vezes nos circuitos. Usar um módulo chama-se **instanciar**. Podemos instanciar módulos que tenham sido criados por nós mesmos ou que já estejam prontos, de alguma forma.

Quando definimos um módulo também informamos quais são as suas entradas e suas saídas. Podemos ver como ficam as definições dos quatro blocos lógicos da Figura 2 como módulos Verilog na Figura 3. Não importa por enquanto o que fica dentro dos módulos, apenas a sua interface. De posse da interface, podemos fazer as interconexões, ou seja, instanciar os módulos.

4.2 Módulo topo

Tudo em Verilog acontece dentro dos módulos. Assim o circuito completo também é um módulo. Ele não é especial, mas por ser o módulo que contém todos os outros, dizemos que ele fica no topo da hierarquia, sendo chamado de módulo *top*.

Nos experimentos do laboratório, chamamos o módulo topo de “Mod_Teste”. Em um circuito em FPGA, o módulo topo fica ligado nos pinos do FPGA, sendo reservado para os testes na placa. Este módulo tem este nome porque o usamos para testar os módulos criados nos experimentos. Este teste é feito colocando os outros módulos no interior de Mod_Teste. Esta colocação é, obviamente, apenas uma instanciação. Devemos instanciar os módulos criados e as outras partes necessárias para o teste. Por exemplo, para testar um somador temos que instanciar-lo, conectando as portas de entrada e saída e poderíamos instanciar decodificadores de binário para 7 segmentos para que possamos visualizar os valores nos displays da placa. O módulo top (“Mod_Teste”) é o único que conhece os detalhes da placa, ou seja, nomes de chaves e de LEDs. Este módulo instancia todos os outros, hierarquicamente.

4.3 Instanciação

Para se instanciar os módulos (usá-los nos circuitos), podemos usar uma das duas sintaxes existentes em Verilog. A sintaxe antiga, do Verilog 1995, baseia-se na posição para informar o que está ligado onde. Deve-se lembrar que em Verilog, quando um “fio” (um nome de fio ou de “variável”) aparece em dois lugares, há uma conexão entre estes dois lugares. Isto fica melhor explicado com exemplo.

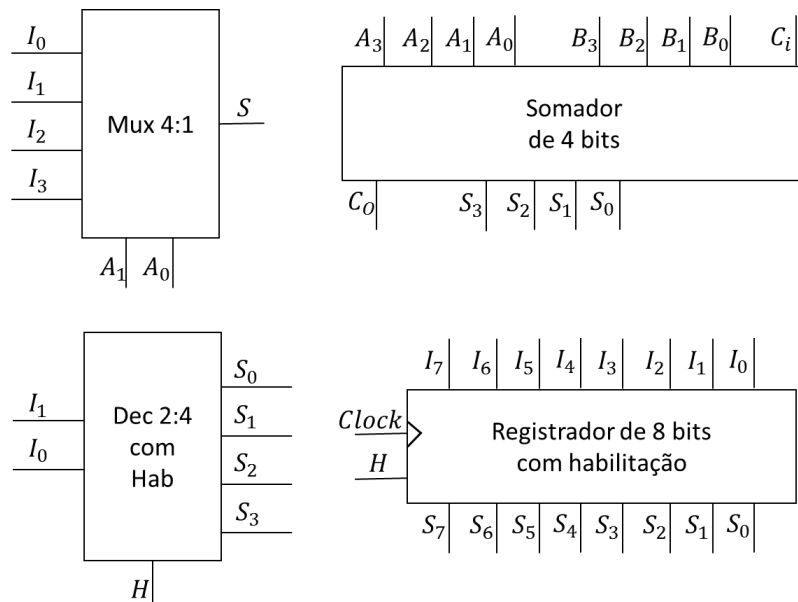


Figura 2. Alguns exemplos de blocos lógicos.

```

module mux4to1 (input i3, i2, i1, i0, a1, a0, output s);
...
endmodule

module somador4b (input a3, a2, a1, a0, b3, b2, b1, b0, ci,
                  output s3, s2, s1, s0, co);
...
endmodule

module dec2to4 (input i1, i0, h, output s0, s1, s2, s3);
...
endmodule

module reg8b (input i7, i6, i5, i4, i3, i2, i1, i0, h, clock,
              output s7, s6, s5, s4, s3, s2, s1, s0);
...
endmodule

```

Figura 3. Módulos correspondentes aos blocos lógicos da Figura 2.

```

wire y3, y2, y1, y0, z, w1, w0;

mux4to1 u1 (y3, y2, y1, y0, w1, w0, z);

```

Figura 4. Instanciação do Mux 4 para 1 usando o método antigo

No exemplo da Figura 4, criamos 7 fios, usando a diretiva *wire* de Verilog. Em seguida conectamos estes fios no multiplexador 4 para 1. A palavra *mux4to1* coloca o mux no circuito. A palavra *u1* indica um nome de instância que serve para identificar quando tem mais de uma instância do mesmo tipo. As conexões são realizadas na ordem, de acordo como o módulo foi definido (Figura 3). Assim os fios *y*'s são ligados às entradas *i*, os fios *w*'s às entradas de seleção *a* e o fio *z* a saída *s*.

Vê-se que se errar a ordem ou esquecer alguma entrada, as conexões vão estar todas erradas. Para corrigir isto foi criado no Verilog 2001 o método novo de instanciação que utiliza a notação ponto (.).

Veja como fica a mesma instanciação da Figura 4 com este método:

```

mux4to1 u1 (.i3(y3), .i2(y2), .i1(y1), .i0(y0),
            .a1(w1), .a0(w0), .s(z));

```

Figura 5. Instanciação do Mux 4 para 1 usando notação ponto

Observe a lei de formação. Colocamos o nome da porta (pino) de entrada ou saída precedida por um ponto. Este é o nome usado na definição do módulo. Usamos parênteses e colocamos dentro destes o nome local que deve ser conectado a este pino.

Vemos que assim podemos trocar a ordem e até mesmo omitir conexões. Este método elimina muitos erros. Relembre que para conectar os elementos basta que o mesmo nome apareça em dois lugares. Um truque útil para não errar ou esquecer conexões e digitar menos é ***copiar e colar a definição do módulo*** e inserir os pontos, parênteses e conexões.

Há sempre uma grande confusão nos alunos quanto à instanciação. A linha em Verilog se parece muito com uma chamada de função em C. A semelhança é apenas visual. Deve-se sempre imaginar a instanciação como a **colocação de um componente no circuito**, porque é verdadeiramente isto que está sendo feito.

4.4 Fios de entrada e saída

Nas definições dos módulos, as portas de entrada e de saída também definem fios internos com os mesmos nomes. Por exemplo, na definição do decodificador binário 2 para 4, os fios *i1*, *i0*, *h*, *s0*, *s1*, *s2* e *s3* já estão definidos internamente e quando usados vão fazer as conexões com estas portas de entrada e saída. Como exemplo, veja na Figura 6, que o uso do termo `LEDR[0]` no parêntesis da saída *s* na instanciação do `mux4to1` já é suficiente para conectar essa saída ao LED mencionado. Não é necessário e não recomendado fazer a conexão em duas partes, criando um fio e usando a diretiva `assign`, vista em seguida.

```
mux4to1 u1 (.i3(y3), .i2(y2), .i1(y1), .i0(y0),  
            .a1(w1), .a0(w0), .s(LEDR[0]));
```

Figura 6. Exemplo de uso de `assign`

4.5 `assign`

Os fios não têm direção, mas as portas de entrada e saída sim. As entradas não definem o valor internamente, mas recebem o valor lógico de quem se conectar a elas. As saídas definem valores lógicos para serem ligados em outras entradas.

Podemos ligar um fio em outro através da diretiva `assign` (associar). Quando associamos um fio a outro, temos que seguir algumas regras básicas. Veja o exemplo da Figura 7.

```
assign a = b;  
assign LEDR[3] = SW[2];
```

Figura 7. Exemplo de uso de `assign`

No lado esquerdo do sinal de igual está quem recebe a atribuição e no lado direito que gera o valor lógico. Assim, na primeira linha da Figura 7, *a* recebe o valor lógico de *b*. Na segunda linha, um LED (veremos o colchete em breve) recebe o valor lógico de uma chave. Estes símbolos obviamente precisam estar definidos (*a*, *b*, *LEDR*, *SW*) no módulo em que os `assign` aparecem.

Uma associação contínua define o valor de fios ou de portas de saída. **É um erro comum associar continuamente ao tipo `reg`**. Cada fio pode ter apenas uma fonte de sinal, seja uma atribuição contínua ou uma conexão a uma saída em um módulo.

4.6 Vetores de bits e concatenação

É muito comum em Verilog e circuitos digitais agruparmos os sinais, indicando que estes devem ser operados conjuntamente. Em Verilog, podemos agrupar vários fios (ou variáveis) de um bit, formando um fio

composto, denominado vetor de bits ou barramento. Este fio, com largura n , pode agora representar 2^n valores. A sintaxe é a seguinte:

```
wire [3:0] y;

assign y[2:1] = 2; // 1 0 em binário, y[2] <- 1, y[1] <- 0

assign y[2:1] = {1'b1, 1'b0};
```

Figura 8. Exemplo de uso de assign

Para definir o vetor de bits usamos a sintaxe da primeira linha. Observe que a dimensão vem antes do nome do fio ou variável. Para utilizar, podemos pegar qualquer parte do vetor de bits. Na segunda linha do exemplo, estamos pegando 2 bits de um vetor de 4 bits e estamos associando aos valores binários 1 e 0 (2 é 10 em binário). A terceira linha faz exatamente a mesma coisa da segunda. Usamos uma concatenação, ou seja, uma união de bits separados, para criar temporariamente um vetor. A sintaxe é usando o colchete. Os bits dentro dos colchetes são unidos em um vetor na ordem em que aparecem.

Na Figura 9 podemos ver os mesmos blocos lógicos da Figura 2, usando vetores de bit. Na Figura 9 temos as definições dos módulos em Verilog referentes a estes blocos lógicos.

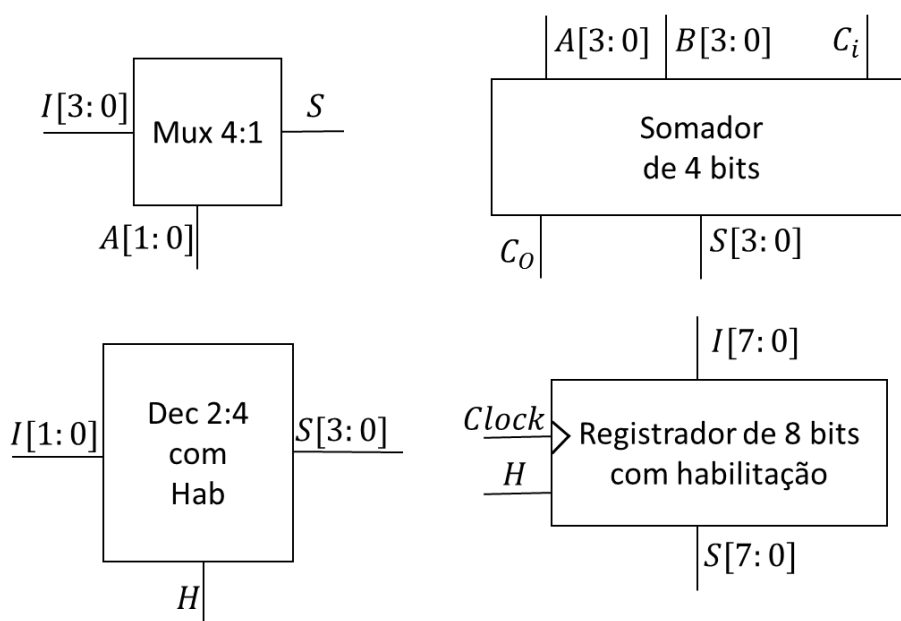


Figura 9: Blocos lógicos com Vetores de Bits

```

module mux4to1 (input [0:3] i, input [1:0] a, output s);
...
endmodule

module somador4b (input [3:0] a, b, input ci,
                  output [3:0] s, output co);
...
endmodule

module dec2to4 (input [1:0] i, input h, output [3:0] s);
...
endmodule

module reg8b (input [7:0] i, input h, clock, output [7:0] s);
...
endmodule

```

Figura 10. Módulos correspondentes aos blocos lógicos da Figura 8 usando vetores de bits.

5 SOMADOR DE 16 BITS USANDO SOMADORES DE 4 BITS

Veja o circuito da Figura 11. Trata-se de um somador de 16 bits formado a partir de 4 somadores completos de 4 bits. Observe o uso dos vetores de bits.

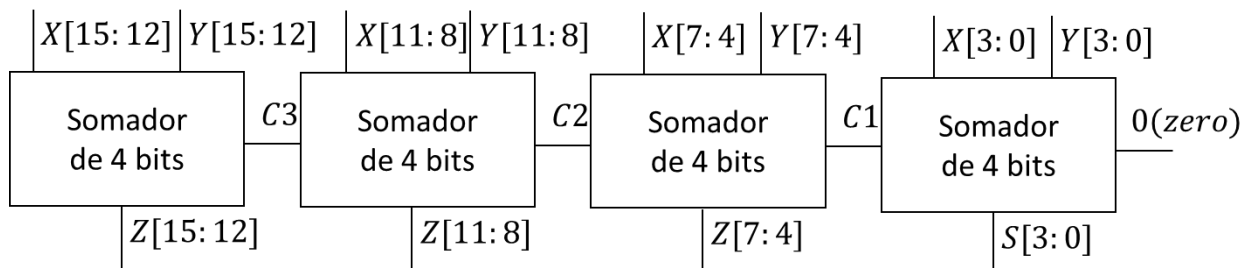


Figura 11. Somador de 16 bits realizado com somadores de 4 bits

TAREFA: Crie um novo arquivo Verilog: File -> New -> Verilog File. Faça a declaração do módulo correspondente ao bloco lógico do somador de 16 bits. As entradas são X e Y e a saída é Z.

TAREFA: Declare os fios internos do somador de 16 bits.

Responda – quantos fios são necessários? Precisa de fios para X, Y ou Z?

TAREFA: instancie 4 módulos somadores de 4 bits (*somador4b*). Cada um deve receber um nome de instância e as partes corretas dos vetores de bits. Use a notação com ponto.

DICA: O arquivo com o somador está na pasta copiada para o seu diretório. Abra o arquivo verilog (File -> Open) e copie e cole a linha module e reaproveite os nomes das portas de entrada e saída.

Para referência, o somador de 4 bits pode ser realizado como na Figura 12.

```
module somador4b (input [3:0] a, b, input ci,
                  output [3:0] s, output co);

assign {co,s} = a + b + ci;

endmodule
```

Figura 12. Implementação do somador de 4 bits. Observe que utilizamos a mesma interface descrita na Figura 10 e que o somador não foi construído a partir de somadores de 1 bit.

6 CIRCUITO PARA APRESENTAÇÃO DE SOMA NO LCD

Vamos agora realizar o circuito apresentado na Figura 14 através de instanciação dos módulos anteriormente descritos. Para tanto, vamos abrir o projeto INSTANCIA2 no mesmo diretório anterior.

TAREFA: Abrir o projeto instancia2.qpf com um duplo clique no arquivo. Observe que o arquivo verilog *mod_teste.v* não possui nenhuma instanciação.

TAREFA: Incluir os arquivos usados no projeto. Clique com o botão direito em Files -> Add/Remove Files. Ver Figura 13. Acrescente o somador16b.v criado na tarefa anterior.

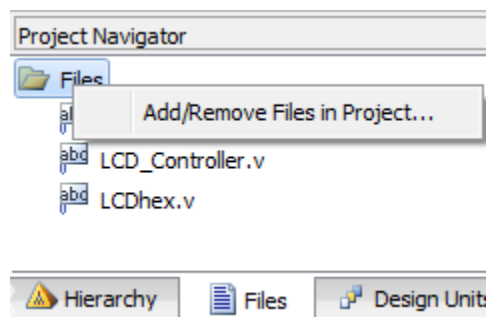


Figura 13. Acrescentando arquivos a um projeto existente

TAREFA: Instanciar os diversos elementos da Figura 14, no arquivo `mod_teste.v`, abaixo das definições de entrada e saída

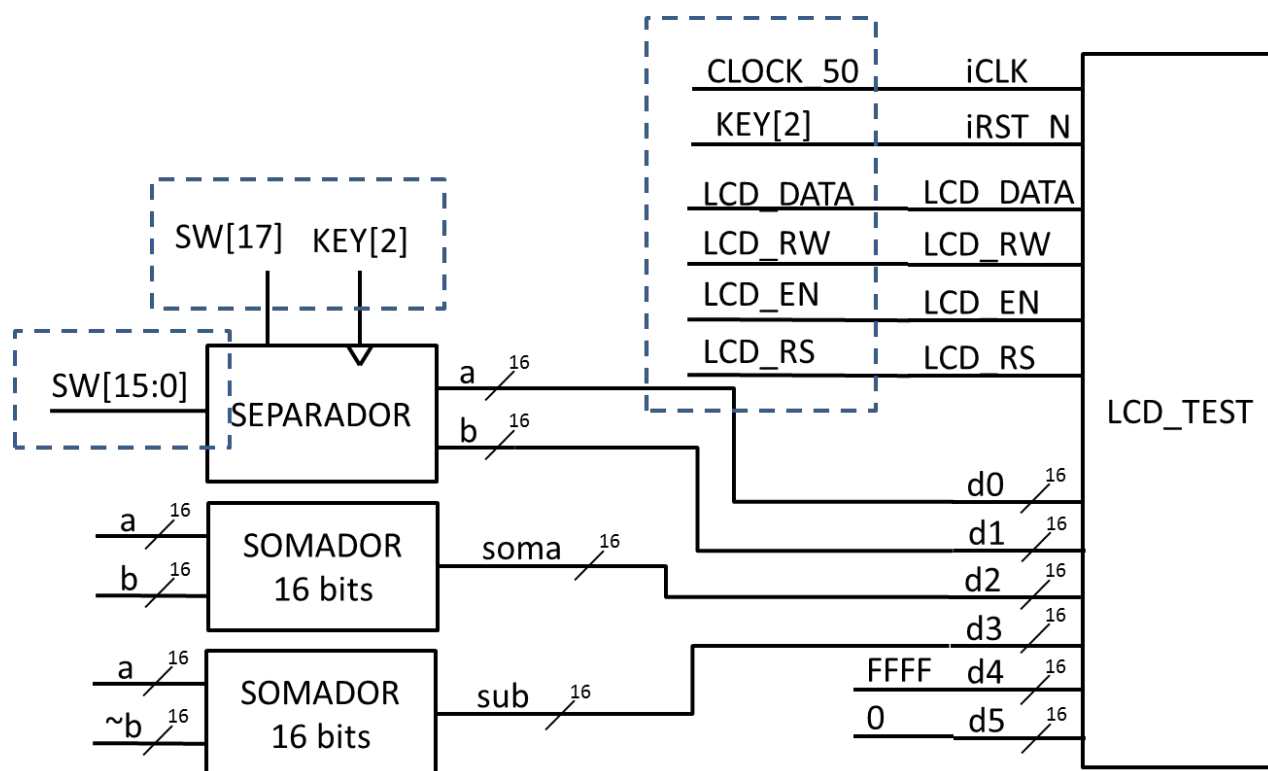


Figura 14. Circuito com somadores de 16 bits e mostrador LCD. Os nomes no interior das caixas tracejadas são entradas e saídas do `Mod_Teste`.

7 PROBLEMAS MAIS COMUNS E SUAS SOLUÇÕES

O Quartus adota a nomenclatura oficial do padrão IEEE Std - 1800-2012, unificado para Verilog e SystemVerilog. Com isso, as mensagens de erro parecem criptográficas para os alunos. Para descomplicar, devemos aprender alguns dos conceitos usados no padrão. Primeiro, `reg` e `wire` são tipos de dados. `wire` é um indicador de tipo `net`, *net type* em inglês. `reg` é um indicador de tipo variável, *variable type* em inglês.

Como sabemos, o tipo `net` é usado para fazer conexões, e por isso, não guarda valores. O seu valor é definido pela saída em que está ligado, o que em Verilog chamamos de *driver*. Ou seja, um `wire` em Verilog não é como um fio físico. O `wire` recebe o valor do *driver*. O driver mudando, o `wire` muda. Podemos ligar um `wire` a um driver usando `assign` ou a uma saída, quando instanciamos. Um `assign` é uma atribuição contínua, em inglês *continuous assignment*. Devido a isso, não podemos atribuir um valor a um `wire` em dois lugares, ou seja, não pode ter mais de um *driver*.

O tipo variável, ou seja, `reg`, é usado para guardar valores ou como auxiliar no desenvolvimento da descrição da lógica. O tipo `reg` obtém o seu valor sempre de forma explícita, através de uma atribuição bloqueante ou não bloqueante, dentro de um procedimento `always` (o padrão mudou o nome de bloco para *procedure*).

Outra questão importante são os lados de uma atribuição, tanto contínua (assign =) quanto procedural (always), bloqueante (=) quanto não-bloqueante (<=). As atribuições têm o lado esquerdo, em inglês, *left-hand side*, ou seja, antes do operador = ou <=, e o lado direito, em inglês, *right-hand side*, após o operador. O lado esquerdo recebe os valores produzidos pelo lado direito.

Com essas informações podemos preencher a tabela abaixo com os erros mais comuns.

Código	Erro informado	Significado	Correção
<pre>reg ra; assign ra = 1'b1;</pre>	Error (10219): Verilog HDL Continuous Assignment error at Mod_Test.v(44): object "ra" on left-hand side of assignment must have a net type	<p>Usou um reg onde era para usar um wire.</p> <p>Erro de atribuição contínua: o objeto “ra” no lado esquerdo da atribuição deve ser do tipo net.</p>	<p>Tornar ra um fio.</p> <pre>wire ra; assign ra = 1'b1;</pre> <p>ou trocar o assign por always</p> <pre>always @* ra = 1'b1;</pre>
<pre>wire wa; always @* wa = 1'b1;</pre>	Error (10137): Verilog HDL Procedural Assignment error at Mod_Test.v(45): object "wa" on left-hand side of assignment must have a variable data type	<p>Usou um wire onde era para usar um reg.</p> <p>Erro de atribuição procedural: o objeto “wa” no lado esquerdo da atribuição deve ter o tipo variável.</p>	<p>Tornar wa uma variável (reg).</p> <pre>reg wa; always @* wa = 1'b1;</pre> <p>ou trocar o always por assign</p> <pre>assign wa = 1'b1;</pre>
<pre>module mymod (input a, output b,);</pre>	Error (10170): Verilog HDL syntax error at mymod.v(4) near text "); expecting a direction	<p>Uma vírgula no final.</p> <p>Erro de sintaxe perto de “)”; esperando uma direção</p>	<p>Retirar a vírgula.</p> <pre>module mymod (input a, output b);</pre>
<pre>wire res_a; mymod m1 (.a(1'b1), b(res_a));</pre>	Error (10267): Verilog HDL Module Instantiation error at Mod_Test.v(15): cannot connect instance ports both by order and by name	<p>Esqueceu o ponto.</p> <p>Erro de instanciação: não pode conectar portas de instâncias tanto por ordem quanto por nome</p>	<p>Colocar o ponto.</p> <pre>mymod m1 (.a(1'b1), .b(res_a));</pre>
<pre>always @ (posedge clk or negedge clk) begin if (clk == 1'b1) q_pos <= data; else q_neg <= data;</pre>	Error (10239): Verilog HDL Always Construct error at mymod.v(7): event control cannot test for both positive and negative edges of variable "clk"	<p>Usou borda positiva e negativa</p> <p>Erro no always: não pode testar bordas positivas e negativas</p>	<p>Separar em dois always</p> <pre>always @(posedge clk) begin q_pos <= data; end always @(negedge clk) begin q_neg <= data; end</pre>

OBJETIVOS

- Conhecer a divisão entre caminho de dados e controle em um sistema digital, em particular de um processador;
- Conhecer os operadores de deslocamento e concatenação de vetores de bits;
- Utilizar a instanciação de módulos existentes;
- Realizar uma descrição Verilog a partir de uma especificação.

INTRODUÇÃO

A especificação para um computador consiste na descrição da sua aparência para o programador no nível mais baixo, sua *arquitetura de conjunto de instruções* (ISA, *instruction set architecture*). A partir da ISA, é formulada uma descrição de alto nível do hardware que implementa o computador, chamada de *arquitetura do computador*. Esta arquitetura, para um computador simples, é tipicamente dividida em um caminho de dados (*datapath*) e um controle. O *datapath* é definido por três componentes básicos:

1. Um conjunto de registradores;
2. As microoperações realizadas nos dados armazenados nos registradores e
3. A interface de controle

A unidade de controle fornece sinais que controlam as microoperações realizadas no *datapath* e em outros componentes do sistema, tais como memórias. Além disso, a unidade de controle realiza seu próprio controle, determinando a sequência de eventos que ocorrem. Esta sequência pode depender dos resultados das microoperações executadas agora e no passado. Em um computador mais complexo, tipicamente encontramos múltiplas unidades de controle e *datapaths*.

O conceito de *datapath* está completamente associado ao conceito de transferência entre registradores, com o conteúdo sendo modificado por elementos combinacionais. Uma microoperação determina que registradores são utilizados como fontes e como destinos destas transferências e que função combinacional deve ser realizada.

Neste experimento será implementado o conjunto de registradores e sua interface de controle.

DATAPATH

Ao invés de ter cada registrador individual realizando suas microoperações diretamente, sistemas computacionais normalmente empregam um número de registradores de armazenamento em conjunto com uma unidade de operações compartilhada chamada de unidade lógica e aritmética, abreviada como ULA. Para realizar as microoperações, os conteúdos dos registradores fonte especificados são aplicados às entradas da ULA

compartilhada. A ULA realiza uma operação e o resultado desta operação é transferido para um registrador de destino. Com a ULA como um circuito combinacional, a operação inteira de transferência entre registradores originando-se nos registradores fonte, passando pela ULA e indo para o registrador de destino é realizada durante um ciclo de clock. As operações de deslocamento podem ser realizadas em uma unidade separada ou podem ser implementadas dentro da ULA.

Relembrando, a combinação de um conjunto de registradores com uma ULA compartilhada e os caminhos de interconexão (os barramentos) forma o datapath para o sistema. Podemos construir um processador simples através da associação de um datapath e uma unidade de controle associada.

O datapath e a unidade de controle são as duas partes de um processador, ou CPU, de um computador. Além dos registradores, o datapath contém a lógica digital que implementa as várias microoperações. Esta lógica digital consiste de barramentos, multiplexadores, decodificadores e circuitos de processamento. Quando um grande número de registradores é incluído em um *datapath*, os registradores são mais convenientemente conectados através de um ou mais barramentos. Registradores em um datapath interagem através da transferência direta dos dados, como também na realização de vários tipos de microoperações. Um datapath simples, baseado em barramento com quatro registradores, uma ULA e um deslocador (combinacional) é mostrado na Figura 1. Cada registrador está conectado a dois multiplexadores para formar os barramentos 1 e 2, que são as entradas para a ULA e o deslocador. As entradas de seleção (*ra1* e *ra2*) de cada multiplexador selecionam um registrador (*\$x*) para o barramento correspondente. Para o barramento 2, há um multiplexador adicional, MUX 2, tal que constantes possam ser trazidas para o interior do datapath através da entrada de constantes (Constant In, CI). O barramento 2 também se conecta a saída de dados (Data Out), para enviar dados para fora do datapath para outros componentes do sistema, como memória ou entrada-saída. Da mesma forma, o barramento 1 conecta a saída de endereços (Address Out) para enviar informações de endereço para fora do datapath para memória ou entrada-saída.

A unidade de controle para o datapath direciona o fluxo de informação através dos barramentos, a ULA, o deslocador e os registradores ao aplicar sinais às entradas de seleção. Por exemplo, para realizar a microoperação

$$\$1 \leftarrow \$2 + \$3$$

a unidade de controle deve fornecer valores de seleção para os seguintes conjuntos de entradas de controle:

1. “*Register Address 1 (ra1)*”, para colocar o conteúdo de *\$2* no barramento “1”;
2. “*Register Address 2 (ra2)*”, para colocar o conteúdo de *\$3* no barramento “2”, aplicando este valor na entrada “0” do MUX 2;
3. “*G select*”, para selecionar a operação $A + B$, a ser realizada pela ULA;
4. “*MF select*”, para colocar a saída da ULA na saída do MUX F;
5. “*MD select*”, para colocar a saída do MUX F no barramento D;
6. “*Write Address (wa3)*”, para selecionar *\$1* como o destino dos dados no barramento D;
7. “*Write Enable (we3)*”, para habilitar um registrador, neste caso *\$1*, para ser carregado.

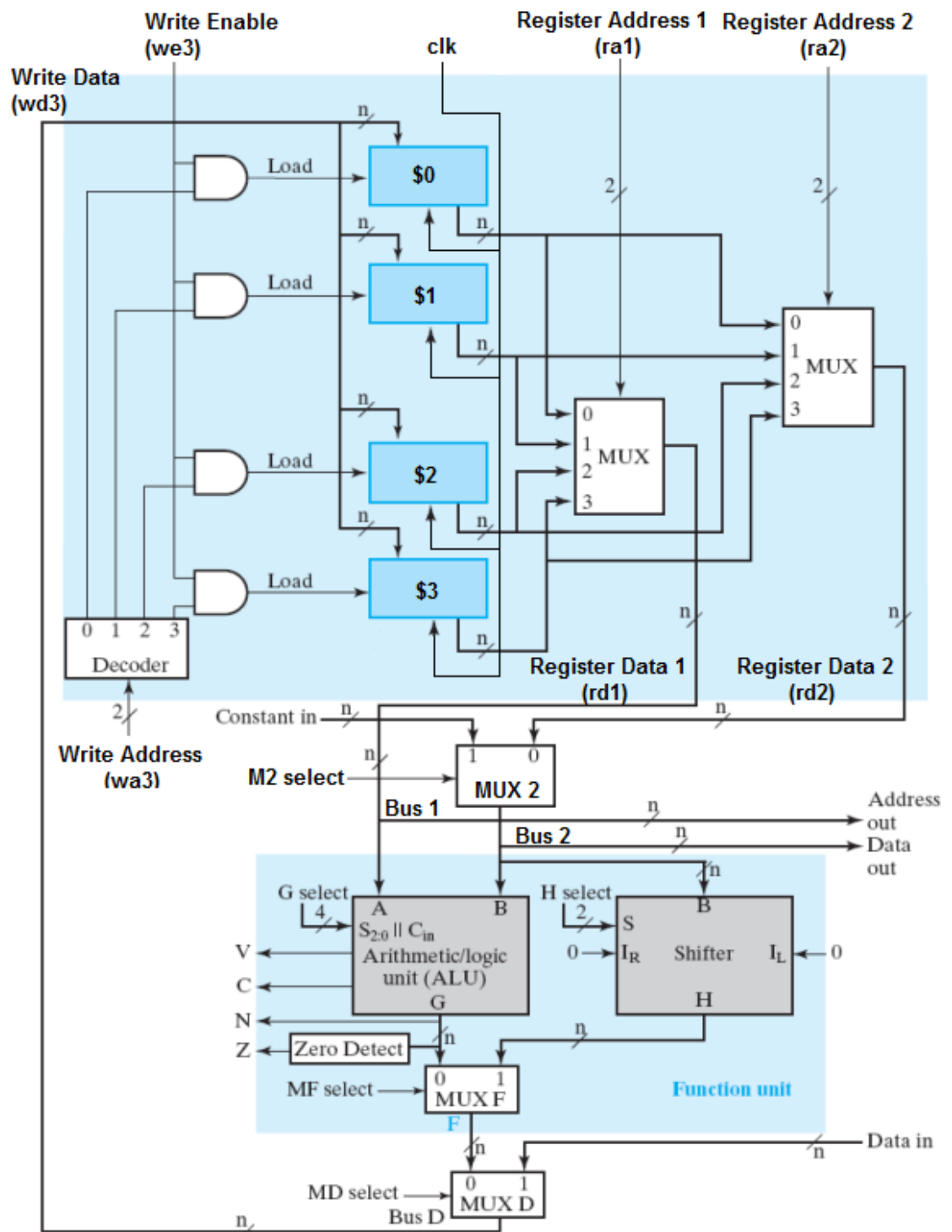


Figura 1. Diagrama de um datapath com um banco de 4 registradores

Concatenação de vetores de bits

Para complementar a facilidade com que Verilog trabalha com vetores de bits, existe o operador de concatenação. Através deste operador é possível definir vetores de bits formados por bits individuais ou partes de vetores de bits. A sintaxe utilizada é o uso de chaves. Por exemplo, assumindo que **b** seja um vetor de 8 bits [7:0] e **a** um sinal de um único bit, a concatenação {b[2:0], a} representa um vetor de 4 bits, com a seguinte formação (b₂ b₁ b₀ a) e

{a, b} é um vetor de 9 bits (a b₇ b₆ b₅ b₄ b₃ b₂ b₁ b₀). A concatenação pode ser utilizada no lado direito e no lado esquerdo de atribuições contínuas e de atribuições procedurais.

Um exemplo interessante é o uso do operador para realizar a operação de deslocamento para esquerda ou para a direita. Na descrição abaixo, considere que na entrada “in” de 8 bits entra o número b₇ b₆ b₅ b₄ b₃ b₂ b₁ b₀. A saída “out” será atribuída o número cujo bit mais significativo será “0” e os demais sete bits serão os bits b₇ b₆ b₅ b₄ b₃ b₂ b₁ da entrada “in”, formando assim o número 0 b₇ b₆ b₅ b₄ b₃ b₂ b₁ de 8 bits.

```
module desloca_direita (out, in);  
  
input [7:0] in;  
output [7:0] out;  
  
assign out = {1'b0, in[7:1]};  
  
end
```

Operadores de deslocamento

Verilog possui os operadores de deslocamento de bits para a esquerda e para a direita (<< e >>, respectivamente). Esses operadores operam como na linguagem C. Por exemplo, supondo que “A” seja um registrador de 8 bits armazenando o valor 40 em decimal, 0010_1000 em binário, após a operação A = A << 1, o valor de A será 0101_0000, ou 80 em decimal.

Modularidade e Instanciação

Quando trabalhamos com descrição de hardware normalmente utilizamos módulos distintos para funções distintas. A modularidade permite o reuso dos módulos através da instanciação. Por exemplo, na descrição Verilog a seguir é descrito um Flip-Flop tipo T, que apresenta duas entradas, “clk” e “reset” e uma saída “q”. A descrição representa o dispositivo mostrado na figura 2.

O uso do módulo T_FF no nosso módulo *top*, módulo “Mod_Teste” pode ser feito através do uso de uma instância do módulo T_FF, que é implementada inserindo em uma linha do módulo “Mod_Teste”, fora de qualquer *always* ou *initial*, o nome do módulo, “T_FF”, seguido por um nome de instância, seguidos pela lista de argumentos.

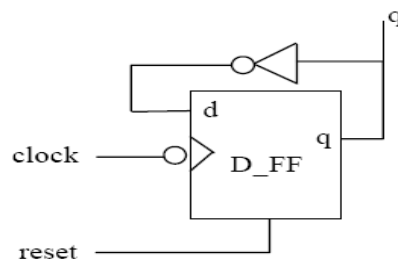


Figura 2 – Flip-Flop tipo T

```
module T_FF(q, clk, reset);  
// descrição do flip flop tipo T  
output q;  
input clk, reset;
```

```

        wire d;
        D_FF dff0(q, d, clk, reset); // instância de D_FF
        not n1(d, q)                 // a porta inversora é uma porta
                                     // básica do Verilog
    endmodule

end

```

A sua instanciação em um módulo local seria

```

module Mod_Teste (... ); // onde eu vou utilizar o flip-flop

wire soma;
reg [7:0] dado;

...
T_FF t1(dado[7], clock, clear);

...

endmodule

```

Foi criada uma instância de “*T_FF*” chamada “*t1*”, onde a saída *q* do Flip-Flop foi ligada ao *dado[7]*, a entrada “*reset*” foi ligada ao sinal “*clear*”, e a entrada “*clk*” foi ligada ao sinal “*clock*”. Esta ligação é caracterizada pela sequência em que as portas de entrada e saída são definidas na descrição do módulo. Como isto pode levar a erros, normalmente utiliza-se o formato mais moderno em que as ligações são explícitas (sintaxe com ponto).

```

module Mod_Teste (... ); // onde eu vou utilizar o flip-flop

wire soma;
reg [7:0] dado;

...
T_FF t1(.q(dado[7]), .clk(clock), .reset(clear);

...

endmodule

```

Desta forma, no laboratório **devemos** então descrever módulos que serão instanciados pelo módulo “*Mod_Teste*”. Este módulo “*Mod_Teste*” é o único que é específico para a placa do laboratório. Os módulos criados podem então ser reutilizados nos laboratórios seguintes.

Questões de Preparação

1. Descreva em Verilog um módulo com um deslocador combinacional para a esquerda de 8 bits. Acrescente uma saída de um bit que recebe o valor que seria deslocado para fora. Utilize a concatenação de vetores de bits.
2. Descreva em Verilog um registrador de deslocamento de 8 bits. Este registrador possui uma entrada clock, sensível a borda de descida, os sinais de controle *clear*, *load* e *shift*, todos ativos baixos, uma entrada e uma saída de 8 bits e uma saída de 1 bit, para o bit

que corresponderá ao vai-um (*carry*). O registrador deverá operar como indicado abaixo. As operações ocorrerão SEMPRE na borda de descida do *clock*:

- Sinal Clear* ativo: Conteúdo é apagado, independente dos sinais *load* e *shift*;
 - Sinal Clear* inativo e *sinal load* ativo: Conteúdo de 8 bits na entrada do registrador é armazenado no mesmo;
 - Sinal Clear* inativo, *sinal load* inativo e *sinal shift* ativo: Conteúdo do registrador é deslocado um bit para a direita;
3. Implemente em verilog um registrador de rotação para a direita de 8 bits. A operação é demonstrada na figura abaixo. Utilize operadores de deslocamento.

Antes:

B7							B0
----	--	--	--	--	--	--	----

Depois:

B0	B7						B1
----	----	--	--	--	--	--	----

4. Explique o que a descrição abaixo realiza.

```
module bancoreg (dado_ent, select, clk);
    input [31:0] dado_ent;
    input [1:0] select;
    input clk;

    reg [31:0] banco [3:0];

    always @(posedge clk)
        case (select)
            2'b00: banco[0] <= dado_ent;
            2'b01: banco[1] <= dado_ent;
            2'b10: banco[2] <= dado_ent;
            2'b11: banco[3] <= dado_ent;
        end
    end
endmodule
```

- Faça um módulo em verilog, completamente combinacional, com 8 entradas de 16 bits cada, e a partir de 3 bits de seleção, selecione o dado de uma das 8 entradas, colocando-o na saída.
- Faça um módulo em verilog, completamente combinacional, com uma entrada de 16 bits e 4 saídas 16 bits. A partir de 2 bits de seleção, ponha o conteúdo da entrada de 16 bits na saída selecionada pelos bits de seleção.
- Pesquise e responda se em Verilog, Verilog 2001, Verilog 2005 ou SystemVerilog é permitida a indexação direta de um banco de registradores.

8. Faça uma descrição de hardware para o módulo de registradores indicado na figura 3, considerando que “n”, que aparece no digrama de blocos do módulo, representa barramentos de 8 bits.

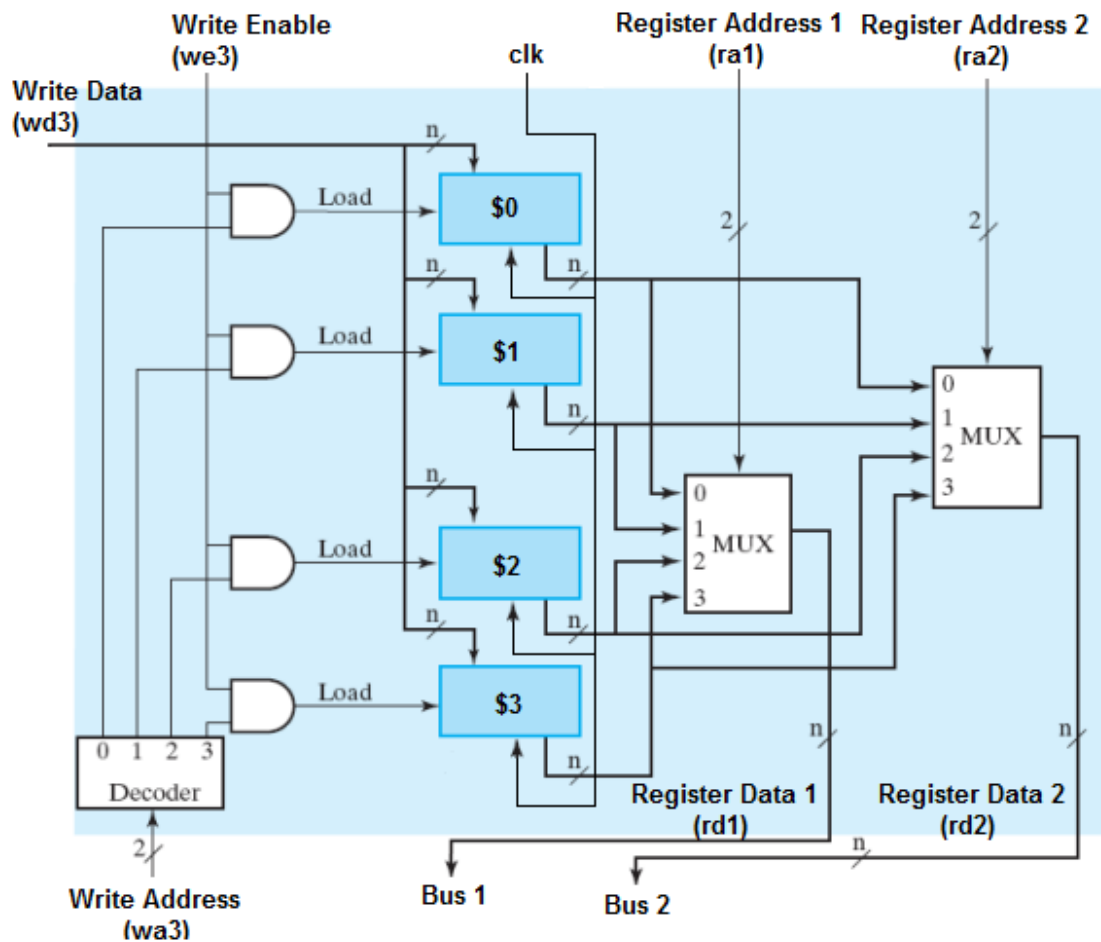


Figura 3 – Módulo de registradores de 8 bits

- 9 – Descreva um procedimento que permita avaliar o funcionamento do módulo gerado, considerando que as portas de entrada e saída do módulo poderiam ser conectadas aos recursos do módulo de testes “Mod_Teste”, implementado na prática 1 do Lab. de Arquitetura de Sistemas Digitais.

PREPARAÇÃO HDL_2

Implementação do *datapath* de um processador, parte II

OBJETIVOS

- Conhecer as funções realizadas pela Unidade Lógica/Aritmética (ULA);
- Implementar uma ULA completamente combinacional;
- Compreender os mecanismos de geração de circuitos combinacionais a partir de uma descrição com bloco always procedural;
- Compreender as questões de projeto de um datapath – compartilhamento de recursos e desempenho
- Complementar o datapath iniciado no experimento anterior
- Verificar o funcionamento de um datapath, realizando diversas operações manualmente.

INTRODUÇÃO

Toda operação em um *datapath* é realizada como uma transferência de dados. O caso mais comum é a transferência entre registradores. Esta transferência pode ser realizada sem que haja nenhuma modificação dos dados, quando dizemos que estamos realizando uma operação de movimentação (*move*, em inglês). Também podemos efetuar as transferências modificando os dados e ou utilizando mais de uma fonte de dados. Todas estas transferências, alterando ou não os dados, chamamos de microoperações. As alterações nos dados podem ser de diversos tipos. Normalmente nos processadores as operações mais comuns são aritméticas e lógicas. Estas operações são realizadas por um circuito dedicado denominado Unidade Lógica/Aritmética, ou ULA. As operações realizadas pela ULA, juntamente com o restante da arquitetura, estão intimamente ligadas com as instruções realizadas pelo processador. Como exemplos de operações aritméticas encontramos soma e subtração de valores inteiros com e sem sinal. As operações lógicas comuns são AND, OR e XOR, realizadas entre os bits individuais dos operandos.

O conceito de tamanho da palavra (*word*) de um processador está associado ao tamanho em bits dos registradores, dos barramentos e das operações realizadas pela ULA. Alguns processadores possuem mecanismos para utilizar palavras menores do que o *default*. Por exemplo, muitos processadores com palavras de 32 bits podem operar com dados de 16 bits (*half word*) e 8 bits (*char*).

Neste experimento será implementada uma ULA completamente combinacional com palavras de 8 bits. A especificação com as operações que devem ser realizadas, juntamente com a codificação de bits de cada operação será apresentada adiante. De posse desta implementação da ULA e da implementação do banco de registradores do experimento anterior podemos concluir o *datapath* do processador em estudo. O funcionamento deste

datapath pode ser verificado manualmente, realizando-se diversas operações, através da aplicação dos sinais de controle e valores dos dados nas chaves da placa DE2 e observação dos resultados através dos LEDs ou *displays* de 7 segmentos.

UNIDADE LÓGICA/ARITMÉTICA

A ULA é um circuito combinacional que efetua um conjunto de microoperações aritméticas e lógicas básicas. Ela possui algumas linhas de seleção usadas para determinar a operação a ser realizada. As linhas de seleção são decodificadas dentro da ULA, de tal forma que k linhas de seleção podem especificar até 2^k operações distintas.

A Figura 1 mostra o símbolo para uma ULA típica de 8 bits. As 8 entradas de *SrcA* são combinadas com as 8 entradas de *SrcB* para gerar o resultado de uma operação nas saídas *ULAResult*. A entrada de seleção de modo *ULAControl* especifica as cinco possíveis operações lógicas ou aritméticas.

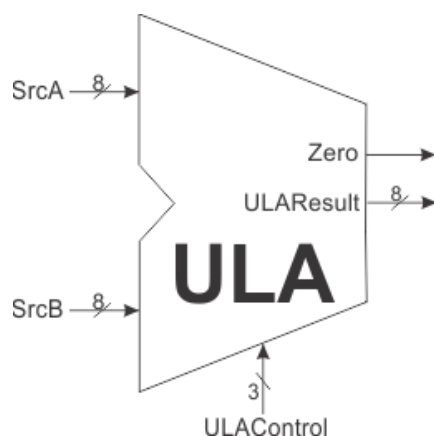


Figura 1. Símbolo para uma ULA de 8 bits.

CIRCUITO ARITMÉTICO

O componente básico de um circuito aritmético é um somador paralelo, o qual é construído com alguns circuitos somadores completos conectados em cascata, como mostrado na Figura 2. Ao controlar as entradas de dados do somador paralelo, é possível obter diferentes tipos de operações aritméticas. O diagrama de blocos na Figura 3 demonstra a configuração em que um conjunto de entradas do somador paralelo é controlado pelas linhas de seleção S_1 e S_0 . Há n bits no circuito aritmético com duas entradas A e B e saída G . Os n bits da entrada B passam pela lógica de entrada "*B input logic*" indo para a entrada Y do somador paralelo. A entrada de *carry* C_{in} vai para a entrada do somador completo na posição do bit menos significativo. A saída de *carry* C_{out} vem do somador completo, da posição do bit mais significativo. A saída do somador paralelo é calculado como

$$G = X + Y + C_{in}$$

onde X é o número de n bits da entrada “A” e Y é o número de n bits gerado pela lógica de entrada “B input logic”. C_{in} é a entrada de *carry*, que pode ser 0 ou 1. Note que o símbolo + na equação denota soma aritmética.

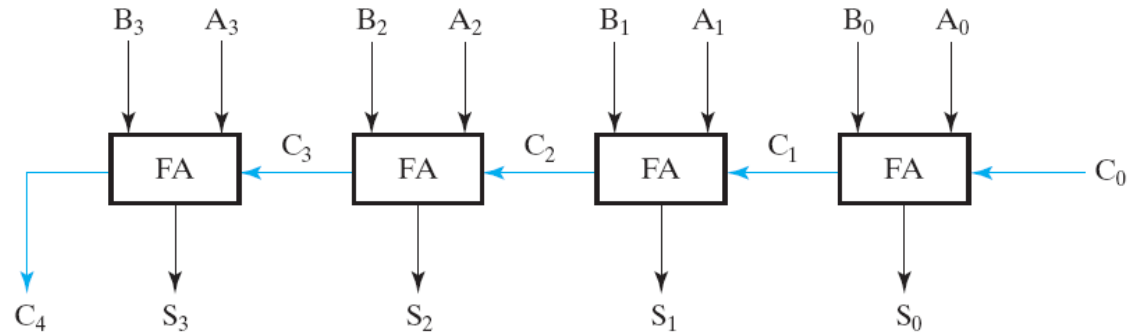


Figura 2. Somador em onda paralelo completo

A Tabela 1 mostra as operações aritméticas que podem ser obtidas ao controlar o valor de Y com as duas entradas de seleção S_1 e S_0 . Se os dados da entrada B são ignorados e os dados na entrada Y do somador completo forem forçados para o estado 0 (zero), a saída da soma torna-se $G = A + 0 + C_{in}$. Isto resulta $G = A$ quando $C_{in} = 0$ e $G = A + 1$ quando $C_{in} = 1$. No primeiro caso, temos uma transferência direta da entrada A para a saída G . No segundo caso, o valor A é incrementado de 1. Para uma soma aritmética normal é necessário aplicar B na entrada Y do somador paralelo. Obtemos $G = A + B$ quando $C_{in} = 0$. A subtração aritmética é obtida ao aplicarmos o complemento da entrada B na entrada Y do somador paralelo e quando $C_{in} = 1$, obtendo-se $G = A + \bar{B} + 1$. Isto resulta em A mais o complemento de 2 de B , o que é equivalente a subtração em complemento de 2. Tudo em 1 é a representação em complemento de 2 para -1. Logo, aplicando-se tudo em 1 na entrada Y , com $C_{in} = 0$ produz a operação de decremento $G = A - 1$.

Select		Input	$G = (A \text{ } Y \text{ } C_{in})$	
S_1	S_0	Y	$C_{in} = 0$	$C_{in} = 1$
0	0	all 0s	$G = A$ (transfer)	$G = A + 1$ (increment)
0	1	B	$G = A + B$ (add)	$G = A + B + 1$
1	0	\bar{B}	$G = A + \bar{B}$	$G = A + \bar{B} + 1$ (subtract)
1	1	all 1s	$G = A - 1$ (decrement)	$G = A$ (transfer)

Tabela 1. Funções do Circuito Aritmético

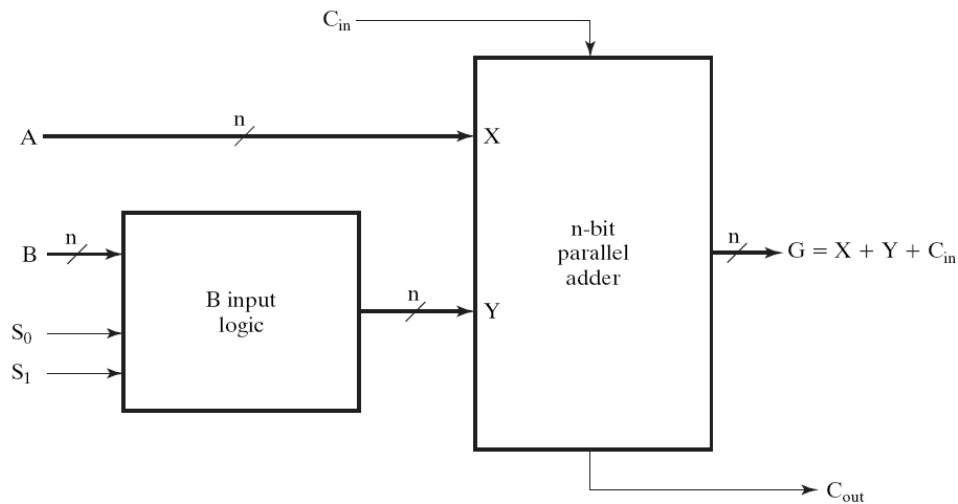


Figura 3. Diagrama de blocos de um Circuito Aritmético

IMPLEMENTAÇÃO EM VERILOG

A implementação em HDL pode ser feita de forma completamente estrutural, utilizando o circuito acima como base. Pode também ser feita utilizando a capacidade de geração de circuitos aritméticos do sintetizador. No primeiro caso, temos um controle maior do circuito que será obtido. No segundo caso, a descrição é muito mais simples.

A utilização dos operadores $+$ e $-$ em Verilog requer alguns cuidados. Em primeiro lugar, o resultado destes operadores em vetores de n bits produz resultados de $n + 1$ bits. Este bit adicional produzido é o *carry* do valor mais significativo. Ele pode ser capturado se realizarmos a operação como $\{C, F\} = A + B$, onde C é o bit de *carry*, F é o resultado da soma e A e B são operandos. Deve-se lembrar que as chaves realizam a concatenação de vetores de bits. Outra observação que deve ser considerada é que os vetores de bits representam inteiros sem sinal em Verilog. Este fato é mais relevante quando realizamos simulações.

CIRCUITO LÓGICO

As microoperações lógicas manipulam os bits dos operandos ao tratar cada bit de um registrador como uma variável binária, realizando operações bit-a-bit. Há quatro operações lógicas usadas normalmente – AND, OR, XOR e NOT – das quais as outras podem ser convenientemente derivadas.

Na Figura 4 é mostrado um estágio do circuito lógico. Ele consiste de quatro portas lógicas e um multiplexador 4-para-1. A tabela da Figura 4b lista as operações lógicas obtidas para cada combinação dos valores de seleção.

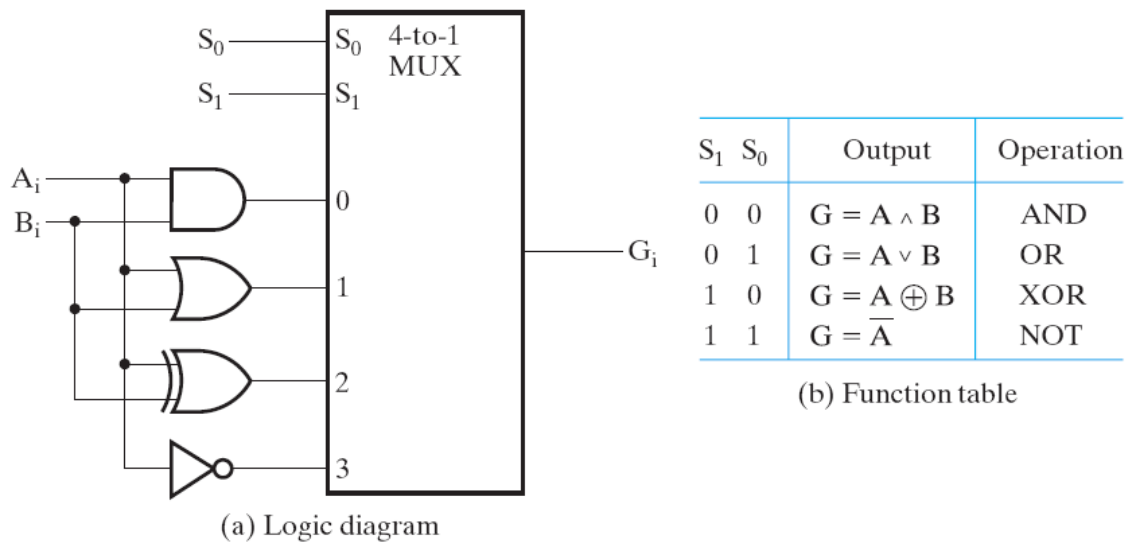


Figura 4. Um estágio do Circuito Lógico

UNIDADE LÓGICA/ARITMÉTICA

O circuito lógico pode ser combinado com o circuito aritmético para produzir uma ULA. Nessa atividade será implementada uma ULA com 5 operações selecionáveis através da entrada *ULAControl*. A Tabela 2 lista as suas operações.

Operação	ULAControl	ULAResult
Add	010	= SrcA + SrcB
Subtract	110	= SrcA + $\overline{\text{SrcB}}$ + 1
And	000	= SrcA & SrcB
Or	001	= SrcA SrcB
Set less than	111	1, se SrcA < SrcB

Tabela 2. Funções da ULA

REPRESENTAÇÃO DO DATAPATH

Na Figura 5 vemos uma representação do datapath em uma estrutura hierárquica, que reduz a complexidade aparente do datapath. Os dois módulos principais são o banco de registradores e a ULA. O banco de registradores foi implementado no experimento anterior e deverá ser reaproveitado. A ULA deve ser implementada neste experimento.

O banco de registradores é um tipo de memória rápida que permite que uma ou mais palavras sejam lidas e uma ou mais palavras sejam escritas simultaneamente. As entradas *ra1*, *ra2* e *wa3* são endereços. O endereço *wa3* seleciona em qual registrador o dado *wd3* será armazenado, assim como os endereços *ra1* e *ra2* definem quais registradores terão seus

conteúdos disponibilizados respectivamente nos barramentos *Bus1* e *Bus2*. Todos estes acessos ocorrem no mesmo ciclo de *clock*. A entrada *we3* corresponde ao sinal *Load Enable*. Quando em 1, o sinal de *we3* permite que os registradores sejam carregados (alterem seus valores) e quando em zero, evita qualquer alteração. O tamanho do banco de registradores é $2^m \times n$, onde m é número de bits de endereços e n é o número de bits por registradores. Utilizaremos neste experimento $m = 3$ e $n = 8$, em um total de 8 registradores de 8 bits.

A Unidade Lógica e Aritmética (ULA) é responsável pelas operações da Tabela 2. A seleção é feita pela entrada *ULAControl*. Os operandos vêm das entradas de dados conectadas aos barramentos *Bus1* e *Bus2* e o resultado *ULAResult* é encaminhado para o MUX D. É disponível também uma saída de status adicional que indica quando o resultado de alguma operação é zero.

FLAG Z – Indica se o dado gerado pela ULA é igual a zero. (dado igual a zero: Z=1; dado diferente de zero: Z=0);

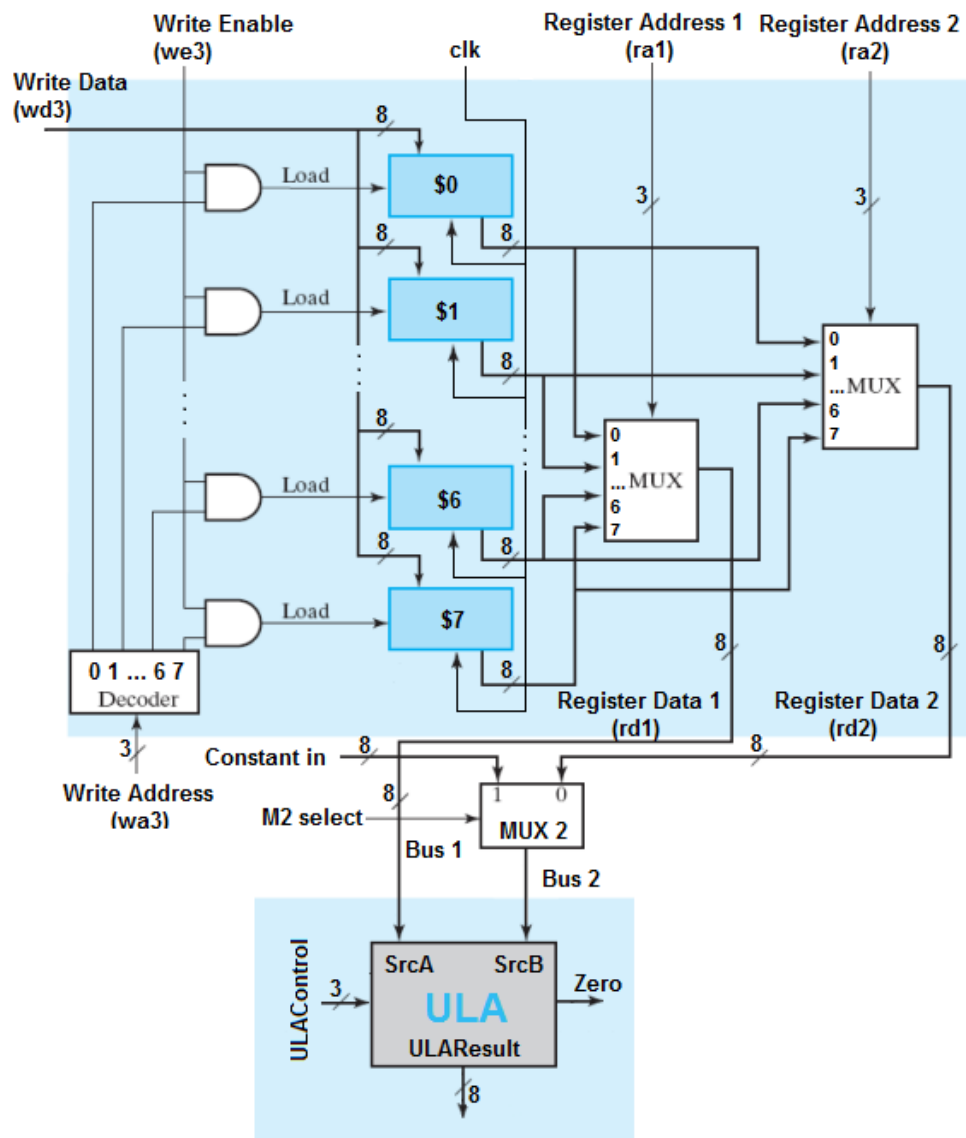


Figura 5. Diagrama de blocos do datapath usando Banco de Registradores e ULA

Questões de Preparação

1. Descreva em Verilog um somador completo (com *carry in* e *carry out*) paralelo de 8 bits de forma estrutural.
2. Descreva em Verilog o somador da questão anterior de forma procedural.
3. Implemente em Verilog a lógica que coloca o valor de todos os registradores de um banco de registradores em zero utilizando um laço *for*.
4. Explique como podemos utilizar a diretiva *parameter* de verilog para definirmos a quantidade de registradores e o tamanho em bits de um datapath. Que vantagens você vê em codificar um módulo parametrizável?
5. Faça a descrição em Verilog da ULA apresentada na figura 5 do guia. Considere que os operandos A e B são números de 4 bits.
6. Faça a instanciação do módulo ULA implementado no item 7 dentro no módulo “Mod_Testes”. Utilize os recursos de entrada do módulo “Mod_Testes” para gerar os sinais de controle e operandos de entrada. Utilize os recursos de saída do módulo “Mod_Testes” para visualização dos resultados das operações geradas pela ULA.
7. Descreva um procedimento, indicando os estados de chaves/botões do “Mod_Testes” e o que deve ser observado nos LED’s, Displays do mesmo módulo, de modo a se testar o funcionamento do módulo ULA implementado no item 7 e instanciado no item 8.

Implementação da unidade de controle de um processador

OBJETIVOS

- Conhecer as funções realizadas pela Unidade de Controle
- Implementar uma unidade de controle para o *datapath* estudado, completando um processador simples
- Entender a codificação de instruções em campos de bits
- Conhecer a sequência básica de execução de instruções e os mecanismos que alteram esta sequência, como também como estes afetam a implementação do contador de programa (*PC*)
- Compreender a formação de sinais de controle a partir das instruções armazenadas em memória
- Implementar uma memória ROM completamente combinacional através de uma descrição em Verilog
- Verificar o funcionamento de um processador através da execução de algumas instruções

INTRODUÇÃO

Neste experimento será implementada uma unidade de controle, que juntamente com o *datapath* desenvolvido nos experimentos anteriores, define um processador MIPS bastante simples, capaz de executar 10 instruções. Este processador, apesar de muito simples, possui algumas características típicas de projetos mais modernos, e servirá como base para o estudo de outros processadores existentes.

O funcionamento deste *processador* pode ser verificado manualmente, realizando-se diversas instruções, através da aplicação dos sinais de controle e valores dos dados nas chaves da placa DE2 e a observação dos resultados através dos LEDs e *displays* de 7 segmentos. Também podemos armazenar um programa em uma memória de instruções. Esta memória completamente combinacional pode ser criada em Verilog, utilizando um array de vetores de bits, ou utilizando uma descrição em Verilog similar a um decodificador (opção que será implementada neste experimento).

UMA ARQUITETURA DE UM COMPUTADOR SIMPLES

Está sendo introduzida uma arquitetura de um computador simples para obter um entendimento inicial sobre projeto de computadores e para ilustrar projetos de controle para sistemas programáveis. Em um sistema programável, uma parte da entrada do processador consiste de uma sequência de *instruções*. Cada instrução especifica uma operação para o sistema realizar, quais operandos usar para a operação, onde armazenar os resultados da operação e/ou, em alguns casos, qual instrução executar em seguida. Para um sistema programável, as instruções são normalmente armazenadas em memória, que pode ser RAM (memória de leitura e escrita) ou ROM (memória de leitura apenas). Para executar as

instruções em sequência, é necessário fornecer o endereço na memória da instrução a ser executada. Em um computador, este endereço vem de um registrador chamado de contador de programa (*PC*, do inglês *program counter*). Como o nome sugere, o *PC* tem lógica que permite que ele conte. Além disso, para mudar a sequência de operações usando decisões baseadas em informações de estado, o *PC* precisa de capacidade de carregamento paralelo. Assim, no caso de sistemas programáveis, a unidade de controle contém um *PC* e lógica de decisão associados, como também a lógica necessária para interpretar as instruções para poder executá-las. Executar uma instrução significa ativar a sequência de microoperações necessária para efetuar a operação especificada pela instrução.

ARQUITETURA DO CONJUNTO DE INSTRUÇÕES

O usuário especifica as operações a serem realizadas e sua sequência usando um programa, que é uma lista de instruções. O processamento de dados realizado pelo computador pode ser alterado ao especificar um novo programa com diferentes instruções ou ao especificar as mesmas instruções com dados diferentes. Instruções e dados são normalmente armazenados conjuntamente na mesma memória. Por meio de técnicas como memória cache, pode parecer que estes provêm de memórias diferentes. A unidade de controle lê uma instrução da memória, decodifica e executa a instrução ao lançar uma sequência de uma ou mais microoperações. A habilidade de executar um programa proveniente da memória é a propriedade singular mais importante de um computador de propósito geral. A execução de um programa armazenado em memória contrasta fortemente com o controle feito com máquinas de estado usuais, que executam operações sequenciadas apenas pelas entradas e sinais de estado.

Uma *instrução* é uma coleção de bits que instrui o computador a realizar uma operação específica. Chamamos a coleção de instruções de um computador seu *conjunto de instruções* e a descrição detalhada do conjunto de instruções a sua *arquitetura do conjunto de instruções* (ISA, do inglês *Instruction Set Architecture*). Arquiteturas de conjunto de instruções simples possuem três componentes principais: os recursos de armazenamento, os formatos de instruções e as especificações das instruções.

RECURSOS DE ARMAZENAMENTO

Os recursos de armazenamento para um computador simples são representados pelo diagrama da Figura 1. No diagrama é colocada a estrutura do computador como vista por um usuário que esteja programando em uma linguagem que especifique diretamente as instruções a serem executadas. Note que a arquitetura inclui duas memórias, uma para armazenar instruções e outra para armazenar os dados. Estas podem ser realmente memórias diferentes ou podem ser a mesma memória, mas vistas como diferentes do ponto de vista da CPU (devido ao uso de estruturas auxiliares, como memória cache). Também é visível ao programador, no diagrama, um banco de registradores com oito registradores de 8 bits e um contador de programa de 8 bits.

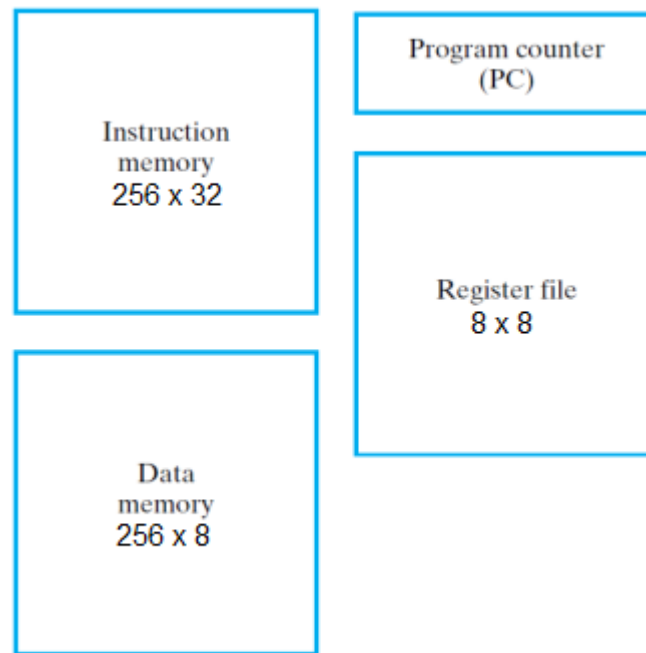


Figura 1. Diagrama com os Recursos de Armazenamento do Computador Simples

FORMATOS DE INSTRUÇÃO

O formato de uma instrução é normalmente descrito por uma caixa retangular simbolizando seus bits, como estas aparecem em palavras na memória ou em um registrador de controle. Os bits são divididos em grupos ou partes denominadas campos. A cada campo é atribuído um item específico, como o código da operação, um valor constante ou um endereço do banco de registradores. Os vários campos especificam diferentes funções para as instruções e, quando mostradas conjuntamente, constituem um formato de instrução.

O *código da operação* de uma instrução, normalmente denominado de “opcode” (do inglês, *operation code*), é o grupo de bits que especifica uma operação, tal como somar, subtrair, deslocar ou complementar. O número de bits requeridos pelo é uma função do número total de operações do conjunto de instruções. Ele deve consistir de pelo menos m bits para até 2^m operações distintas. O projetista atribui uma combinação de bits (um código) para cada operação. O computador é projetado para aceitar esta configuração de bits no momento apropriado na sequência de atividades e para fornecer uma sequência de palavras de controle (conjunto de sinais de controle) que executam a operação especificada. Como um exemplo específico, considere um computador com um máximo de 128 operações distintas, uma delas sendo uma operação de soma. O opcode atribuído para esta operação consiste de sete bits 0000010. Quando o opcode 0000010 é detectado pela unidade de controle, uma sequência de palavras de controle é aplicada ao datapath para realizar a soma pretendida.

O opcode de uma instrução especifica a operação a ser executada. A operação deve ser realizada usando dados armazenados nos registradores do computador ou em memória (o conteúdo de um dos recursos de armazenamento). Uma instrução, portanto, deve especificar não apenas a operação, mas também os registradores ou posição de memória em que os operandos são encontrados e onde o resultado deverá ser salvo. Os operandos podem ser especificados por uma instrução de duas formas:

- Um operando é dito ser especificado **explicitamente** se a instrução contém bits determinados para sua identificação. Por exemplo, a instrução que realiza uma soma pode conter três números binários especificando os registradores que contêm os dois operandos e o registrador que recebe o resultado.
- Uma operação é dita como definida **implicitamente** se é incluída como parte da definição da própria operação, como representada pelo opcode, ao invés de ser dado na instrução. Por exemplo, em uma operação de Incrementar Registrador, um dos operandos é implicitamente +1.

A arquitetura MIPS faz o compromisso de definir três formatos de instrução: Tipo-R, Tipo-I e Tipo-J. Esse pequeno número de formatos proporciona certa regularidade entre todos os tipos e, portanto, um hardware mais simples

- **Tipo Registrador:** Instruções tipo-R utilizam três registradores como operandos: dois como fonte e um como destino. A Figura 2 mostra o formato de instrução de máquina tipo-R. a instrução de 32-bits possui seis campos: op, rs, rt, rd, shamt e funct. Cada campo é de cinco ou seis bits, como indicado.

A operação que a instrução realiza é codificada nos dois campos destacados em azul: op (também chamado opcode, ou código de operação) e funct (também chamado de função). Todas as instruções tipo-R possuem um opcode de 0. A operação específica tipo-R é determinada pelo campo field. Por exemplo, os campos opcode e funct para a instrução add são 0 (0000002) e 32 (1000002), respectivamente. Similarmente, a instrução sub possui os campos opcode e funct de 0 e 34.

Os operandos são codificados em três campos: rs, rt e rd. Os primeiros dois registradores, rs e rt, são os registradores fonte; rd é o registrador de destino.

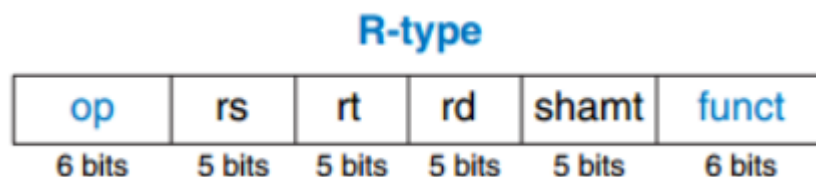


Figura 2. Instrução do tipo R

- **Tipo Imediato:** Instruções tipo-I utilizam dois operandos registradores e um operando imediato. A Figura 3 mostra o formato de instrução de máquina tipo-I. A instrução de 32-bits possui quatro campos: op, rs, rt e imm. Os primeiros três campos, op, rs e rt, são como aquelas instruções do tipo-R. o campo imm mantém um imediato de 16 bits.

A operação é determinada exclusivamente pelo opcode, destacado em azul. Os operandos são especificados em três campos, rs, rt e imm. rs e imm são sempre utilizados como operandos fonte. rt é utilizado como destino por algumas instruções (como addi e lw), mas também como outra fonte por outras (como sw).

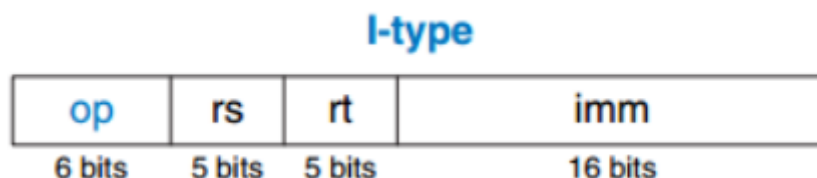


Figura 3. Instrução do tipo I

- **Tipo Jump:** O nome “tipo-J” é uma abreviação para “tipo salto (jump)”. Esse formato de instrução utiliza um único operando de endereço de 26-bits, *addr*, como mostrado na Figura 4. Assim como outros formatos, as instruções tipo-J iniciam-se com um opcode de 6 bits. Os bits remanescentes são utilizados para especificar o endereço, *addr*.

Em contraste com os outros dois formatos, instruções do tipo J não modificam nenhum conteúdo do banco de registradores ou da memória. Em vez disso, alteram a ordem em que as instruções são buscadas da memória, por meio da escrita no PC.

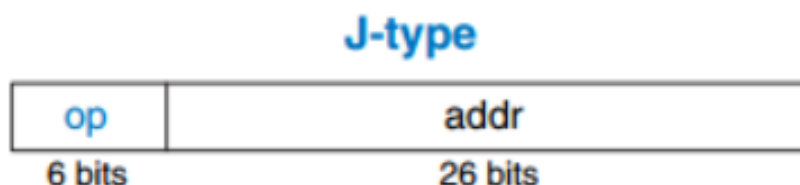


Figura 4. Instrução do tipo J

LINGUAGEM ASSEMBLY

A linguagem assembly é a representação legível para humanos da linguagem nativa dos computadores. Cada instrução da linguagem assembly especifica tanto a operação a ser realizada quanto o operando sobre o qual opera. Para cada instrução, o opcode é dado juntamente com um nome abreviado chamado de *mnemônico*, que indica a finalidade da instrução. O conjunto básico de instruções que serão implementadas no laboratório está especificado na Tabela 1

Instrução	Descrição	Algoritmo
ADD \$X, \$Y, \$Z	Adicionar	$\$X = \$Y + \$Z$
SUB \$X, \$Y, \$Z	Subtrair	$\$X = \$Y - \$Z$
AND \$X, \$Y, \$Z	AND Bit a bit	$\$X = \$Y \& \$Z$
OR \$X, \$Y, \$Z	OR Bit a bit	$\$X = \$Y \$Z$
SLT \$X, \$Y, \$Z	Menor que	$\$X = 1$ se $\$Y < \Z e 0 c.c.
LW \$X, i(\$Y)	Carregar da memória	$\$X \leftarrow \text{Cont. do end. } (\$Y + i)$
SW \$X, i(\$Y)	Armazenar na memória	$\text{End. } (\$Y + i) \leftarrow \X
BEQ \$X, \$Y, i	Desviar se igual	Se $\$X == \Y , $\text{PC} = \text{PC} + 1 + i$
ADDi \$X, \$Y, i	Adicionar Imediato	$\$X = \$Y + i$
J i	Desvio incondicional	$\text{PC} = i$

Tabela 1. Conjunto básico de instruções MIPS.

MEMÓRIA

Serão implementadas duas memórias distintas nessa atividade de laboratório. Uma memória de instruções de 32-bits e uma memória de dados de 8-bits

A memória de instrução tem uma única porta de leitura. Recebe uma entrada de endereço de 8-bits, A, e escreve as instruções de 32 bits na saída de dados de leitura, RD. É ilustrado na Figura 5 um exemplo de programa armazenado em uma memória de instruções. Perceba que o código assembly é convertido para código de máquina antes de ser armazenado.

A memória de dados tem uma única porta de leitura/escrita. Se o enable de escrita, WE, estiver em 1, escreve os dados WD no endereço A na subida do clock. Se o enable de escrita estiver em 0, ele lê o endereço A para RD.

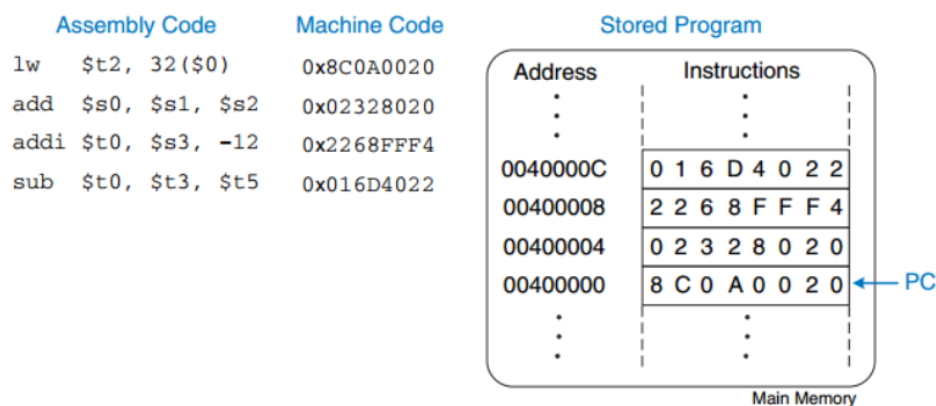


Figura 5. Programa armazenado em uma memória de instruções.

Neste ponto, é vital reconhecer a diferença entre uma *operação* do computador e uma *microoperação* do hardware. Uma operação é especificada por uma instrução armazenada em binário na memória do computador. A unidade de controle do computador utiliza o endereço ou endereços fornecidos pelo contador de programa para recuperar a instrução da memória. Ela então decodifica os bits do opcode e outras informações na instrução para realizar as microoperações necessárias para a execução da instrução. Diferentemente, uma microoperação é especificada pelos bits que formam uma palavra de controle, que determinará ao hardware do computador as ações que devem ser implementadas para executar a microoperação. A execução de uma operação do computador normalmente requer uma sequência ou programa de microoperações no lugar de uma única microoperação.

CONTROLE EM LÓGICA PARA CICLO ÚNICO

O diagrama de blocos de um processador MIPS de ciclo único é ilustrado na Figura 6. Em outras palavras, o processador busca e executa uma instrução em um único ciclo de clock. A unidade de controle é o elemento que faltava para enfim completarmos a CPU do laboratório.

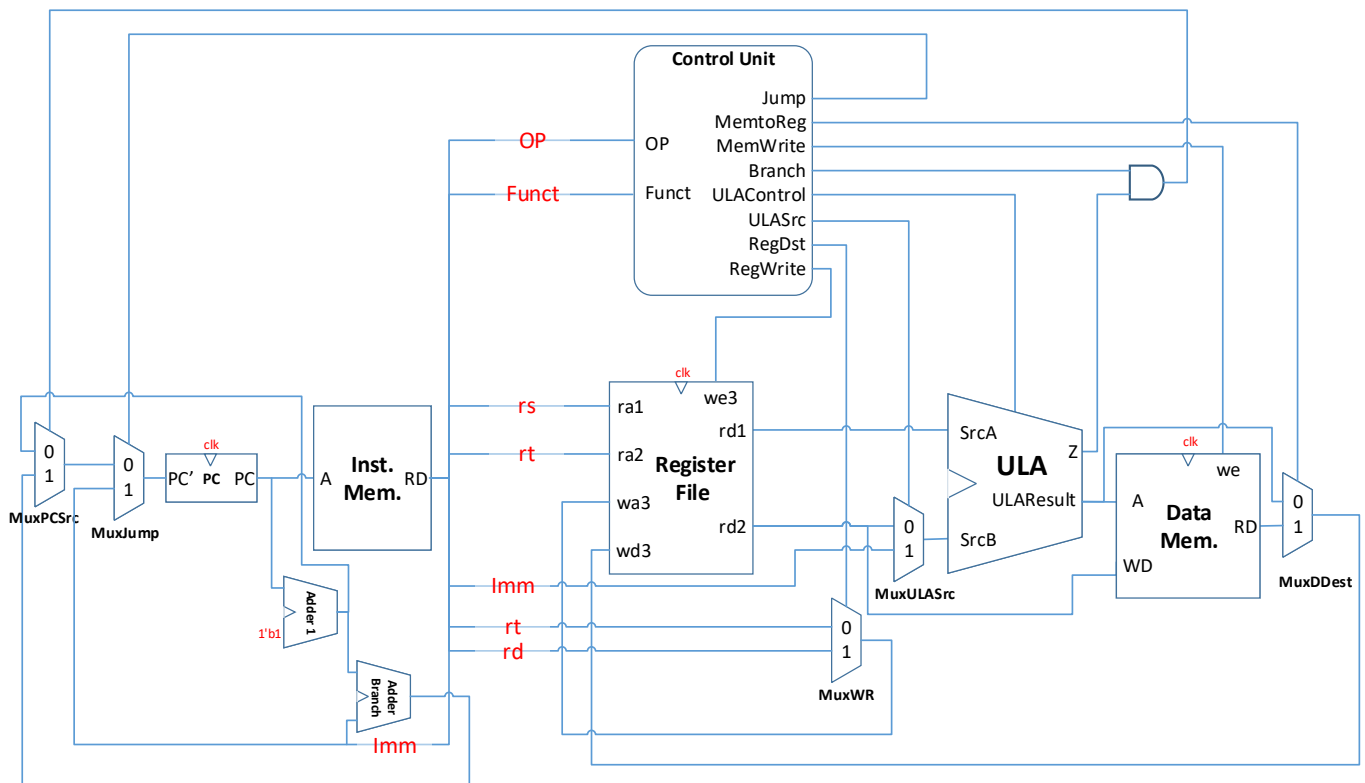


Figura 6. Diagrama de Blocos de um processador MIPS de Ciclo Único

Como discutido previamente, o PC fornece o endereço da instrução para a memória de instruções e a saída de instruções da memória vai para a lógica de controle, que neste caso é o decodificador de instruções.

O PC é atualizado em cada ciclo de clock. O comportamento do PC, que é um registrador complexo, é determinado pelo opcode, e pelo bit de estado Z. O PC opera como contador com carga de valor inicial. Se ocorre um salto, o novo PC torna-se o valor do campo imediato da instrução. Se um desvio é implementado, então o novo valor do PC é a soma do PC anterior e o deslocamento de endereço com sinal (representação em complemento de 2) acrescido de 1. Caso contrário, o PC é somente incrementado de 1.

Observe que não há lógica seqüencial na parte de controle além do PC. Logo, afora fornecer o endereço para a memória de instruções, a lógica de controle é combinacional. Este fato, combinado com a estrutura do datapath e o uso de memórias de instruções de dados separadas, permitem que o computador de ciclo único obtenha e execute uma instrução da memória de instruções, tudo em um único ciclo de clock.

DECODIFICADOR DE INSTRUÇÕES

O decodificador de instruções (unidade de controle) é um circuito combinacional que fornece todas as palavras de controle para o datapath, baseado nos conteúdos dos campos da instrução. É ilustrado na Tabela 2 o valor dos bits de controle para cada instrução (conjunto OP + Funct)

Instr	OP	Funct	RegWrite	RegDst	ULASrc	ULAControl	Branch	MemWrite	MemtoReg	Jump
ADD	000000	100000	1	1	0	010	0	0	0	0
SUB	000000	100010	1	1	0	110	0	0	0	0
AND	000000	100100	1	1	0	000	0	0	0	0
OR	000000	100101	1	1	0	001	0	0	0	0
SLT	000000	101010	1	1	0	111	0	0	0	0
LW	100011	xxxxxx	1	0	1	010	0	0	1	0
SW	101011	xxxxxx	0	x	1	010	0	1	x	0
BEQ	000100	xxxxxx	0	x	0	110	1	0	x	0
ADDi	001000	xxxxxx	1	0	1	010	0	0	0	0
J	000010	xxxxxx	0	x	x	xxx	x	0	x	1

Tabela 2. Tabela do decodificador de instruções.

IMPLEMENTAÇÃO EM VERILOG

A implementação em HDL do circuito da unidade de controle pode ser realizada de diversas formas. Mantendo o alto nível de abstração a unidade de controle deve ser codificada de maneira trivial por meio de um “case”.

O módulo do PC é um circuito sequencial que guarda um vetor de 8 bits cada vez que uma borda de subida do clock é detectada. Usar “always @(posedge clock)”.

Todos os outros elementos da CPU já foram implementados em aulas passadas e devem ser reaproveitados.

MEMÓRIA COMBINACIONAL

Para que este processador opere com um único ciclo de clock, é necessário que a memória de instruções se comporte como um circuito puramente combinacional. Neste experimento, isto é exatamente o que vai ser feito. Uma memória de apenas leitura combinacional é basicamente um decodificador que tem os endereços como entrada e os dados como saída. A sua implementação é feita da mesma forma que o decodificador de sete segmentos implementado nos experimentos anteriores. Deve-se lembrar que, como o número de bits de endereço é muito grande (8) haverão diversos códigos não definidos, que precisam ter valor “default”.

Esta memória deve ser definida como um módulo separado porque nos próximos experimentos ela será implementada usando uma ferramenta específica do Quartus. Para isto, é preciso também que o módulo da CPU possua uma saída adicional de endereços (Code Addr) e uma entrada para as instruções (Inst In).

MODULARIDADE E INSTANCIAÇÃO

Como explicado nos experimentos anteriores, o módulo “Mod_Teste”, que é o módulo TOP, deve ser reservado para os testes na placa. Ele não faz parte da definição da CPU. Ele instancia a CPU e as outras partes necessárias para o teste desta. Deve-se implementar a CPU de forma que ela tenha todas as suas partes constituintes instanciadas internamente. Esta CPU terá como interface com o mundo exterior os sinais de CLOCK, RESET e os Sinais de interface

com as memórias. Outras portas podem ser acrescentadas apenas para depuração, como por exemplo, portas que permitam a visualização dos registradores.

O módulo TOP ("Mod_Teste") é o único que deve conhecer detalhes da placa, ou seja, nomes de chaves e de LEDs. Este módulo instancia a CPU, a memória e os circuitos de teste e os conecta nestas interfaces da placa.

Questões de Preparação

1. Quais as conexões que devem ser feitas entre a unidade de controle e o datapath, de acordo com a Figura 6?
2. Implemente em Verilog o decodificador de instruções da Figura 6. Este decodificador transforma os 32 bits de uma instrução nos 10 bits da palavra de controle.
3. Determine o código binário do conjunto de instruções abaixo:
ADD \$2, \$0, \$1 SUB \$3, \$2, \$1 AND \$2, \$1, \$3
OR \$2, \$1, \$3 SLT \$1, \$0, \$2 LW \$1, 9(\$0)
SW \$1, 9(\$0) BEQ \$X, \$Y, i ADDi \$2, \$0, 7
J 6
4. Descreva em Verilog uma memória de instruções de 256 posições. Deve existir uma entrada de endereços e uma saída de instruções de 32 bits.
5. Analise o seguinte trecho de código e informe o valor dos registradores \$1 e \$2 quando PC = 5:

Posição	Instrução
0000	ADDi \$1, \$0, 5
0001	ADDi \$2, \$0, 0
0002	ADDi \$2, \$2, 1
0003	BEQ \$1, \$2, 1
0004	J 2
0005	FIM

6. Escreva um programa de teste para esta CPU e converta cada instrução para o equivalente em binário. Este programa deve testar as diversas classes de instrução da CPU. Por exemplo, uma instrução para teste de operações com registradores e operações de desvio e jump;

PREPARAÇÃO HDL_4

Implementação de Memória de Instrução a partir de uma ROM

OBJETIVOS

- Executar um programa de forma automática, ou seja, com sinal de clock real;
- Aprender a criar um sinal de clock de baixa frequência a partir de um sinal de alta frequência utilizando um contador como divisor de frequência digital;
- Utilizar o Editor de Conteúdo da Memória para modificar o programa após a programação do FPGA.
- Verificar a criação de uma memória ROM utilizando a ferramenta de configuração de blocos com “mega-funções” do FPGA;

INTRODUÇÃO

Neste experimento serão implementadas memórias de instrução e dados através de uma ferramenta gráfica do Quartus II. A memória de instruções que foi descrita como um circuito combinacional será modelada como uma memória ROM. Por sua vez a memória de dados será implementada como uma RAM.

Para que a CPU execute um programa de forma autônoma, é preciso que um sinal de clock periódico seja aplicado na sua entrada. Até agora, foi utilizado um botão para aplicar os pulsos de clock. Neste experimento esse sinal será gerado pelo oscilador a cristal da placa.

É ilustrado na Figura 1 o diagrama da CPU e das memórias de instrução e dados.

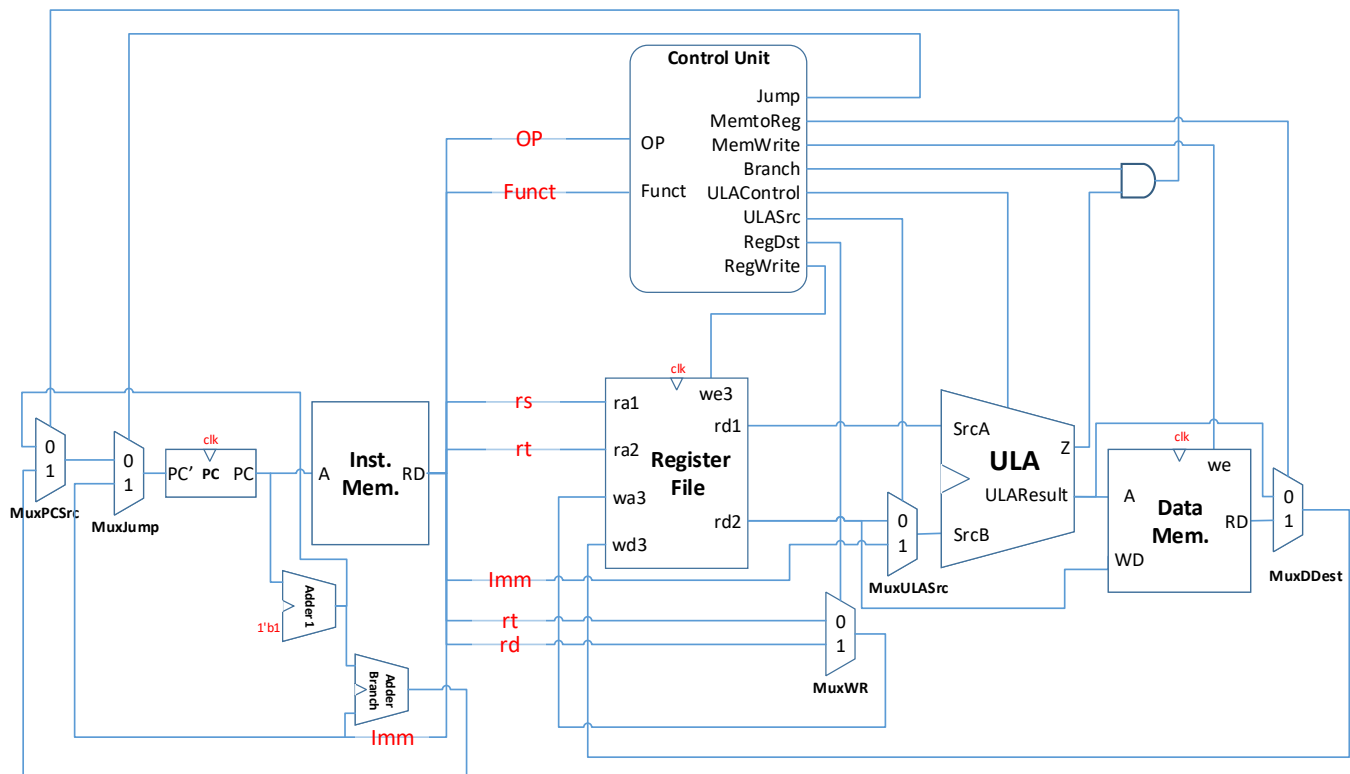
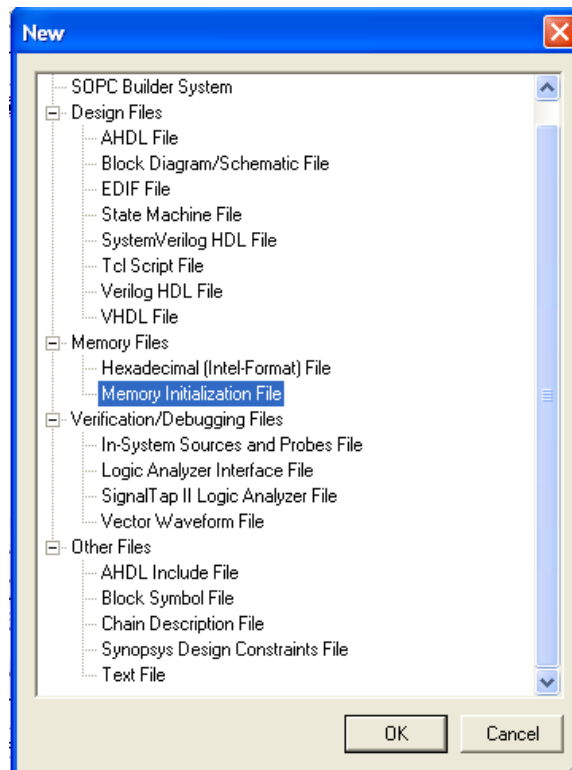


Figura 1. CPU, memórias de instruções e de dados.

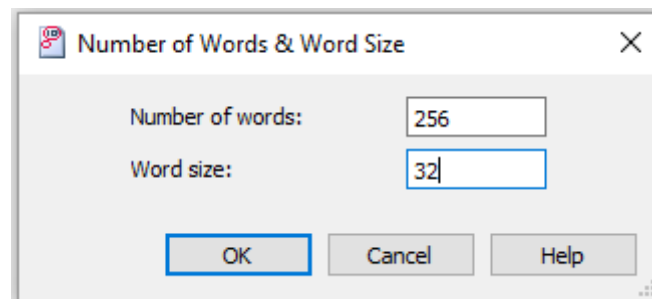
INICIALIZAÇÃO DE MEMÓRIA ROM

A memória ROM que será implementada na sequência deverá ser gravada com as instruções do programa a ser executado, descrito na tarefa de laboratório. No processo de configuração da memória, a mesma será inicializada a partir de um arquivo denominado "MemInstrucao.mif". Na sequência serão apresentados os passos para geração desse arquivo.

Na tela principal do Quartus II, selecione a opção <File->New->Memory Initialization File>. Será apresentada a seguinte tela:



Após selecionar e clicar em ok, será apresentada a caixa de configuração abaixo, onde devem ser definidas as informações de tamanho de posições da memória (“Number of words”) e número de bits de cada posição (“Word size”). Para os mesmos defina os mesmos valores usados na configuração da memória: 256 posições de memória e 32 bits por posição:

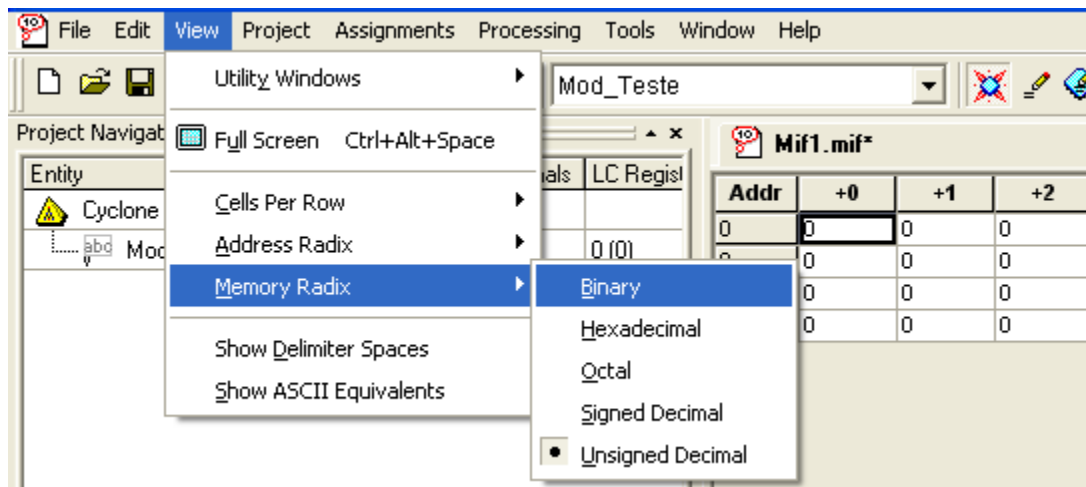


Após definir os valores e clicar em ok, será apresentada a tela a seguir na área de edição do Quartus II. Observe que genericamente o Quartus II define um nome para o arquivo “Mif1.mif” e define todos os valores das posições de memória iguais a zero.

Mif1.mif*								
Addr	+0	+1	+2	+3	+4	+5	+6	+7
0	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0
16	0	0	0	0	0	0	0	0
24	0	0	0	0	0	0	0	0

Altere a visualização dos dados para o formato binário, de modo a que possa definir os valores para as posições de memória de acordo com o código binário das instruções, descritas na

tarefa de laboratório. Para isso selecione a opção <View->Memory Radix->Binary>, conforme indicado na figura abaixo.



Após fazer essa alteração os dados das posições de memória serão apresentados conforme indicado na tela abaixo.

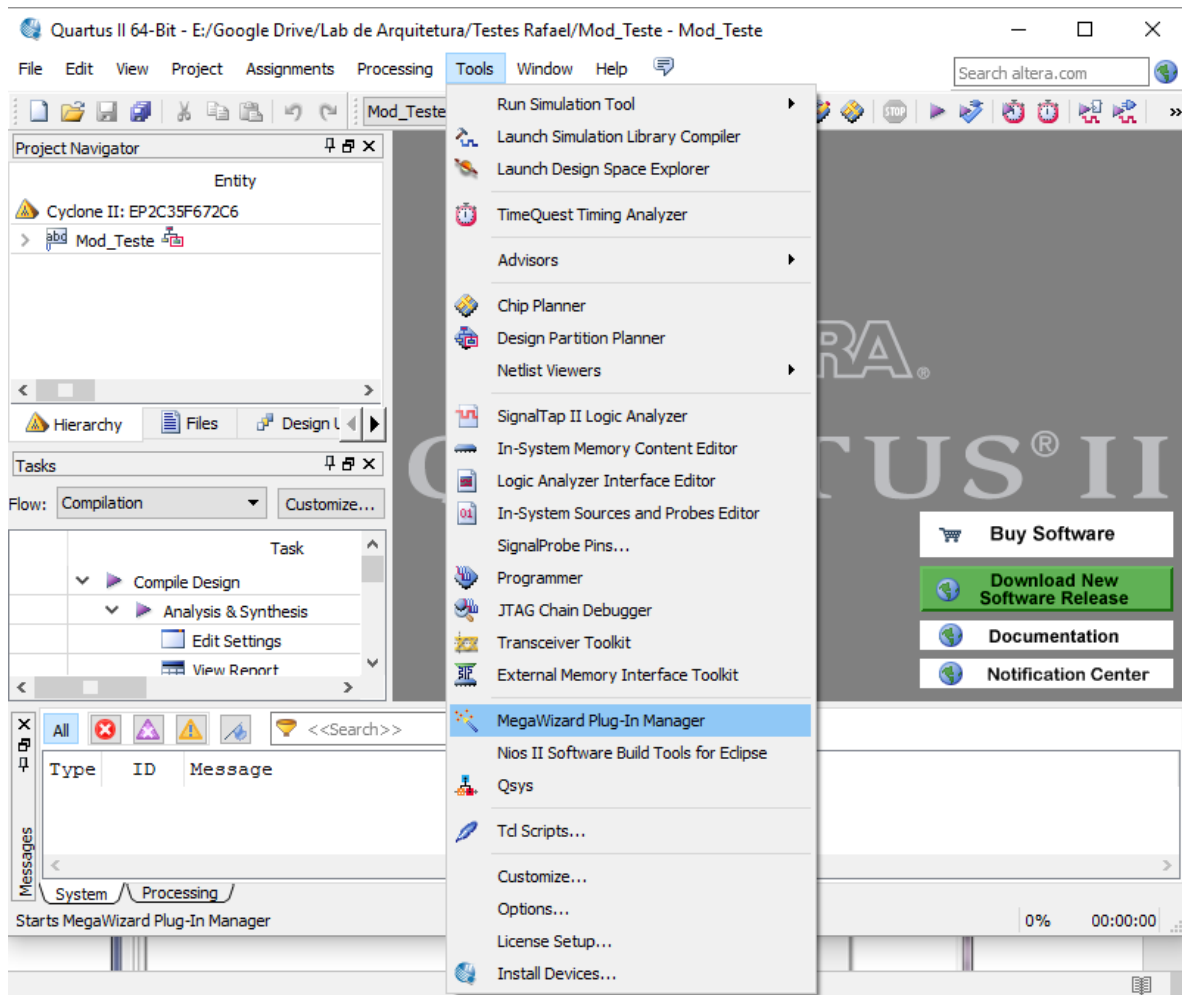
Addr	+0	+1	+2	+3	+4	+5	+6	+7
0	0000000000000000	0000000000000000	0000000000000000	0000000000000000	0000000000000000	0000000000000000	0000000000000000	0000000000000000
8	0000000000000000	0000000000000000	0000000000000000	0000000000000000	0000000000000000	0000000000000000	0000000000000000	0000000000000000
16	0000000000000000	0000000000000000	0000000000000000	0000000000000000	0000000000000000	0000000000000000	0000000000000000	0000000000000000
24	0000000000000000	0000000000000000	0000000000000000	0000000000000000	0000000000000000	0000000000000000	0000000000000000	0000000000000000

Após editar os dados das posições de memória, salve o arquivo com o nome “MemInstrucao.mif” na sua pasta pessoal. Não é necessário acrescentar a extensão.

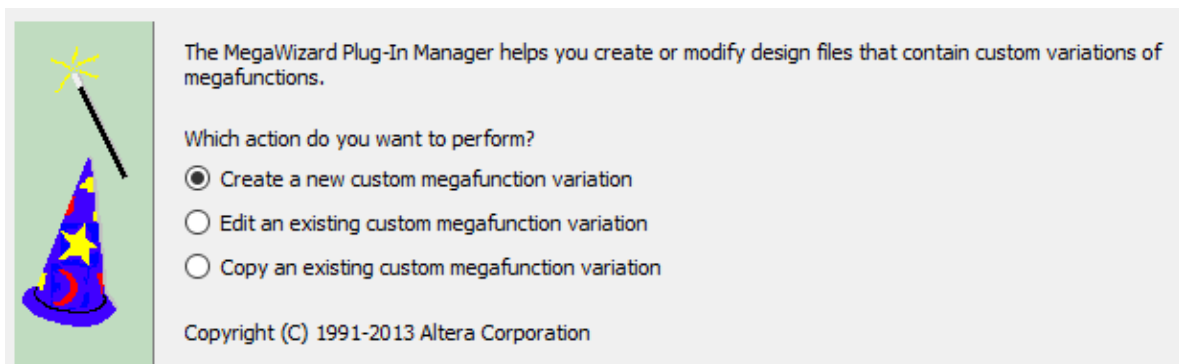
MEMÓRIA ROM

Veremos um passo a passo de como criar uma memória ROM no Quartus II. Observe que nós precisamos de uma memória ROM completamente combinacional para que a CPU funcione corretamente. Infelizmente o software só é capaz de produzir memórias com registros. Estes registros podem estar na entrada de endereços e/ou saída de dados e precisam de um sinal de clock. Utilizamos então um sinal de clock muito mais rápido do que o da CPU. Desta forma, a memória aparece para a CPU como se fosse combinacional, uma vez que uma mudança nos endereços provoca rapidamente uma mudança nos dados.

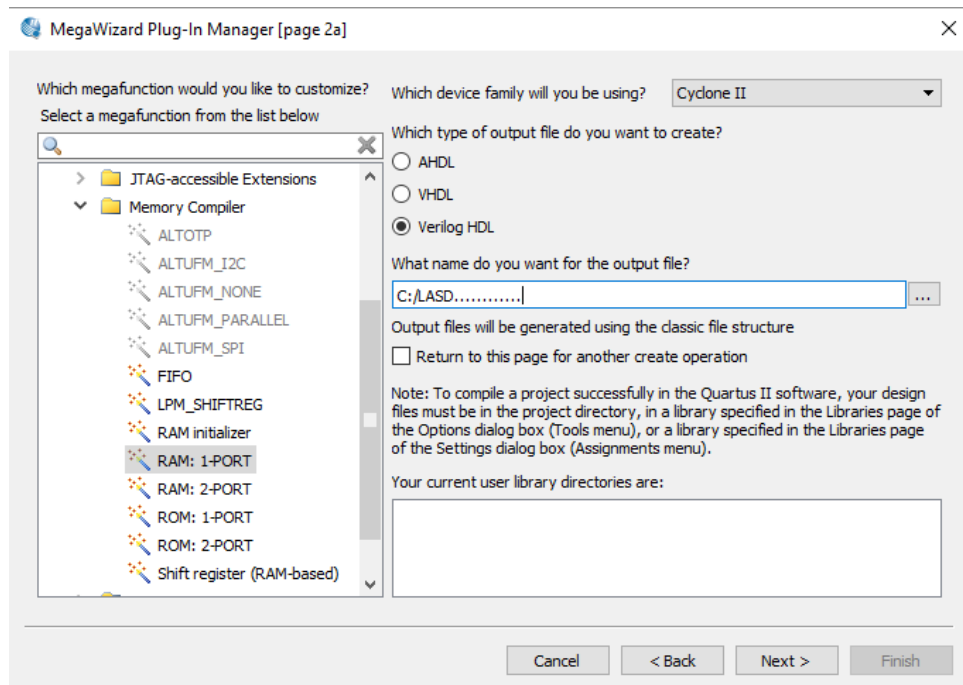
Veremos nas telas a seguir como produzir uma memória ROM no Quartus II. Iniciamos utilizando o menu tools -> megawizar plugin-manager...



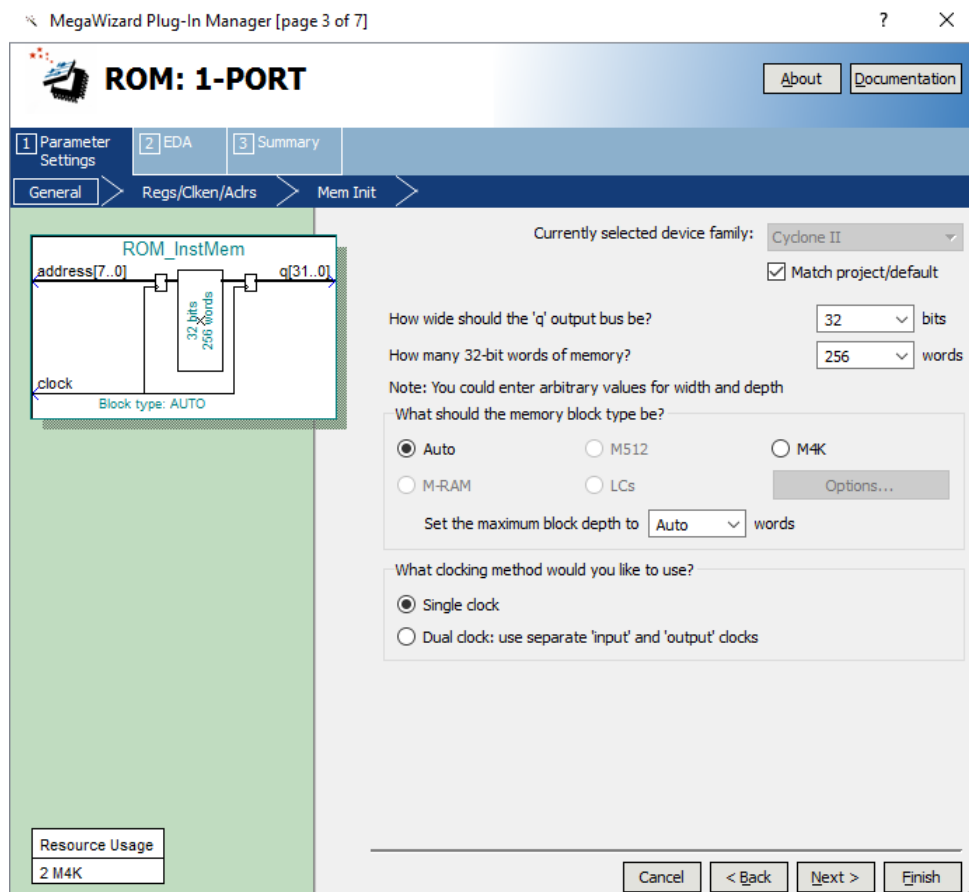
Utilizamos a opção de criar uma nova “mega-função”



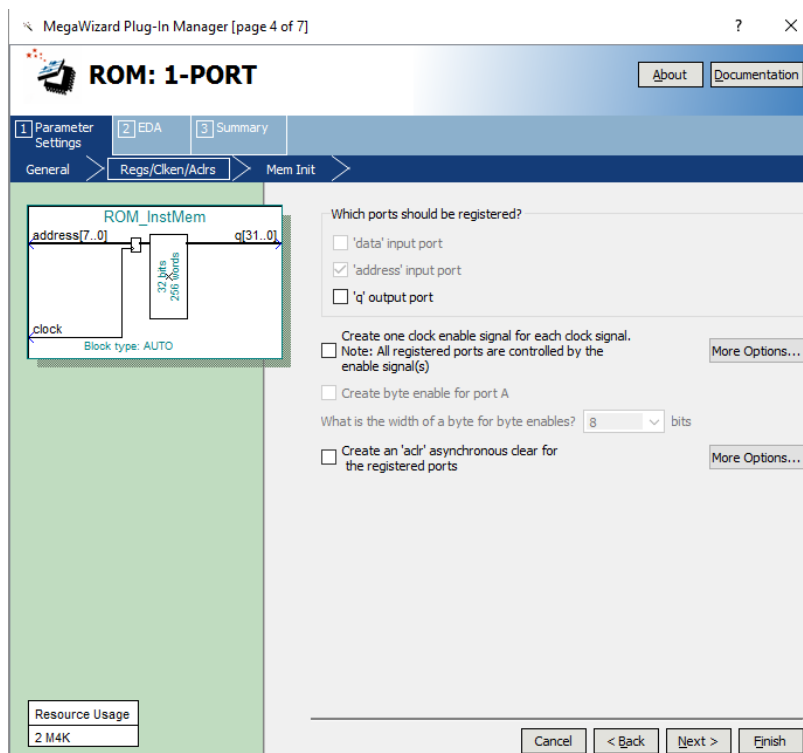
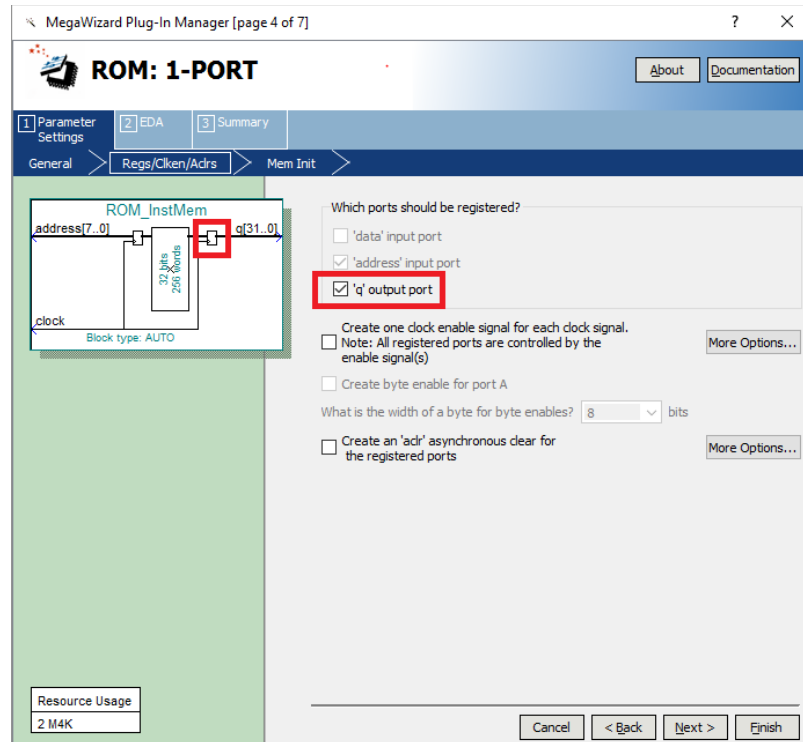
Selecione a opção de memória ROM de uma porta (ROM: 1-PORT):



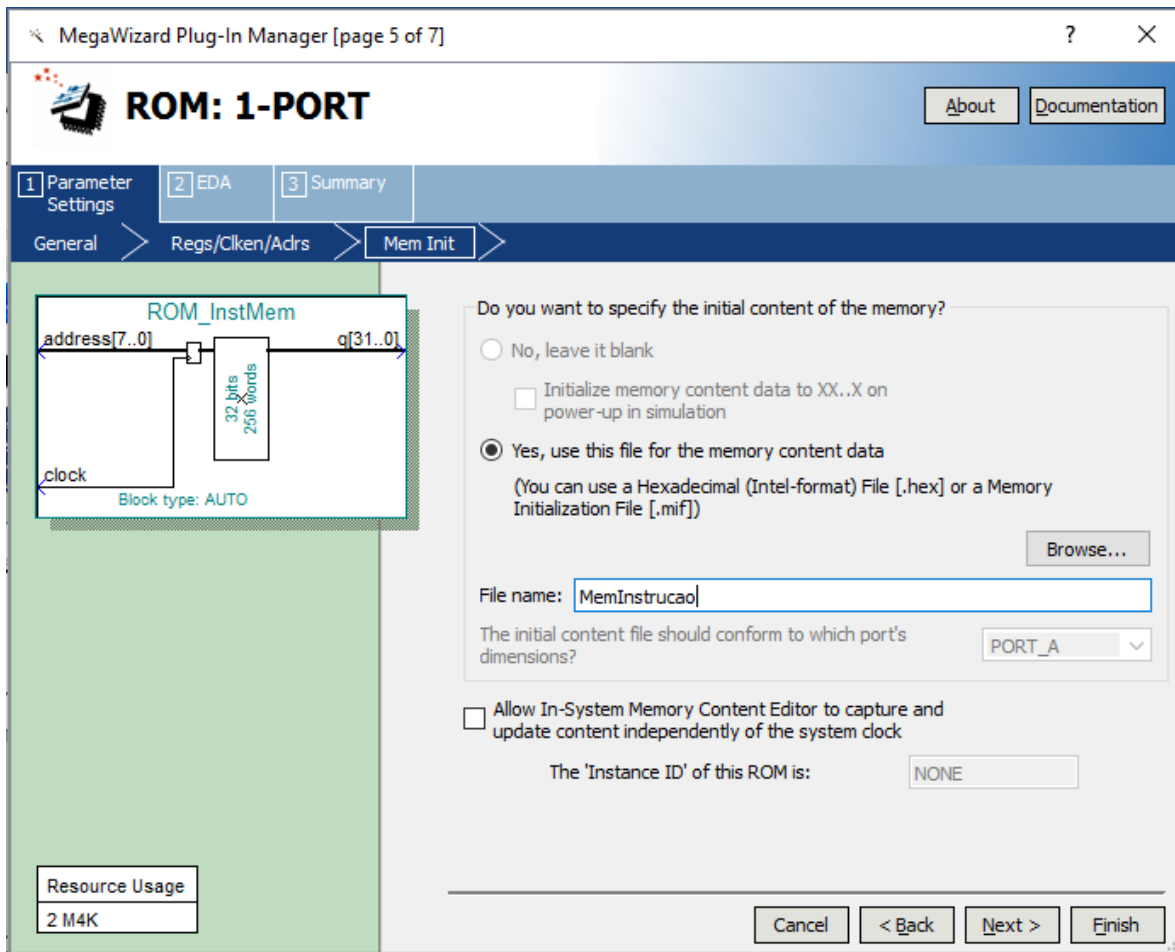
Selecionamos a largura da palavra em bits (32 bits) e a quantidade de palavras da memória (256 palavras). Devemos utilizar a opção de apenas um clock:



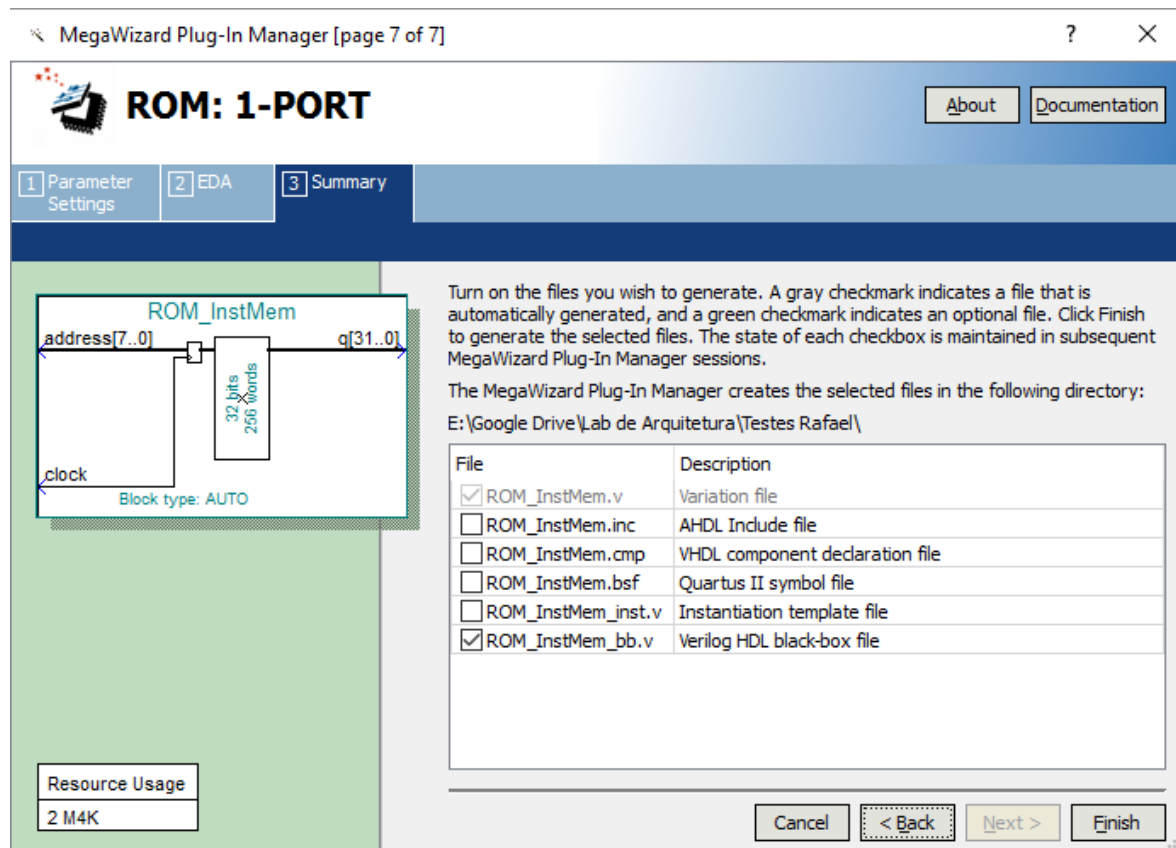
Observe que a memória é criada com um registrador na entrada e na saída (registrador de saída indicado no círculo em vermelho na figura abaixo). O registrador de saída deverá ser excluído, desmarcando a opção “ ‘q’ output port ” indicada no quadro em vermelho. Ao fazer isso, o desenho do módulo da memória fica como indicado na figura seguinte.



Clicamos em next e definimos o arquivo mif (ou hex) que informa o conteúdo da memória. Utilizaremos a opção que permite usar o Editor de Conteúdo da Memória, que permite visualizar e modificar o conteúdo da memória enquanto o FPGA está rodando utilizando uma ferramenta dentro do Quartus. O nome do arquivo.mif é o indicado no quadro vermelho na figura e corresponde ao arquivo MemInstrucao.mif que foi gerado no procedimento descrito no início do Guia. Para garantia de que será usado o arquivo correto, utilize o botão de “BROWSE” para acessar o arquivo gravado na sua pasta de projeto.



Após clicar no botão “Next”, ignore a próxima tela (clique next). Na tela seguinte defina os arquivos que serão gerados conforme indicado na figura abaixo:

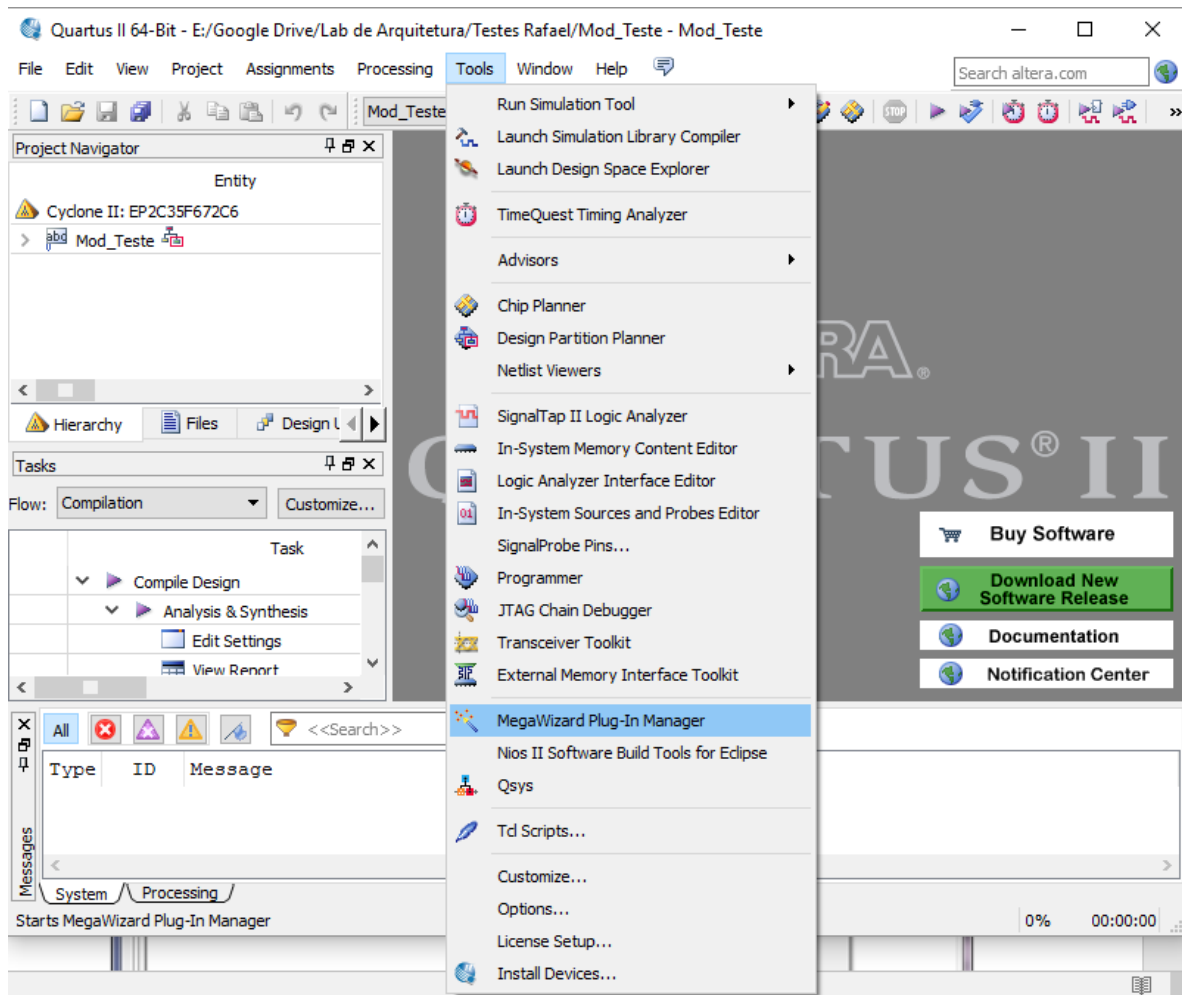


Podemos clicar em finish e utilizar o arquivo verilog gerado “MemInstrucao.v”.

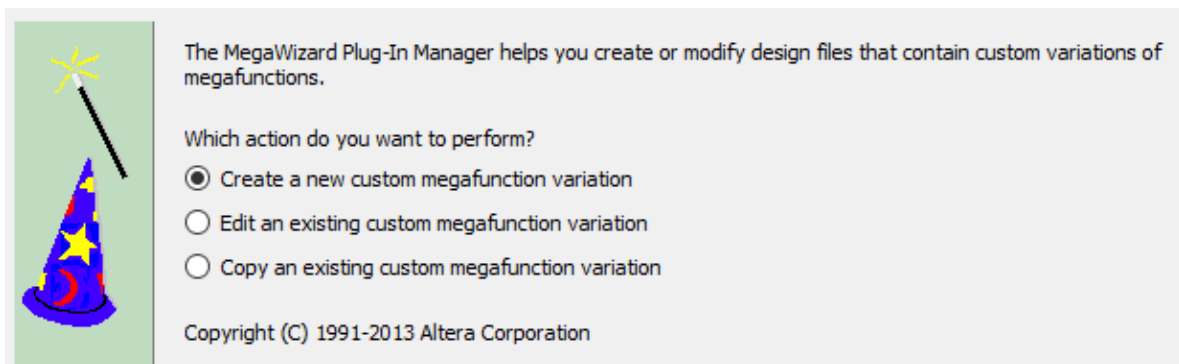
MEMÓRIA RAM

Veremos um passo a passo de como criar uma memória RAM no Quartus II. Observe que nós precisamos de uma memória RAM completamente combinacional para que a CPU funcione corretamente. De modo equivalente a solução adotada para a memória ROM, utilizaremos um sinal de clock muito mais rápido do que da CPU. Desta forma, a memória aparece para a CPU como se fosse combinacional, uma vez que uma mudança nos endereços provoca rapidamente uma mudança nos dados.

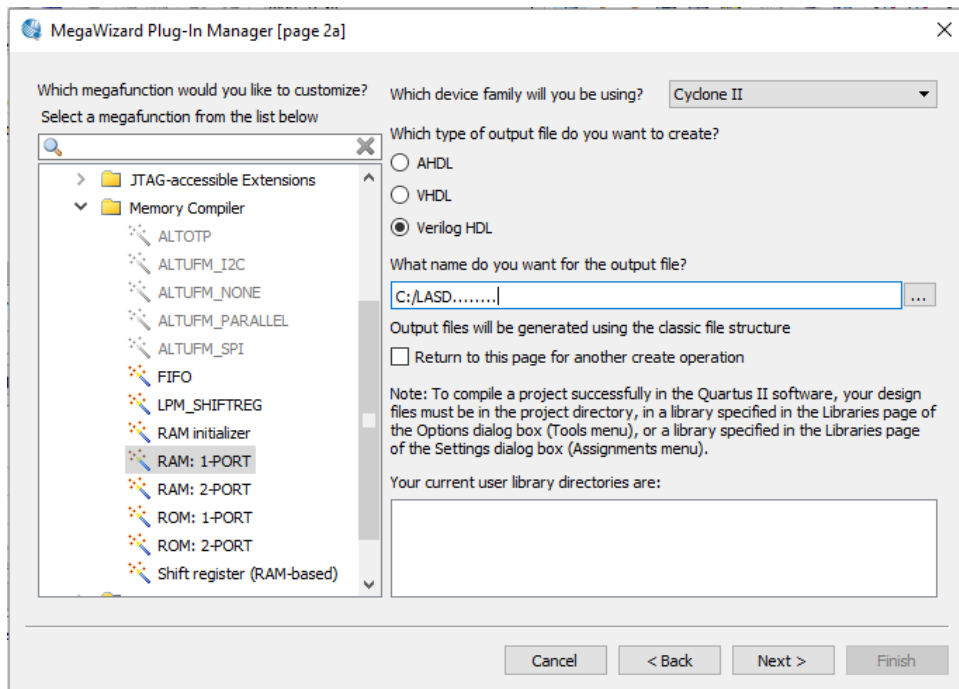
Veremos nas telas a seguir como produzir uma memória RAM no Quartus II. Iniciamos utilizando o menu tools -> megawizar plugin-manager...



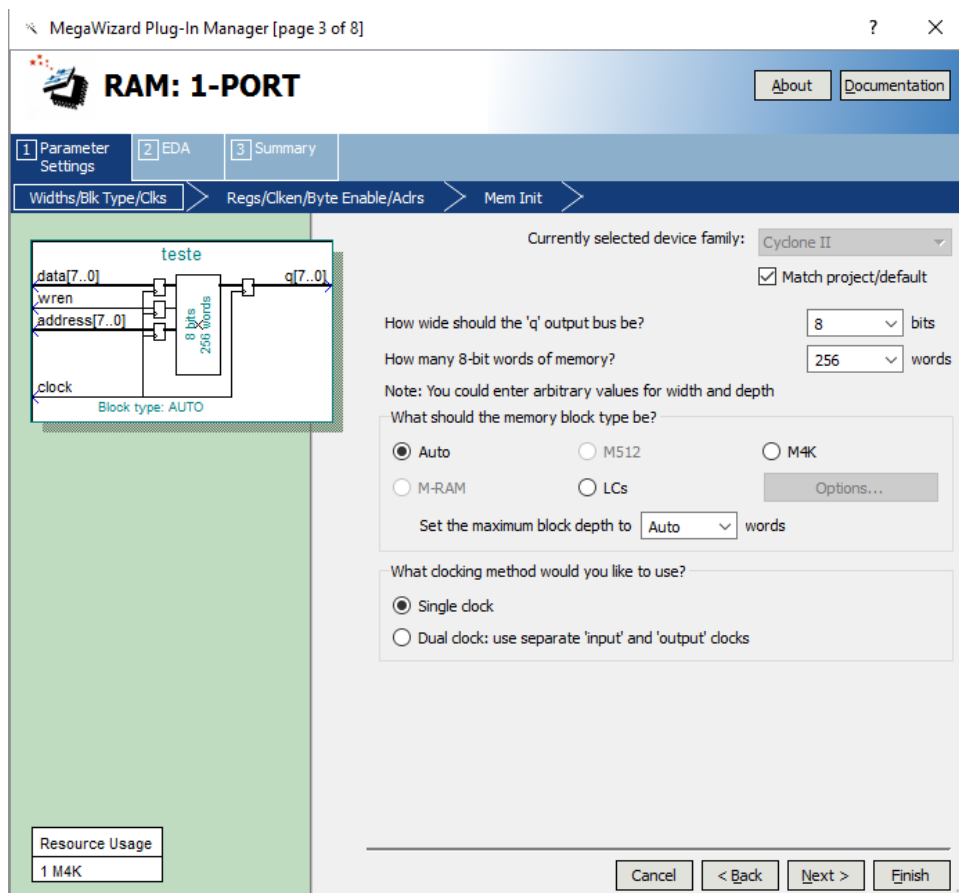
Utilizamos a opção de criar uma nova “mega-função”



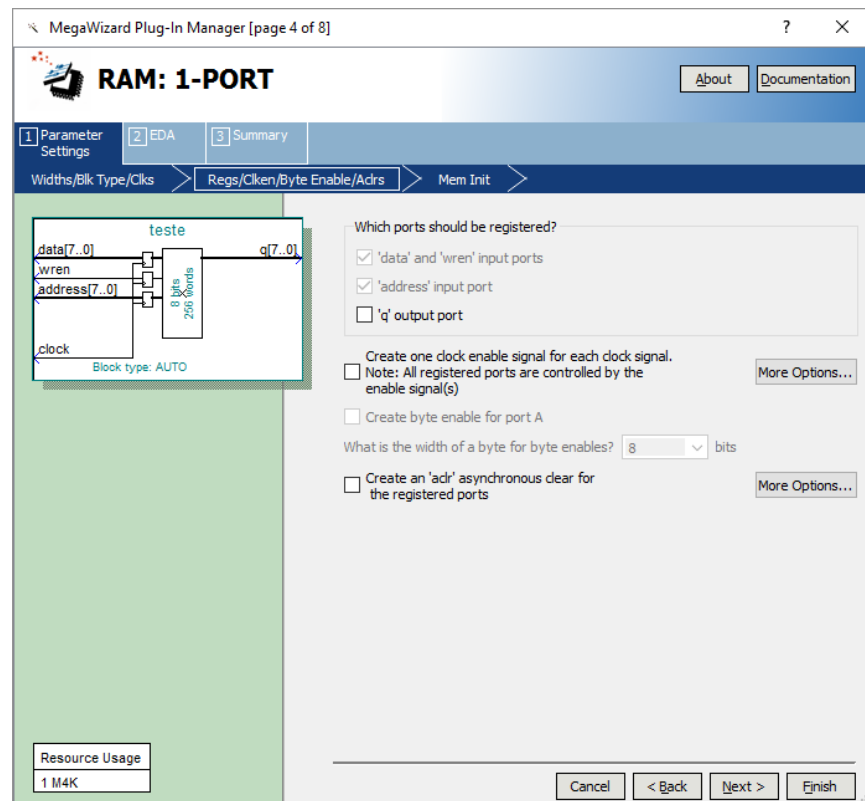
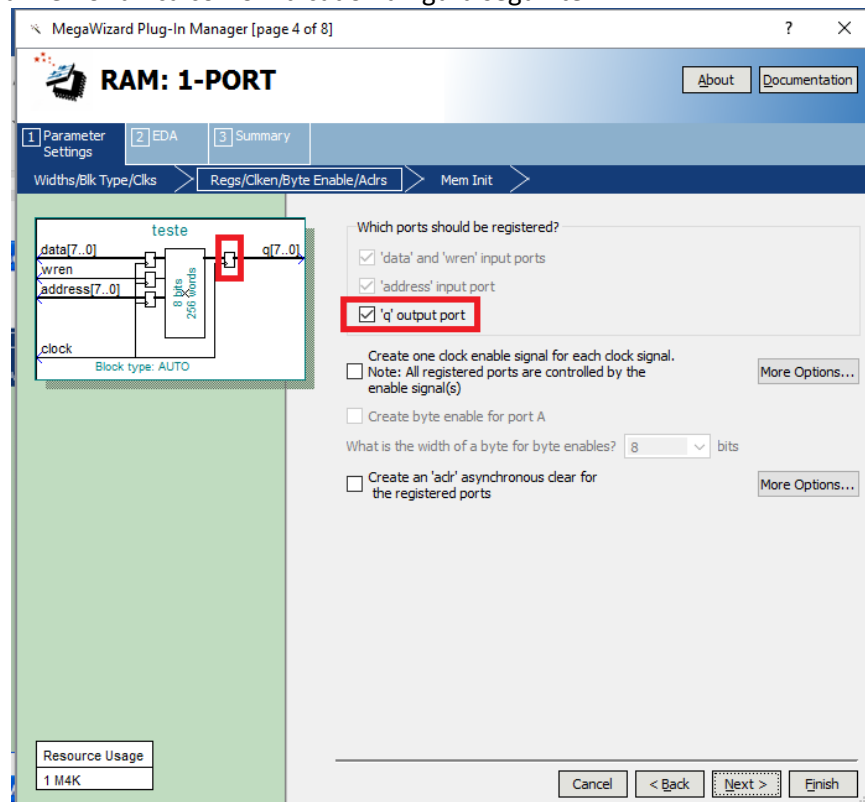
Selecione a opção de memória RAM de uma porta (RAM: 1-PORT), informe o diretório de trabalho e clique next:



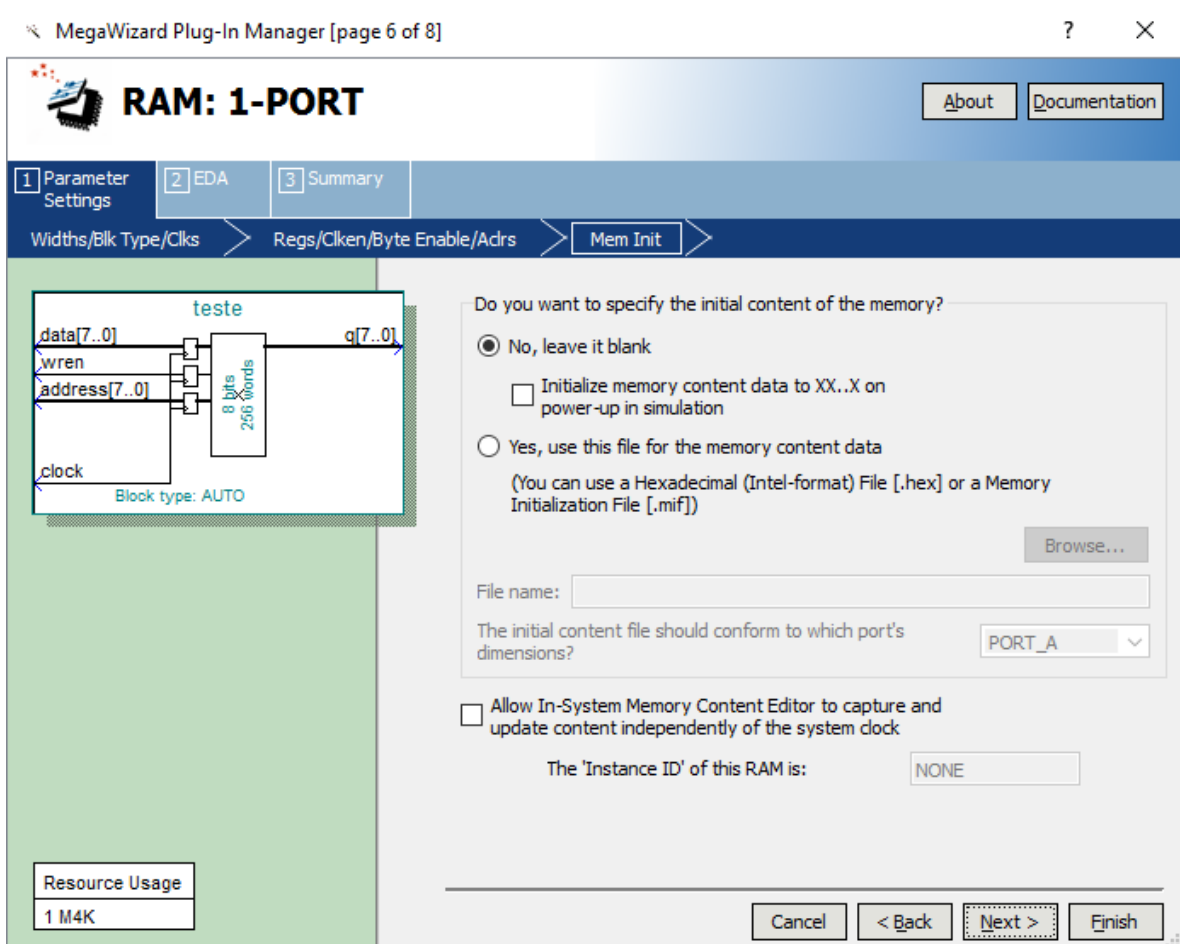
Selecionamos a largura da palavra em bits (8 bits) e a quantidade de palavras da memória (256 palavras). Devemos utilizar a opção de apenas um clock:



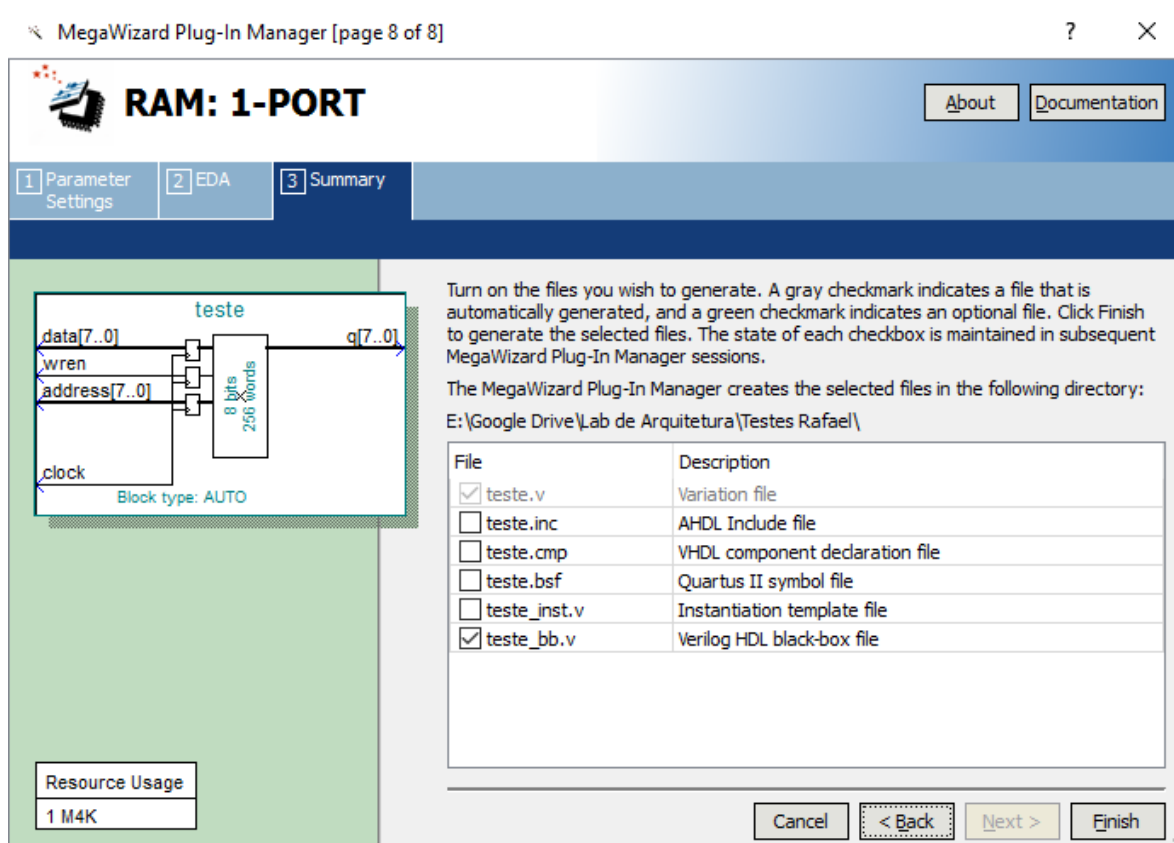
Observe que a memória é criada com um registrador na entrada e na saída (registrador de saída indicado no círculo em vermelho na figura abaixo). O registrador de saída deverá ser excluído, desmarcando a opção “ ‘q’ output port ” indicada no quadro em vermelho. Ao fazer isso, o desenho do módulo da memória fica como indicado na figura seguinte



Clicamos em next e mantemos a opção que faz com que a memória seja inicializada com todos os valores iguais a zero. Caso desejemos, podemos definir um arquivo mif (ou hex) para inicialização da memória RAM, de forma equivalente ao que foi feito para a memória ROM.



Após clicar no botão “Next”, ignore a próxima tela (clique next). Na tela seguinte defina os arquivos que serão gerados conforme indicado na figura abaixo:



Podemos clicar em finish e utilizar o arquivo verilog gerado “MemDados.v”.

IMPLEMENTAÇÃO EM VERILOG

A placa DE2 usada nos laboratórios possui dois osciladores a cristal de 50 MHz e 27 MHz, ligados aos pinos denominados CLK50 e CLK27, respectivamente. Estes sinais já estão conectados ao FPGA em entradas especiais para sinais de clock.

Neste experimento devemos implementar um contador para ser utilizado como divisor de frequência. Utilize o clock de 50MHz e implemente um divisor que gere na sua saída um clock de 1Hz. Esse sinal de 1Hz será conectado a entrada “CLK” dos módulos “RegFile” e “Unidade de Controle”. Essa conexão se dará quando for pedido que se faça o teste da implementação no modo Automático.

Ao realizar as conexões desse experimento, deve-se substituir a “memória de instrução” do experimento anterior, pela memória ROM criada através do procedimento descrito anteriormente. A memória RAM, por sua vez, deverá ser acrescida ao circuito conforme a Figura 1.

Observar que o CLOCK_50 deverá ser ligado a entrada “clock”, tanto da memória ROM como da memória RAM.

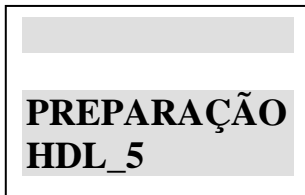
TESTE MANUAL E AUTOMÁTICO

O primeiro teste da execução da CPU pode ser feito utilizando um botão para a entrada de clock. Neste caso, devemos fornecer mecanismos de apresentação dos valores do PC e dos registradores. Quando a CPU estiver funcionando a contento, devemos ligar a entrada de clock na saída do divisor de frequência.

Para que a CPU não execute instruções não programadas, devemos fazer com que ela pare de executar as instruções seguintes. Isto é realizado através de uma instrução que faça o PC voltar para a sua mesma posição atual.

Questões de Preparação

1. Descreva como proceder para fazer um LED da placa DE2 piscar a cada 1s.
2. O clock da memória ROM deve ser o mesmo utilizado na entrada do divisor de frequência para a CPU (50MHz). Descreva a temporização de um ciclo de leitura na memória, supondo que o atraso combinacional da memória, após a entrada ser registrada seja de 80ns. Qual o atraso combinacional máximo para esta memória se o clock do processador for de 1MHz.
3. Faça um programa que leia os valores armazenados na memória de dados nos endereços de 0 a 9 e escreva de volta cada valor incrementado.
4. Como você testaria no laboratório a execução do programa da questão anterior?



Operações de Entrada e Saída

OBJETIVOS

- Integrar a CPU em um componente que acessa outros dispositivos através de suas interfaces
- Conhecer os conceitos de entrada/saída (E/S) mapeada em memória, isolada e SFR.
- Implementar uma porta paralela de entrada e saída em HDL
- Ligar uma porta paralela a CPU e utilizar esta porta para comunicação com LEDs e chaves da placa DE2
- Utilizar um módulo pronto de comunicação serial assíncrona e fazer a comunicação entre a placa DE2 e o microcomputador
- Entender a temporização dos sinais dos barramentos de endereço, de dados e de controle da CPU, memória e E/S
- Entender a conversão de níveis elétricos e lógicos através de circuitos de buffers e drivers de corrente e tensão
- Entender a temporização de um programa em Assembly e realizar rotinas temporizadas

INTRODUÇÃO

Nos experimentos anteriores realizamos um computador formado por uma unidade central de processamento (CPU) e memórias de instrução e de dados. Com este computador realizamos alguns testes básicos para garantir o seu funcionamento. Para tanto, colocamos diversas interfaces de saída no computador que nos permitia visualizarmos seus barramentos internos, valores de registradores e do contador de programas, etc.

Devemos observar, no entanto, dois fatos: primeiro, que em um computador normal não trabalhamos observando LEDs e displays “piscando”. Ou seja, as interfaces de depuração não fazem parte do computador, e uma vez que já cumpriram o seu papel podem ser retiradas. O segundo fato é que, sem estas interfaces o nosso computador fica praticamente inútil. Ele continua sendo capaz de realizar diversos tratamentos com valores armazenados nos registradores e na memória, mas não podemos interagir com este processamento. Para que um computador seja útil, ele precisa se comunicar com outros dispositivos, recebendo e enviando valores digitais. Isto permite que o computador opere com o mundo físico, real. Esta comunicação com os dispositivos dá-se através das interfaces de entrada e saída. Já conhecemos muitos dispositivos externos ao computador, tais como: mouse, teclado, monitor, câmeras, etc. Todos estes dispositivos se comunicam com o computador através de interfaces de E/S. Também temos muitos dispositivos de E/S que normalmente são internos ao computador como conhecemos, tais como dispositivos de som, disco rígido, etc. Do nosso ponto de vista, não há distinção entre estes dispositivos internos e externos, pois todos são ligados através de interfaces de E/S com o restante do computador, a saber, CPU e memórias.

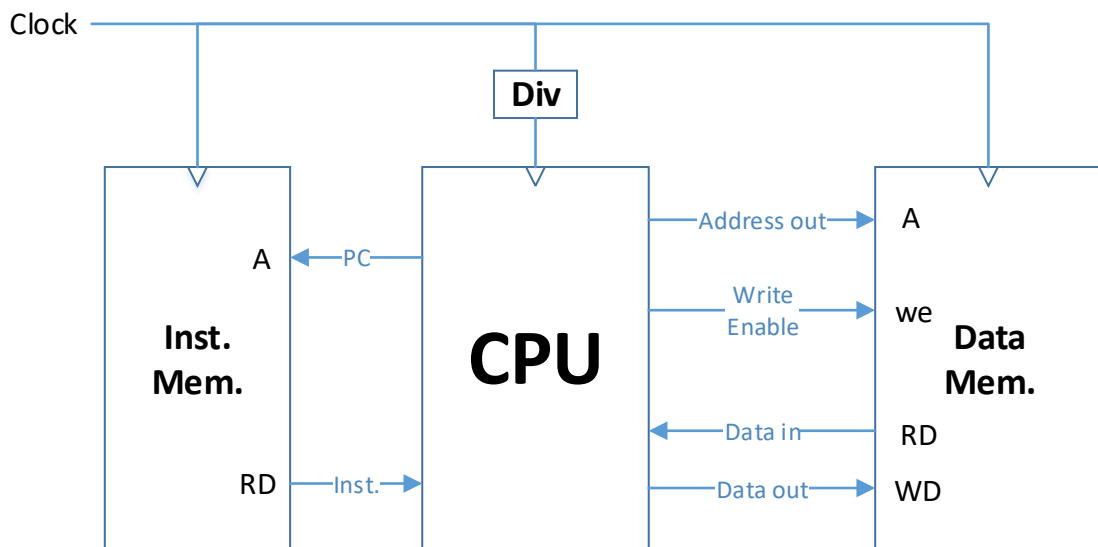


Figura 1. Computador implementado no laboratório

Interface com o computador

Mas como estas interfaces são conectadas com o computador se tudo o que ele possui são a própria CPU conectada com as memórias, como mostrado na Figura 1?

A idéia básica de um sistema de entrada e saída é utilizar exatamente esta única interface existente entre o processador e a memória de dados. A interface é conectada aos barramentos de endereço *Address out* e dados. Ligamos ao barramento de leitura de dados, *Data in*, se a interface for de entrada, *Data out*, se a interface for de saída.

Este tipo de E/S em que utilizamos o sistema de memória do processador chamamos de E/S mapeada em memória. A expressão “mapeada em memória” significa que o circuito de entrada e saída aparece como posições de memória para o processador. Há outros mecanismos de entrada e saída que serão vistos durante o curso.

Interface de saída paralela mapeada em memória

Como exemplo, veremos como implementar uma interface simples de saída paralela, ou seja, com todos os bits apresentados *simultaneamente*. Esta interface será mapeada em memória na nossa CPU e utilizaremos o endereço FFh, neste exemplo. Ou seja, ao endereçarmos para escrita esta posição de memória específica, através da instrução *store* da nossa CPU, estaremos ativando a nossa interface paralela de saída e o valor do dado que seria armazenado, vai ser entregue a interface. Devemos lembrar, que a nossa memória de dados não deve realizar nenhuma operação de escrita quando esta instrução for executada.

Um hardware que realiza estas operações como descritas acima pode ser visto na Figura 2. Mostramos apenas os sinais que nos interessam.

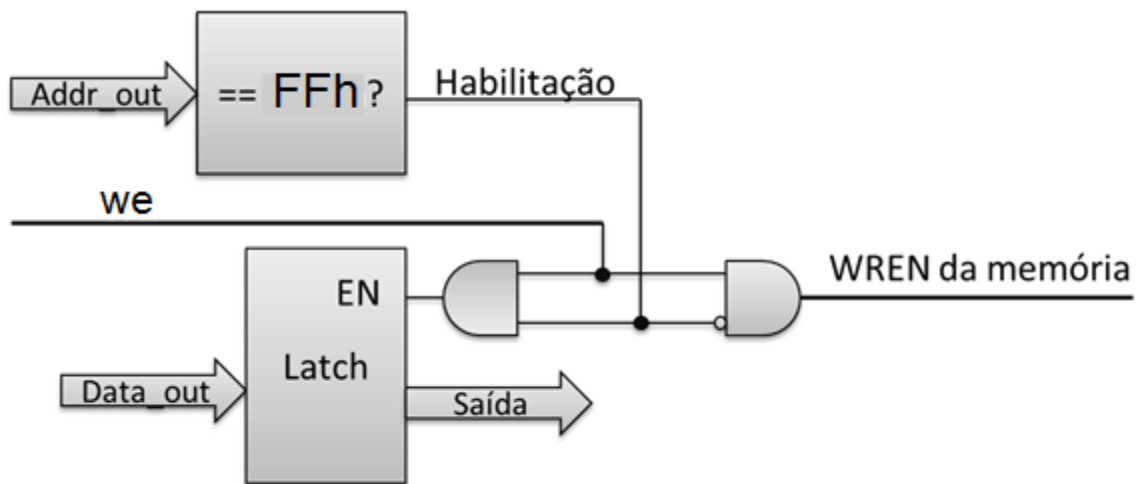


Figura 2. Interface de saída paralela mapeada em memória

Podemos observar na Figura 2 que o sinal de habilitação só é ativado quando o endereço que estiver no barramento *Address out* for FFh. Este sinal, juntamente com o sinal de escrita na memória *we*, é utilizado para definir o sinal de habilitação de um latch. Este latch (de 8 bits) tem como entrada de dados o barramento *Data out* e sua saída de dados é a própria saída do computador. Esta saída pode ser ligada a algum dispositivo, tal como um conjunto de LEDs. Para evitar que a memória de dados realize alguma escrita quando estivermos operando neste endereço específico, o sinal WREN só é habilitado se não estivermos no endereço FFh.

Estes sinais, *Address out*, *Data out* e *we* são gerados quando ha execução de uma instrução STORE. Suponha que \$2 possua o valor 55h. A execução da instrução:

SW \$2 FF(\$0)

fará *Address out* = FFh, *Data out* = 55h e *we* = 1, durante um ciclo de clock da CPU. Neste tempo, o latch ficará habilitado, ou seja, ficará transparente, deixando o valor de *Data out* passar para a saída. No final do ciclo, o sinal de habilitação do latch será desativado, fazendo-o armazenar o valor que estava apresentando. No lugar de um latch podemos utilizar um registro (conjunto de flip-flops) conectado ao clock da CPU. Neste caso, o valor de saída só será atualizado no final do ciclo de clock.

Observe que apenas um dos elementos ligados ao barramento deve ser ativado por vez. Este comportamento de apenas um sinal ativado por vez é típico de decodificadores binários. E por isto usualmente denominamos o circuito que ativa os dispositivos de acordo com os endereços de *decodificador de endereços*, mesmo que ele não possua nenhum decodificador binário.

Interface de entrada paralela mapeada em memória

A leitura em uma porta de entrada mapeada em memória é feita de forma similar, como é mostrada na Figura 5.

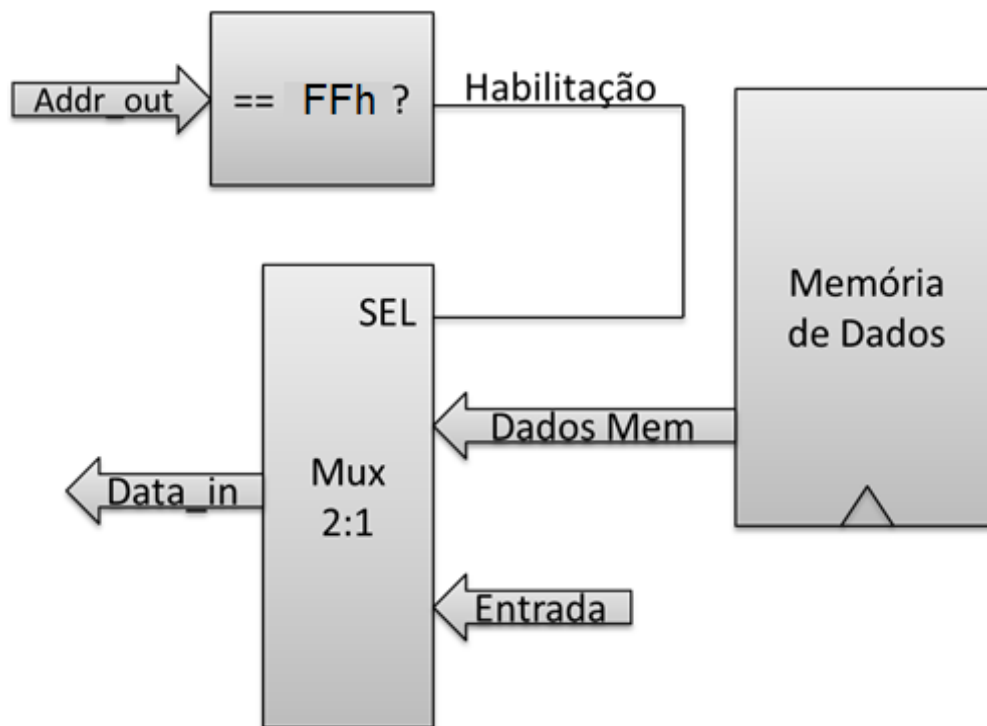


Figura 3. Leitura de porta feita de entrada mapeada em memória no endereço FFh.

Comunicação Serial e Paralela

A comunicação entre equipamentos digitais podem ser feitas através de interfaces paralelas ou seriais. Uma interface paralela como a que vimos nos exemplos anteriores, lê ou escreve os diversos bits de dados simultaneamente. Uma interface serial, diferentemente, opera com apenas 1 bit por vez. Para enviar ou receber uma palavra de dados, os bits individuais desta palavra são enviados por esta interface de um bit em instantes de tempo diferentes. O processo de separar os bits no tempo é chamado de serialização e o processo de unir os bits e apresentá-los todos juntos, em paralelo, é chamado “de-serialização”. Veremos mais adiante no curso mais detalhes sobre os diversos mecanismos de comunicação paralela e serial, síncrona e assíncrona, como também os mecanismos de *handshake* e protocolos.

No entanto, podemos resumir rapidamente a comunicação serial de forma a realizarmos um experimento de comunicação entre o computador construído no FPGA e o PC. O material a seguir é baseado no texto encontrado no site: <http://www.fpga4fun.com>, que possui diversos projetos e tutoriais sobre FPGA e HDL. Utilizaremos a interface serial assíncrona desenvolvida neste site, com pequenas modificações. O código Verilog da interface será deixado na página da disciplina.

Comunicação Serial Assíncrona

A comunicação serial que utilizaremos é assíncrona, ou seja, o clock utilizado no receptor não possui nenhuma relação com o clock que foi utilizado no transmissor.

Transmissor Assíncrono

Cria o sinal "TxD" ao serializar o dado que vai ser transmitido

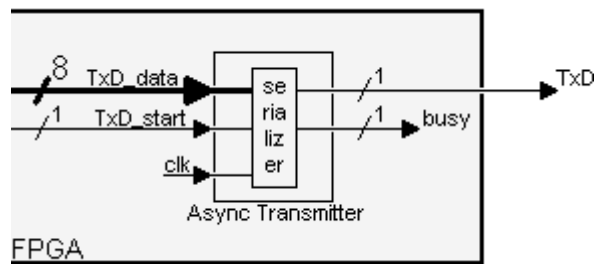


Figura 4. Transmissor serial assíncrono com 8 bits de dado e sinal de ocupado

Receptor Assíncrono

Captura o sinal "RxD" de fora do FPGA e "de-serializa" para que possa ser usado facilmente dentro do FPGA.

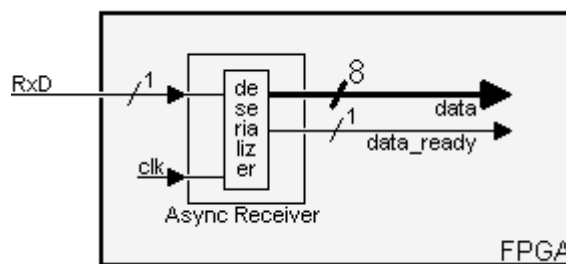


Figura 5. Receptor serial assíncrono com 8 bits de dados e sinal de dado recebido

Interface serial RS-232

Existem diversos padrões de comunicação serial entre equipamentos. Um padrão simples para comunicação ponto-a-ponto é o RS-232 (EIA-232). Este padrão é sucinto e especifica apenas algumas características. De forma específica (fonte: wikipedia):

A Eletronics Industries Association (EIA), que padronizou o RS-232-C em 1969 define:

- Características elétricas como níveis de tensão, taxa de sinalização, taxa de rotação dos sinais, nível máximo de tensão, comportamento de curto-circuito e carga máxima da capacitância;
- Características mecânicas da interface, conectores "plugáveis" e identificação dos pinos;
- Funções de cada circuito no conector da interface;
- Subconjuntos padrões de circuitos de interface para aplicações selecionadas de telecomunicação.

O padrão não define elementos como:

- Codificação de caracteres (por exemplo, ASCII, código Baudot ou EBCDIC);

- Enquadramento dos caracteres no fluxo de dados (bits por caractere, bits de início e parada, paridade);
- Protocolos para detecção de erros ou algoritmos para compressão de dados;
- Taxas de bit para transmissão, apesar de o padrão dizer ser destinado para taxas de bits menores que 20.000 bits por segundo. Muitos dispositivos modernos suportam velocidade de 115.200 bit/s;
- Fornecimento de energia para dispositivos externos.

Uma interface serial RS-232 possui as seguintes características:

- Usa um conector de 9 pinos, denominado “DB-9” (PCs antigos usavam 25 pinos - “DB-25”)
- Permite comunicação bidirecional *full-duplex* (o PC pode enviar e receber dados ao mesmo tempo).

Conector DB-9

Conector encontrado nos PCs. Está entrando em desuso, sendo substituído pela USB.



Figura 6. Conector DB-9

Este conector possui, segundo o padrão, diversos sinais. Os 3 sinais mais importantes são:

- Pino 2: RxD (Recepção de dados)
- Pino 3: TxD (Transmissão de dados)
- Pino 5: GND (Terra, referência de tensão)

Usando apenas 3 fios podemos enviar e receber dados (simultaneamente).

Observe que a posição e a direção dos pinos é relativa ao PC. Devemos conectar o pino RxD do PC no pino TxD do dispositivo e o pino TxD do PC no RxD do dispositivo. Os cabos que utilizaremos no laboratório já fazem estas conexões.

Comunicação Serial

Os dados são enviados um bit por vez. Um fio é usado para cada direção. Como computadores normalmente utilizam diversos bits de dados, os dados são serializados antes de serem enviados. Os dados são normalmente enviados em blocos de 8 bits (um byte). O LSB (bit menos significativo, bit 0 dos dados) é enviado primeiro, o MSB (bit mais significativo, bit 7) é enviado por último.

Comunicação assíncrona

A interface é assíncrona, significando basicamente que o clock não é transmitido juntamente com os dados. O receptor precisa encontrar um modo de se temporizar de forma alinhada com os bits que estão chegando.

No caso da RS-232, isto é feito da seguinte forma:

1. Os dois lados do cabo concordam previamente com os parâmetros de comunicação (velocidade, formato, etc). Isto é feito manualmente antes da comunicação começar;
2. O transmissor envia “1” enquanto e por tanto tempo quanto a linha esteja desocupada;
3. O transmissor envia um “start” (início), que é um “0”, antes de cada byte transmitido, de forma que o receptor possa identificar que um dado está chegando;
4. Após o “start”, os dados chegam na velocidade e formato acordados, de forma que o receptor possa interpretá-los.
5. O transmissor envia um “stop” (parada), que é um bit “1”, após cada byte.

Vejamos como se parece o byte 0x55 quando é transmitido:

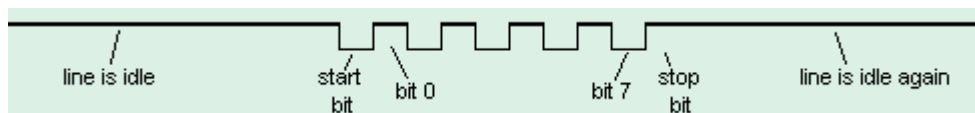


Figura 7. Enviando 0x55

Em binário, 0x55 é 01010101. Mas como o LSB é transmitido primeiro, a linha chaveia assim “1-0-1-0-1-0-1-0”.

Eis outro exemplo:

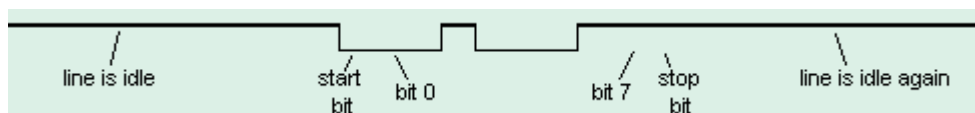


Figura 8. Enviando 0xC4

Aqui o dado é 0xC4, você consegue ver?

As transições de bits são difíceis de ver. Isto ilustra o quanto é importante para o receptor saber em que velocidade o dado é enviado.

Quão rápido podemos enviar os dados?

A velocidade é especificada em baud, ou seja, representa o número de mudanças na linha de transmissão. Isto corresponde no RS-232 basicamente em quantos bits por segundo podem ser enviados. Por exemplo, 1000 bauds seria o equivalente a 1000 bits por segundo, ou que cada bit dura um milissegundo.

As implementações comuns da interface RS-232 (como a usada nos PCs) não permite que qualquer velocidade seja utilizada. Deve utilizar alguma velocidade “padrão”. Os valores comuns são:

- 200 bauds.
- 9600 bauds.
- 38400 bauds.
- 115200 bauds (normalmente o mais rápido que se pode ir).

Em 115200 bauds, cada bit dura $(1/115200) = 8,7\mu\text{s}$. Se você transmitir dados de 8 bits, isto dura $8 \times 8,7\mu\text{s} = 69\mu\text{s}$. Mas cada byte requer um bit extra de start e outro de stop, de forma que precisa-se realmente de $10 \times 8,7\mu\text{s} = 87\mu\text{s}$. Isto corresponde a uma velocidade máxima de 11,5KBytes por segundo.

A 115200 bauds, alguns PCs mais antigos precisam um bit de parada mais longo (1,5 ou 2 bits de comprimento...), o que faz a velocidade máxima cair para algo em torno de 10,5KBytes por segundo.

Camada Física

Os sinais nos fios usam um esquema de tensões positivas e negativas:

- “1” é enviado usando -12V (ou algo entre -3V e -15V);
- “0” é enviado usando +12V (ou algo entre 3V e 15V).

Assim uma linha desocupada mantém sua tensão em algo como -12V.

Gerador de Baud (taxa de comunicação)

Neste projeto vamos utilizar a serial na sua velocidade máxima, ou seja, 115200 bauds. Outras velocidades também são fáceis de serem geradas. FPGAs normalmente “rodam” em velocidades bem acima de 115200 Hz (RS-232 é bastante lenta para os padrões atuais). Isto significa que usaremos um clock de alta velocidade e o dividiremos para gerar um “tick” o mais próximo possível de 115200 por segundo.

Gerando tick síncrono a partir de um clock de 1,8432MHz

Tradicionalmente, chips RS-232 usam um clock de 1,8432MHz porque isto torna a geração de frequências de bauds padrões muito fácil. Se tivéssemos um clock de 1,8432MHz disponível, poderíamos utilizar a seguinte descrição Verilog, sabendo que 1,8432MHz dividido por 16 resulta em 115200.

```
reg [3:0] BaudDivCnt;
always @(posedge clk) BaudDivCnt <= BaudDivCnt + 1;

wire BaudTick = (BaudDivCnt==15);
```

Aqui usamos o fato de que podemos definir um fio e atribuir um valor continuamente, correspondendo a utilizar a construção *assign*.

Assim, “BaudTick” é ativado uma vez a cada 16 clocks, isto é, 115200 vezes por segundo ao usarmos um clock de 1,8432MHz.

Tick síncrono a partir de qualquer frequência

Como proceder se o clock que se tem acesso não é múltiplo da taxa de baud (baudrate)? Por exemplo, vamos supor que temos acesso a um clock de 2MHz. Para gerar 115200 a partir de 2MHz teríamos que dividir por 17.36111..., que não é um número redondo. A solução é dividir algumas vezes por 17, outras vezes por 18, assegurando-se que a razão permanece em 17,36111. Isto é relativamente fácil de fazer:

Observe o seguinte código "C"

```
while(1) // repita para sempre
{
    acc += 115200;
    if(acc>=2000000) printf("*"); else printf(" ");

    acc %= 2000000; // resto da divisão
}
```

Isto imprime o "*" na razão de um a cada "17.36111111..." laços na média. Para obtermos o mesmo de forma eficiente no FPGA, nos baseamos no fato de que uma interface serial pode tolerar um erro de alguns % no gerador de frequência de baud. Realmente não importa se usamos "17,3" ou "17,4".

Gerador de baud em FPGA

Seria desejável que 200000 fosse uma potência de 2 para que usássemos bits do contador. No lugar da razão 2000000/115200 usamos a razão 1024/59 = 17,356. O que é muito perto da nossa razão ideal e faz a implementação em FPGA muito eficiente

```
// 10 bits para o acumulador ([9:0]) e um bit extra para o carry-out do acumulador ([10])
reg [10:0] acc; // 11 bits no total

always @(posedge clk)
    acc <= acc[9:0] + 59; // usamos apenas 10 bits do resultado anterior, mas guardamos os 11 bits

wire BaudTick = acc[10]; // de forma que o 11-esimo bit é o carry-out
```

Usando um clock de 2MHz, "BaudTick" é ativado 115234 vezes por segundo, um erro de 0.03% do ideal de 115200.

Gerador de Baud Parametrizável

O projeto anterior usava 10 bits para acumulador, mas ao passo que o clock de entrada aumenta, são necessários mais bits.

Eis um projeto com clock de 25MHz e um acumulador de 16 bits. O projeto é parametrizado de forma que pode ser facilmente modificado.

```
parameter ClkFrequency = 25000000; // 25MHz
parameter Baud = 115200;
parameter BaudGeneratorAccWidth = 16;
parameter BaudGeneratorInc = (Baud<<BaudGeneratorAccWidth)/ClkFrequency;
```

```

reg [BaudGeneratorAccWidth:0] BaudGeneratorAcc;
always @(posedge clk)
    BaudGeneratorAcc <= BaudGeneratorAcc[BaudGeneratorAccWidth-1:0] + BaudGeneratorInc;

wire BaudTick = BaudGeneratorAcc[BaudGeneratorAccWidth];

```

Esse código tem um pequeno problema de precisão devido ao fato de Verilog utilizar valores intermediários de 32 bits, e o cálculo excede isto. Podemos mudar a linha para a seguinte:

```

parameter BaudGeneratorInc = ((Baud<<(BaudGeneratorAccWidth-4))+(ClkFrequency>>5))/(ClkFrequency>>4);

```

Esta linha também tem a vantagem de obtermos o resultado arredondado e não truncado.

Módulo transmissor RS-232

Vamos construir o módulo transmissor da Figura 3. Ele funciona da seguinte forma:

- O transmissor obtém 8 bits de dados, e os serializa (começando assim que TxD_start for ativado)
- O sinal de ocupado (busy) é ativado enquanto uma transmissão ocorre. O bit de TxD_start é ignorado durante este tempo.

Os parâmetros RS-232 utilizados são fixos: 8 bits de dados, 2 bits de stop, sem paridade.

Serializando os dados

Vamos assumir que possuímos um sinal "BaudTick", ativado 115200 vezes por segundo.

Precisamos gerar o bit de start, os 8 bits de dados e os bits de stop. Uma máquina de estados parece ser apropriada.

```

reg [3:0] state;

always @(posedge clk)
case(state)
    4'b0000: if(TxD_start) state <= 4'b0100;
    4'b0100: if(BaudTick) state <= 4'b1000; // start
    4'b1000: if(BaudTick) state <= 4'b1001; // bit 0
    4'b1001: if(BaudTick) state <= 4'b1010; // bit 1
    4'b1010: if(BaudTick) state <= 4'b1011; // bit 2
    4'b1011: if(BaudTick) state <= 4'b1100; // bit 3
    4'b1100: if(BaudTick) state <= 4'b1101; // bit 4
    4'b1101: if(BaudTick) state <= 4'b1110; // bit 5
    4'b1110: if(BaudTick) state <= 4'b1111; // bit 6
    4'b1111: if(BaudTick) state <= 4'b0001; // bit 7
    4'b0001: if(BaudTick) state <= 4'b0010; // stop1
    4'b0010: if(BaudTick) state <= 4'b0000; // stop2
    default: if(BaudTick) state <= 4'b0000;
endcase

```

Observe como a máquina de estados inicia assim que "TxD_start" é ativado, mas apenas avança quando "BaudTick" é ativado.

Agora precisamos apenas gerar a saída "TxD":

```
reg muxbit;

always @(state[2:0])
case(state[2:0])
0: muxbit <= TxD_data[0];
1: muxbit <= TxD_data[1];
2: muxbit <= TxD_data[2];
3: muxbit <= TxD_data[3];
4: muxbit <= TxD_data[4];
5: muxbit <= TxD_data[5];
6: muxbit <= TxD_data[6];
7: muxbit <= TxD_data[7];
endcase

// combine os bits de start, dados e stop
assign TxD = (state<4) | (state[3] & muxbit);
```

O código completo, incluindo a geração do sinal de ocupado está na página da disciplina.

Módulo receptor RS-232

Vamos construir o receptor mostrado na Figura 4. A implementação funciona assim:

- O módulo monta dos dados a partir da linha RxD ao passo que chegam;
- Quando um byte está sendo recebido, ele aparece no "barramento de dados". Quando um byte completo for recebido, "data_ready" é atribuído por um clock.

Observe que "data" só é válido quando "data_ready" for ativado. No restante do tempo não devemos utilizá-lo.

Sobre-amostragem

Um receptor assíncrono tem de alguma forma ficar "sincronizado" com o sinal que está chegando (já que não tem acesso ao clock usado na transmissão). Para determinar quando um novo dado está chegando (bit start), sobre-amostramos o sinal em uma frequência múltipla da frequência do baudrate. Uma vez que o bit de start foi detectado, amostramos a linha do sinal na taxa de baud conhecida para adquirir o sinal. Os receptores tipicamente sobre-amostram o sinal de entrada em 16 vezes o baudrate. Aqui vamos usar 8 vezes, o que nos dá uma frequência de amostragem de 921600Hz. Vamos assumir então que há um sinal "Baud8Tick" que é ativado 921600 vezes por segundo.

O projeto

Primeiro, o sinal de entrada "RxD" não tem nenhuma relação com nosso clock. Vamos utilizar dos flip-flops tipo D para sobreamostrá-lo e sincroniza-lo com nosso clock.

```
reg [1:0] RxD_sync;
always @(posedge clk) if(Baud8Tick) RxD_sync <= {RxD_sync[0], RxD};
```

Vamos também filtrar os dados de forma que se aparecerem picos rápidos na linha RxD eles não sejam interpretados errados como bits de start.

```

reg [1:0] RxD_cnt;
reg RxD_bit;

always @(posedge clk)
if(Baud8Tick)
begin
  if(RxD_sync[1] && RxD_cnt!=2'b11) RxD_cnt <= RxD_cnt + 1;
  else
    if(~RxD_sync[1] && RxD_cnt!=2'b00) RxD_cnt <= RxD_cnt - 1;

    if(RxD_cnt==2'b00) RxD_bit <= 0;
    else
      if(RxD_cnt==2'b11) RxD_bit <= 1;
end

```

Uma máquina de estado nos permite ir acompanhando a cada bit que é recebido, uma vez que um "start" foi detectado.

```

reg [3:0] state;

always @(posedge clk)
if(Baud8Tick)
case(state)
  4'b0000: if(~RxD_bit) state <= 4'b1000; // achou o bit de start?
  4'b1000: if(next_bit) state <= 4'b1001; // bit 0
  4'b1001: if(next_bit) state <= 4'b1010; // bit 1
  4'b1010: if(next_bit) state <= 4'b1011; // bit 2
  4'b1011: if(next_bit) state <= 4'b1100; // bit 3
  4'b1100: if(next_bit) state <= 4'b1101; // bit 4
  4'b1101: if(next_bit) state <= 4'b1110; // bit 5
  4'b1110: if(next_bit) state <= 4'b1111; // bit 6
  4'b1111: if(next_bit) state <= 4'b0001; // bit 7
  4'b0001: if(next_bit) state <= 4'b0000; // bit de stop
  default: state <= 4'b0000;
endcase

```

Observe que usamos um sinal "next_bit" para ir de bit em bit. Este sinal informa o momento em que devemos observar o bit, normalmente no centro do bit.

```

reg [2:0] bit_spacing;

always @(posedge clk)
if(state==0)
  bit_spacing <= 0;
else
  if(Baud8Tick)
    bit_spacing <= bit_spacing + 1;

wire next_bit = (bit_spacing==7);

```

Finalmente um registrador de deslocamento coleta os bits de dados conforme eles chegam.

```

reg [7:0] RxD_data;
always @(posedge clk) if(Baud8Tick && next_bit && state[3]) RxD_data <= {RxD_bit,
RxD_data[7:1]};

```

Interface serial da placa DE-2

O esquemático da parte de comunicação serial da placa DE-2 pode ser visto na Figura 9.

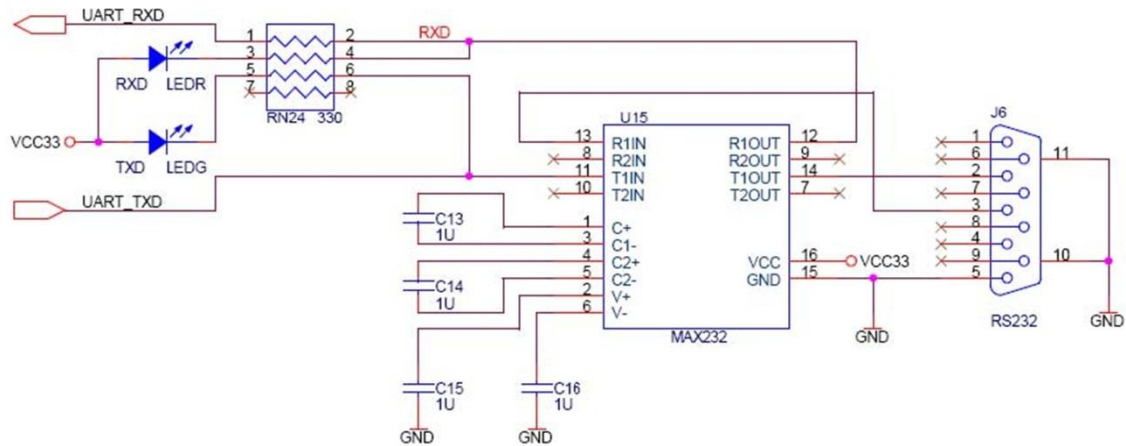


Figura 9. Circuito da porta serial RS-232 da placa DE-2.

Nome do sinal	Descrição
UART_RXD	Pino de recepção
UART_TXD	Pino de transmissão

Podemos observar que foi utilizado o circuito integrado MAX232, que é responsável por produzir os níveis de tensão no padrão RS232 para o pino TxD e converter a tensão da entrada RxD em uma tensão compatível com o FPGA, no caso 3,3V.

Interface entre o processador projetado e a porta serial

Para utilizarmos o circuito das Figuras 4 e 5, primeiro devemos observar que para o circuito que vai utilizar a interface serial dentro do FPGA, no caso, nosso computador, o circuito aparece como uma interface paralela. Esta interface paralela pode ser conectada ao computador projetado de forma similar aos exemplos mostrados nas Figuras 2 e 3.

Interface para transmissão serial

Para conectar o circuito da Figura 4, precisamos de uma porta de saída de 7 bits o TxD_data, somado a 1 bit para o TxD_start, e de uma porta de entrada de 1 bit, para o sinal de busy. Como o nosso barramento de dados é de 8 bits, o TxD_data e o TxD_start podem ser enviados juntos. Por exemplo, podemos montar o dado a ser enviado como :

Start	B6	B5	B4	B3	B2	B1	B0
-------	----	----	----	----	----	----	----

A detecção de ocupado é utilizada para identificarmos quando a transmissão finalizou. Apenas após a finalização da transmissão podemos enviar outro byte. A leitura pode ser feita no mesmo endereço utilizado para a escrita e vamos ler apenas um bit. É sugerido ligar o sinal de busy no bit 0 (bit menos significativo)

nc	nc	nc	nc	nc	nc	nc	busy
----	----	----	----	----	----	----	------

O circuito de transmissão deve ser habilitado no endereço 64 em decimal (40h).

O processo de envio de um byte pela serial deve então seguir os seguintes passos:

Passo 1: Ler o endereço 40h da memória e armazenar em um registrador;

Passo 2: Desviar para o passo 1 caso o bit menos significativo do registrador seja 1;

Passo 3: Escrever uma palavra na posição 40h onde os sete bits menos significativos formam o dado a ser enviado juntamente com o bit de start (bit 7) em 1;

Passo 4: Enviar uma palavra com o bit de start em 0.

Passo 5: Repetir todo o processo para os outros bytes a serem enviados.

Interface para a recepção serial

Para recebermos um valor pela porta serial devemos observar que só devemos ler o valor de `RxD_data` quando `RxD_ready` for ativo. O problema é que este sinal só é ativo por um ciclo de clock (clock da interface serial, não clock do processador). Para que o processador tenha tempo de ler este bit e para garantir que cada byte seja lido apenas uma vez, devemos melhorar o nosso circuito de decodificação de endereço e habilitação para incluir um circuito que armazene o bit `RxD_ready` (em um flip-flop) até que haja a leitura de `RxD_Data`. Ao lermos `RxD_data` podemos limpar este flip-flop.

Com este indicador de recepção estabilizado, podemos compor uma palavra de leitura de 8 bits formada pela `RxD_ready_est` (estabilizado) e pelos 8 bits de `RxD_Data`, assim:

<code>RxD_Ready_est</code>	B6	B5	B4	B3	B2	B1	B0
----------------------------	----	----	----	----	----	----	----

B_i são os bits de `RxD_data`. O bit `RxD_Ready_est` fica na posição do bit mais significativo por ser mais fácil ser testado.

O circuito de transmissão deve ser habilitado no endereço 128 em decimal (80h).

O processo de envio de um byte pela serial deve então seguir os seguintes passos:

Passo 1: Ler o endereço 60h da memória e armazenar em um registrador;

Passo 2: Desviar para o passo 1 caso o bit mais significativo do registrador seja 0 (bit N), indicando que nenhum byte chegou;

Passo 3: Utilizar o byte contido nos bits menos significativos do registrador;

Passo 5: Repetir todo o processo para os outros bytes a serem recebidos.

Questões de Preparação

1. Como você colocaria, utilizando um programa, o valor FFh em um registrador da CPU?
2. Desenhe o circuito completo para transmissão serial. Baseie-se nas Figuras de 2, 3 e 4. Escreva a descrição Verilog deste circuito. Esta descrição deve instanciar o transmissor serial.
3. Desenhe o circuito completo para recepção serial. Baseie-se nas Figuras de 2, 3 e 5. Escreva a descrição Verilog deste circuito. Esta descrição deve instanciar o receptor serial. Lembre que esta descrição tem uma pequena parte sequencial para estabilizar o sinal `RxD_ready`.