

Projeto de Sistemas Digitais

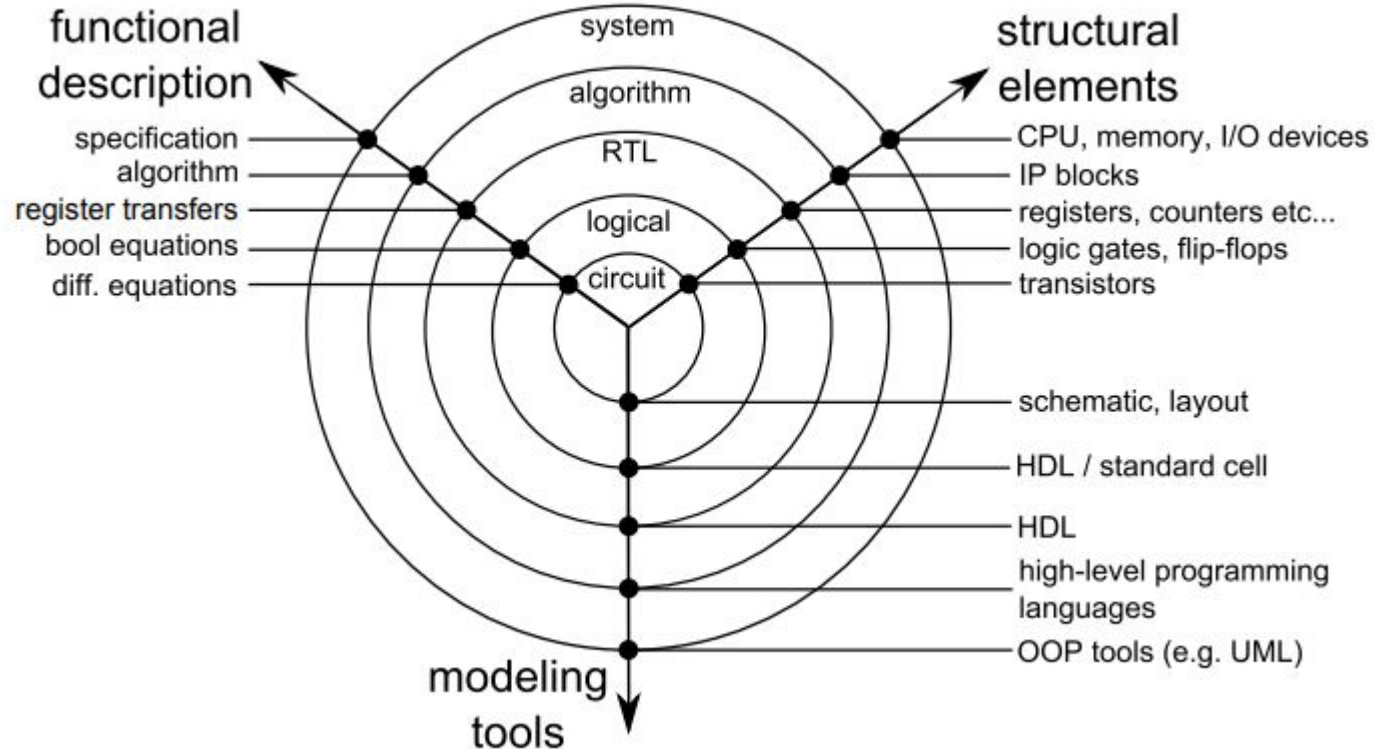
Introdução

Henrique do Nascimento Cunha, MSc.

Introdução

- Níveis de abstração
- Métodos de descrição do projeto
- SystemVerilog
 - Exemplos
 - Meio somador
 - Somador Completo
 - Somador de 4 bits

Níveis de abstração



Níveis de abstração

- **System Level:** Definição de partições e suas interfaces
- **Algorithm (ou Behavioral) Level:** Modelagem comportamental com linguagem de programação de alto nível
- **RTL (Register Transfer Level):** Define a microarquitetura e separação entre controle e *datapath*
- **Logic (ou Gate) Level:** O projeto é descrito como uma netlist de portas lógicas (AND, OR, NOT, etc.) e elementos de armazenamento
- **Circuit (ou Layout) Level:** No nível de *layout*, o projeto é definido como uma rede de portas e registradores instanciados a partir de uma biblioteca de tecnologia, que contém informação de atraso (próprio da tecnologia) para cada porta

Métodos de descrição do projeto

- Diagrama esquemático (ou de blocos)
- Vantagens:
 - Descrição em formato visual
 - Pode ser formal e simulável
 - Fácil de entender
- Muitas desvantagens para sistemas complexos
 - Difícilimo de depurar
 - Fica simplesmente grande demais para dar conta de todos os aspectos

Métodos de descrição do projeto

- Descrição com HDL
 - Descreve um sistema digital de forma textual
 - Sua simulação pode ser automatizada dentro do fluxo de projeto
 - Permite a entrada em todos os níveis de abstração, exceto o nível de sistema (SystemVerilog)
 - Sistemas complexos agora podem ser descritos em alguns milhares de linhas de código
 - Depuração menos dolorosa

SystemVerilog

- Todo ASIC (*Application Specific Integrated Circuit*) é projetado usando
 - SystemVerilog
 - Mais simples e poderosa
 - Parece um pouco com C
 - Abrange vários níveis de abstração
 - VHDL
 - Nível de abstração a partir do RTL
 - Complicada
- Vamos usar SystemVerilog
 - Um subconjunto de SystemVerilog pode ser **sintetizado** para produzir uma *netlist* para FPGA
 - Como usar?

SystemVerilog

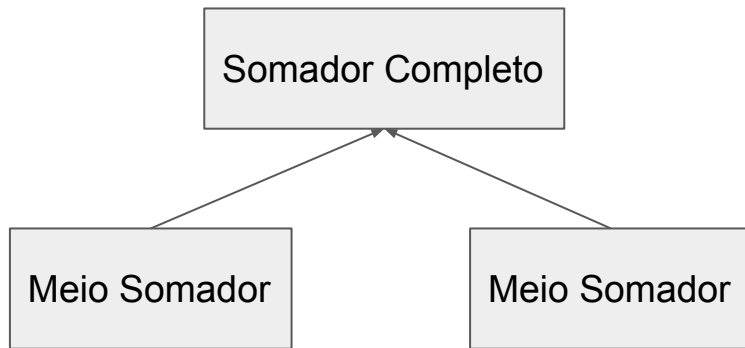
- Como usar?
 - O projeto é descrito usando SystemVerilog
 - Simulado exhaustivamente para verificar a funcionalidade
 - Sintetizado
 - Análise temporal
 - O resultado da síntese é simulado novamente para garantir que os possíveis erros introduzidos pela síntese sejam corrigidos
 - Aí sim, partimos para o *layout*

SystemVerilog

- Como usar?
 - O projeto é descrito usando SystemVerilog
 - Simulado exhaustivamente para verificar a funcionalidade
 - Sintetizado
 - **Análise temporal**
 - O resultado da síntese é simulado novamente para garantir que os possíveis erros introduzidos pela síntese sejam corrigidos
 - Aí sim, partimos para o *layout*

SystemVerilog

- Vamos começar com um projeto simples
 - Soma binária
 - Três estruturas
 - Meio-somador
 - Somador completo
 - Somador de vários bits (*Ripple Carry*)

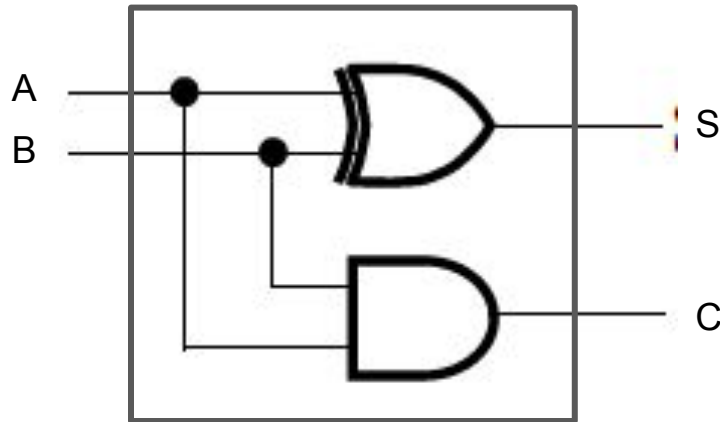


SystemVerilog

- Como implementar um meio-somador:

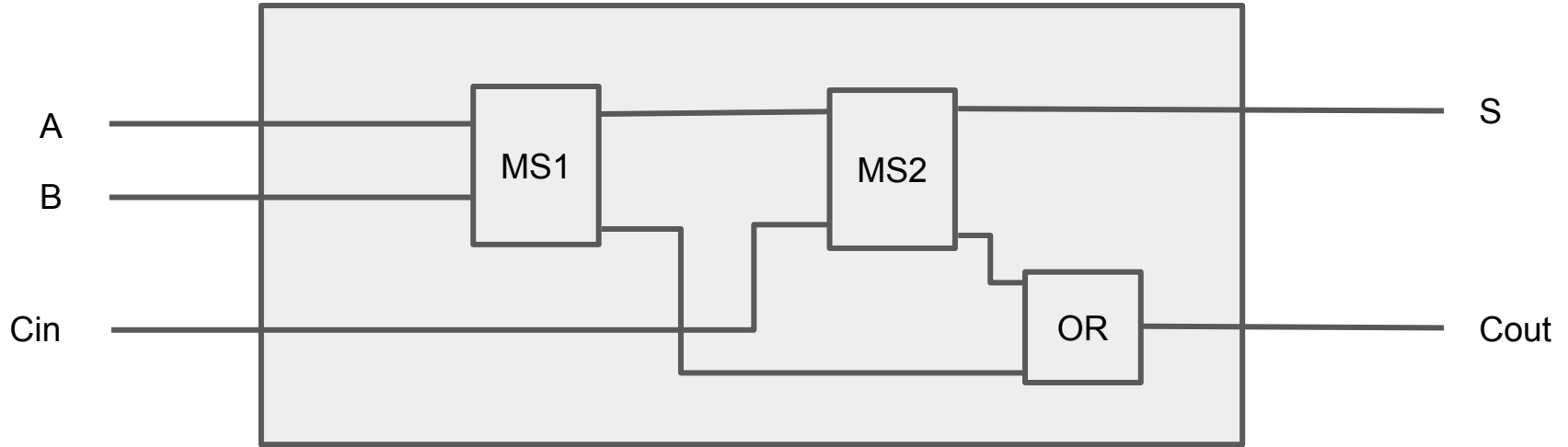
$$S = A \oplus B$$

$$C = A \cdot B$$



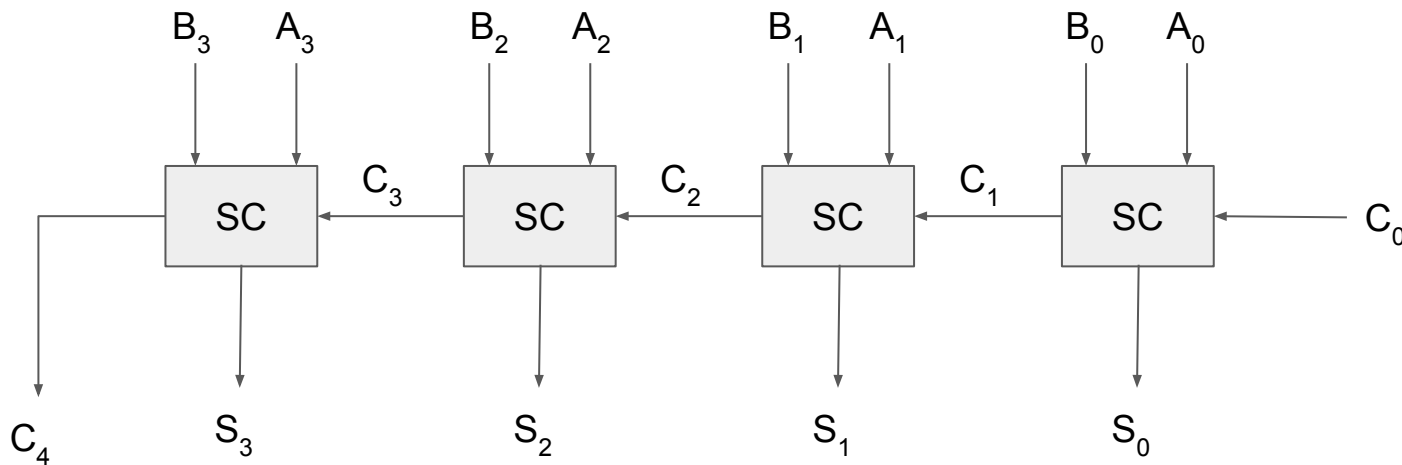
SystemVerilog

- Somador Completo



SystemVerilog

- Somador *Ripple-Carry* de 4 bits a partir de 4 somadores completos



SystemVerilog

- Meio Somador em SV:

```
module meio_somador(a, b, s, c);  
  
    input a, b;  
  
    output s, c;  
  
    assign s = a ^ b;  
  
    assign c = a & b;  
  
endmodule
```

SystemVerilog

- Somador completo a partir de meio-somador:

```
module somador_completo(a, b, cin, s, cout);  
    input a, b, cin;  
  
    output s, cout;  
  
    wire sm, cm, ct  
  
    meio_somador MS1(a, b, sm, cm), MS2(sm, cin, s, ct);  
  
    assign c = ct | cm;  
  
endmodule
```

SystemVerilog

- Somador 4 bits *Ripple-Carry*:

```
module somador_4(b, a, c0, s, c4);  
    input [3:0] b, a;  
    input c0;  
    output [3:0] s;  
    output c4;  
    wire [3:1] c;  
    somador_completo SC0(b[0], a[0], c0, s[0], c[1]),  
                      SC1(b[1], a[1], c[1], s[1], c[2]),  
                      SC2(b[2], a[2], c[2], s[2], c[3]),  
                      SC3(b[3], a[3], c[3], s[3], c4);  
  
endmodule
```


SystemVerilog

- Atividade:
 - Projete e implemente:
 - Um meio subtrator
 - Um subtrator completo
 - Um subtrator de 4 bits

SystemVerilog

- Tipos de dados básicos:
 - `wire`: Declara um fio ou barramento
 - Declaração:
 - `wire a; // 1 fio chamado "a"`
 - `wire[31:0] a; // barramento a de 32 bits`
 - `wire[7:0] rgb[2:0]; // um array de barramentos`
 - Uso:
 - Conectar componentes de um design
 - Pode ser lido
 - Não pode ser escrito em uma função ou bloco
 - Não armazena valor
 - Pode assumir um valor por meio de lógica combinacional
 - `wire a;`
 - `assign a = c | d`

SystemVerilog

- Tipos de dados básicos:
 - reg: Declara uma variável que guarda seu valor entre atribuições procedurais. Não significa necessariamente um registrador físico.
 - Declaração:
 - `reg a; // 1 registrador chamado "a"`
 - `reg [31:0] a; // registrador de 32 bits`
 - `reg [7:0] rgb[2:0]; // um array de registradores de 8 bits`
 - Uso:
 - Guardar valores
 - Pode ser lido
 - Pode ser escrito em função ou bloco
 - Pode ser usado para exercer a função de um registrador físico

SystemVerilog

- Bloco always
 - Serve para descrever eventos que devem acontecer sob determinadas condições
 - Definição:

```
always @(lista_sensibilidade_1, lista_sensibilidade_1, etc)
begin
    /* Este bloco de sentenças é ativado sempre que qualquer
    das variaveis lista_sensibilidade muda de valor */
end
```

SystemVerilog

- Bloco always - Exemplo

```
module mux2to1 (f, a, b, sel);  
    output f;  
    input a, b, sel;  
    reg f;  
    always @(a or b or sel)  
        f= ~sel ? a : b;  
endmodule
```

SystemVerilog

- Bloco always - Exemplo

```
module mux2to1 (f, a, b, sel);  
    output f;  
    input a, b, sel;  
    reg f;  
    always @(a or b or sel)  
        f = ~sel ? a : b;  
endmodule
```

Lista de sensibilidade



SystemVerilog

- Podemos utilizar sentenças procedurais com estilo de linguagem de alto nível (HLL) para implementar lógica.
- Considere o exemplo do mux 2 para 1

```
module mux(f, a, b, sel);  
    output f;  
    input a, b, sel;  
    reg f;  
    always@(a or b or sel)  
        if(~sel)  
            f = a;  
        else  
            f = b;  
endmodule
```

A execução disso não é como a de uma linguagem de programação imperativa de alto nível.

O sintetizador vai olhar para isso e decidir qual circuito melhor implementa essa condição.

SystemVerilog

- Atribuição procedural
- No interior de um bloco initial ou always
 `sum= a + b + cin;`
- Tal como C: O lado direito é avaliado e atribuído ao lado esquerdo antes que a próxima sentença seja avaliada
- A sentença do lado direito pode conter fios (wires) e/ou regs
- A do lado esquerdo tem que ser um reg
- (apenas primitivas ou atribuições contínuas podem atribuir valores a fios)

SystemVerilog

- Projeto assíncrono

```
always @ (reg_1,reg_2,reg_3)
begin
    \\ Sentenças aqui
end
```

Usado para definir lógica combinacional e latches

SystemVerilog

- case

```
always@(*)
    case(net)
        valor_1:
            begin
                acao_1;
                // ...
            end
        valor_2:
            begin
                acao_2;
            end
    endcase
```

SystemVerilog

- Exemplo de case

```
module decode7seg (input [3:0] entrada,  
output reg [0:6] saída);  
    always @ (entrada)  
    begin  
        case (entrada)  
            4'd0: saída= 7'h7E;  
            4'd1: saída= 7'h30;  
            .  
            .  
            `4'd9: saída = 7'b1111011  
        end  
    endmodule
```

SystemVerilog

Operadores Aritméticos	
+	Adição
-	Subtração
*	Multiplicação
/	Divisão
%	Módulo

SystemVerilog

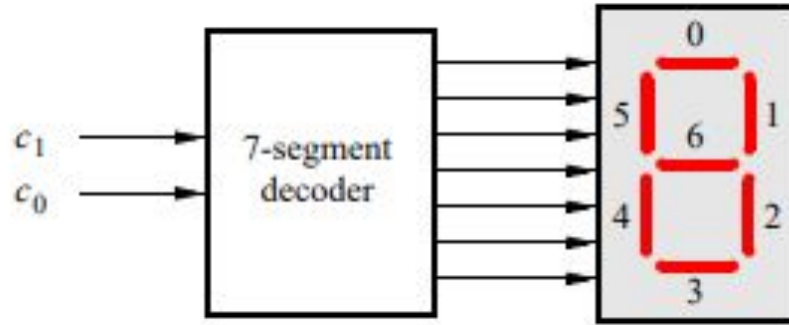
Operadores Booleanos	
&&	AND lógico
!	Not lógico
	OR lógico
&	AND bit-a-bit
~	NOT bit-a-bit
	OR bit-a-bit
^	XOR bit-a-bit

SystemVerilog

Operadores Unários	
&(variável)	AND unário
~(variável)	NOT unário
(variável)	OR unário
^(variável)	XOR unário

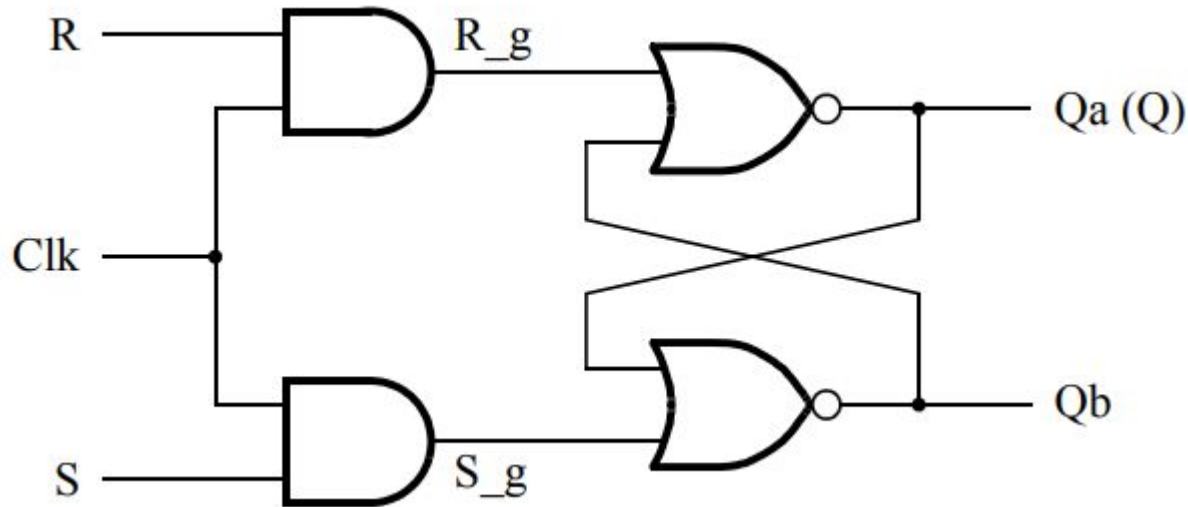
SystemVerilog

- Atividade 1
- Mostrar caracteres nos displays de 7 segmentos



SystemVerilog

- Um latch é um elemento de armazenamento sensível ao nível de uma entrada
- Como fazer latches?



SystemVerilog

```
// A gated RS latch
module part1(Clk, R, S, Q);
    input Clk, R, S;
    output Q;
    wire R_g, S_g, Qa, Qb /* synthesis keep */ ;
    assign R_g = R & Clk;
    assign S_g = S & Clk;
    assign Qa = ~(R_g | Qb);
    assign Qb = ~(S_g | Qa);
    assign Q = Qa;
endmodule
```

SystemVerilog

```
// A gated RS latch
module part1(Clk, R, S, Q);
    input Clk, R, S;
    output Q;
    wire R_g, S_g, Qa, Qb /* synthesis keep */ ;
    and(R_g, R, Clk);
    and(S_g, S, Clk);
    nor(Qa, R_g, Qb);
    nor(Qb, S_g, Qa);
    assign Q = Qa;
endmodule
```

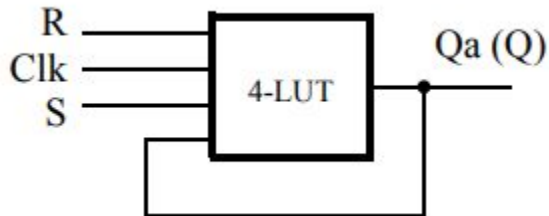
SystemVerilog

```
// A gated RS latch
module part1(Clk, R, S, Q);
    input Clk, R, S;
    output Q;
    wire R_g, S_g, Qa, Qb /* synthesis keep */;
    and(R_g, R, Clk);
    and(S_g, S, Clk);
    nor(Qa, R_g, Qb);
    nor(Qb, S_g, Qa);
    assign Q = Qa;
endmodule
```

O que é isso?

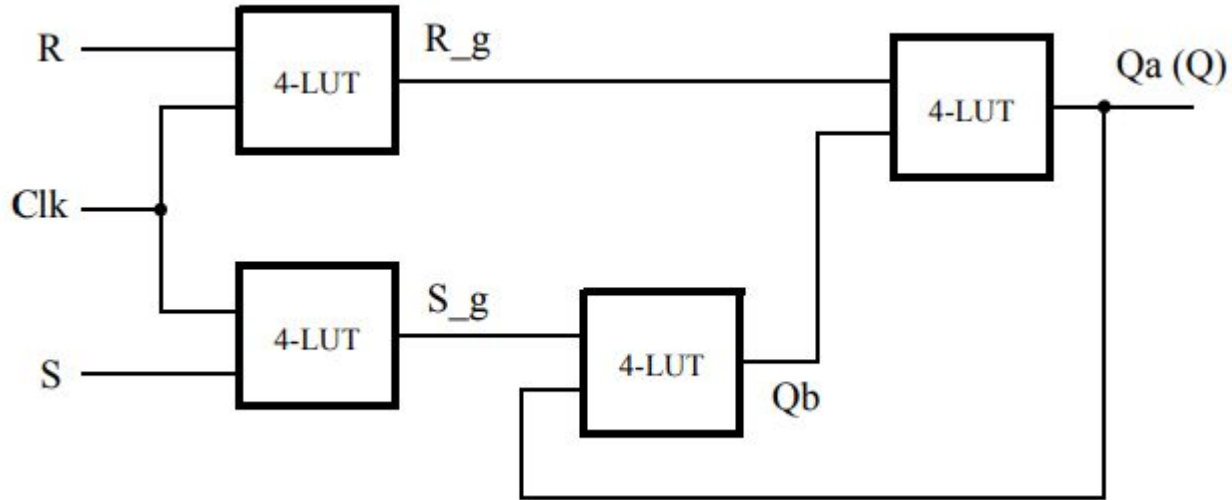
SystemVerilog

- `/* synthesis keep */`
 - É uma diretiva de compilação
 - Ela serve para tornar os sinais internos disponíveis para observação
 - Se você não colocar, o sintetizador da Altera vai usar uma 4-LUT (LookUp table de 4 entradas) para implementar o circuito. Assim:



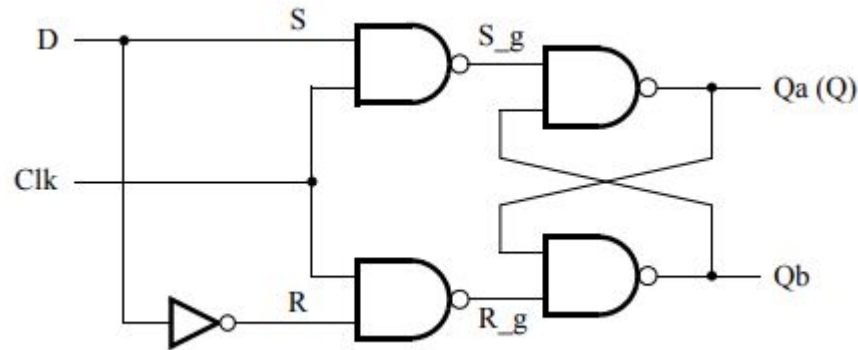
SystemVerilog

- `/* synthesis keep */`
 - Ao colocar essa diretiva, seu circuito fica assim:



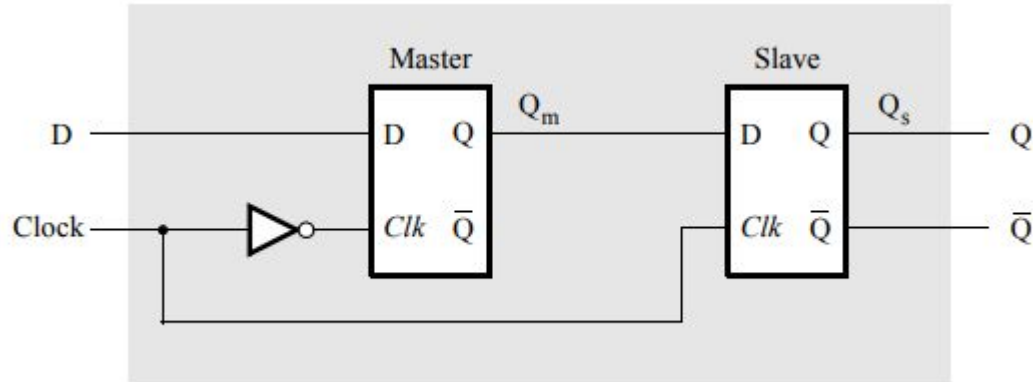
SystemVerilog

- Exercício: Construa e simule um latch tipo D



SystemVerilog

- Flip-flop *master-slave* tipo D



SystemVerilog

- Projeto síncrono
 - Além dos registradores, precisamos definir a borda que ativará o circuito sequencial

```
// Borda de subida  
always @ (posedge clk)
```

```
// Borda de descida  
always @ (negedge clk)
```

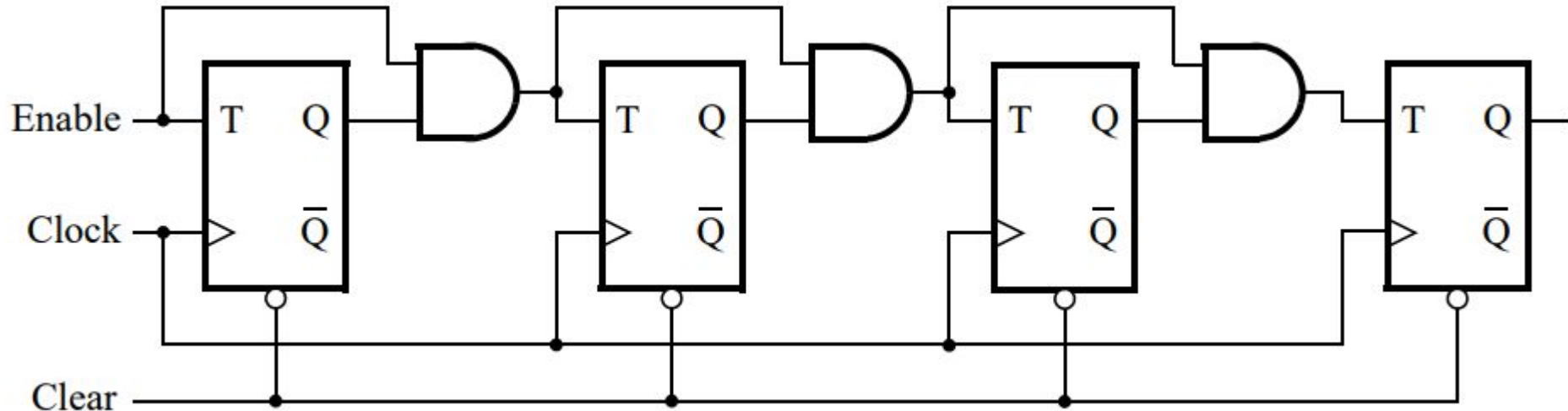

SystemVerilog

- Projeto síncrono
 - Em blocos always só podemos manipular regs
 - Entradas, saídas e fios são nets. Estes não armazenam valores. Para armazenar valores é preciso defini-los novamente como reg.

```
// Saída registrada  
output reg saida_1;
```

SystemVerilog

- Contador



SystemVerilog

- Como fazer o sintetizador inferir um contador?

...

```
reg cont;
```

...

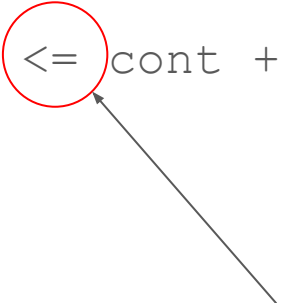
```
cont <= cont + 1;
```

...

SystemVerilog

- Como fazer o sintetizador inferir um contador?

```
...  
reg cont;  
...  
cont <= cont + 1;  
...
```



O que é isso?
É um tipo de operador de
atribuição diferente do que já
vimos.

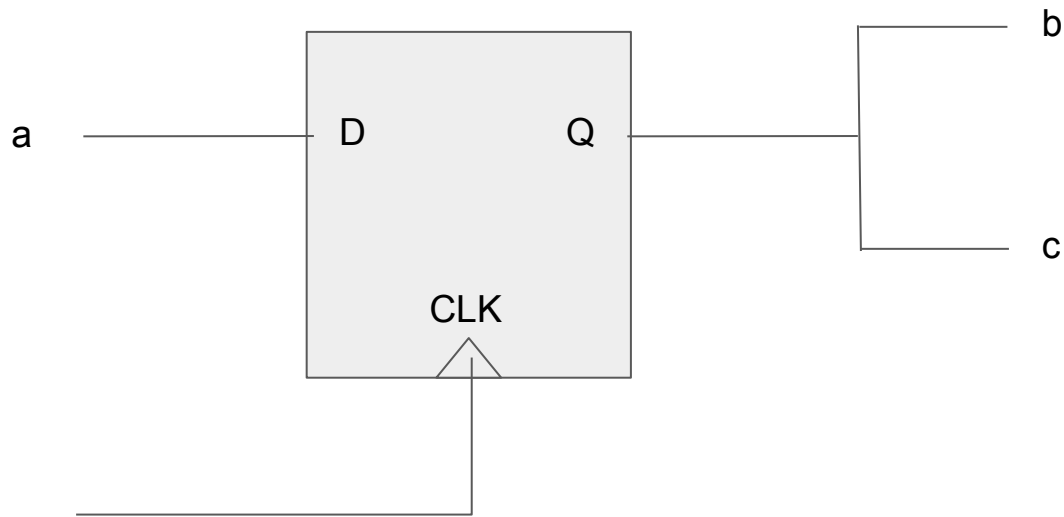
SystemVerilog

- Atribuição bloqueante

```
module blocking (clk,a,c);  
    input clk;  
    input a;  
    output c;  
    wire clk, a;  
    reg c, b;  
    always @ (posedge clk)  
        begin  
            b = a;  
            c = b;  
        end  
endmodule
```

SystemVerilog

- Atribuição bloqueante: Que hardware é gerado por isso?



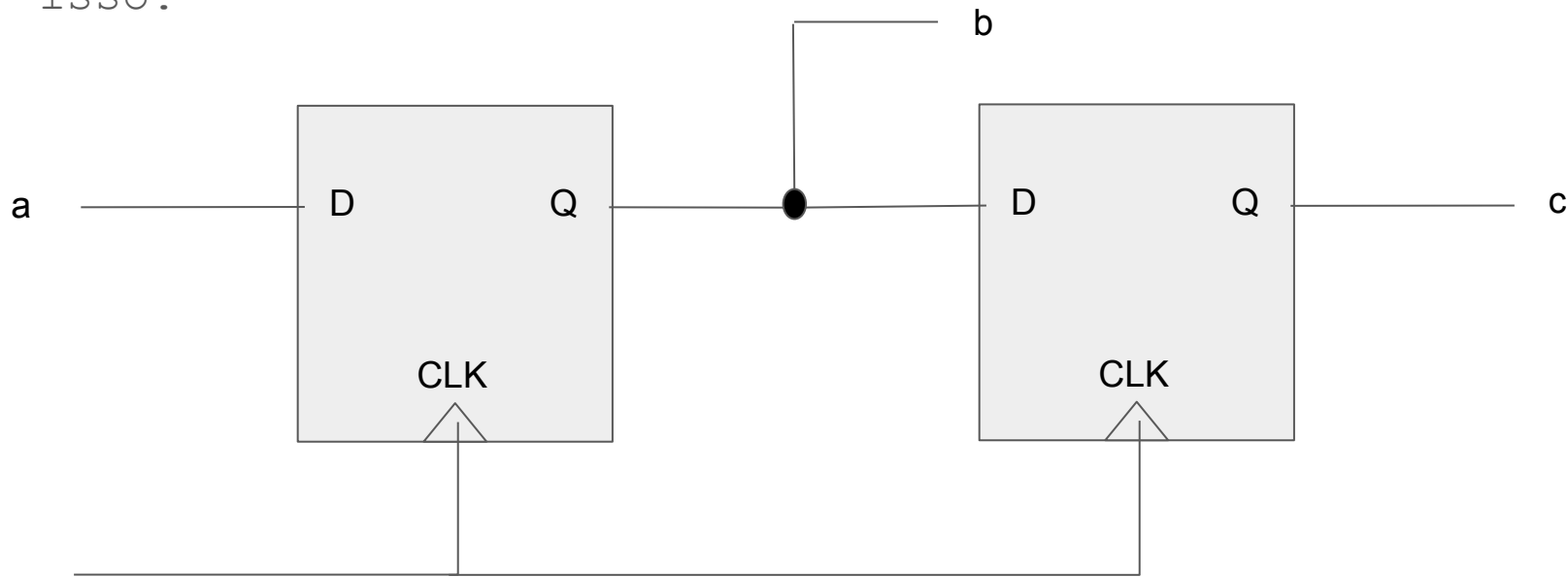
SystemVerilog

- Atribuição não bloqueante

```
module non_blocking (clk,a,c);  
    input clk;  
    input a;  
    output c;  
    wire clk, a;  
    reg c, b;  
    always @ (posedge clk)  
        begin  
            b <= a;  
            c <= b;  
        end  
endmodule
```

SystemVerilog

- Atribuição não bloqueante: Que hardware é gerado por isso?



SystemVerilog

- Contador de 8 bits

```
module counter8(clk, rst_n, cont);  
    input clk, rst_n;  
    output [7:0] cont;  
    reg [7:0] cont;  
    always@(posedge clk)  
    begin  
        if (!rst_n)  
            cont <= 0;  
        else  
            cont <= cont + 1;  
        end  
    endmodule
```

SystemVerilog

- Clock da placa
 - 27MHz
 - 50MHz
- Muito rápido para ver funcionando
- Precisamos diminuir essa velocidade
- Como?
 - Divisor de frequência
 - Várias formas de implementar
 - Supondo um clock de 50MHz a mais simples é ter um (no nosso caso segundo) contador que vai contar 50 milhões de pulsos de clock antes de dar rodar o restante do bloco always 1 vez
 - Dessa forma, teremos um clock de 1 segundo

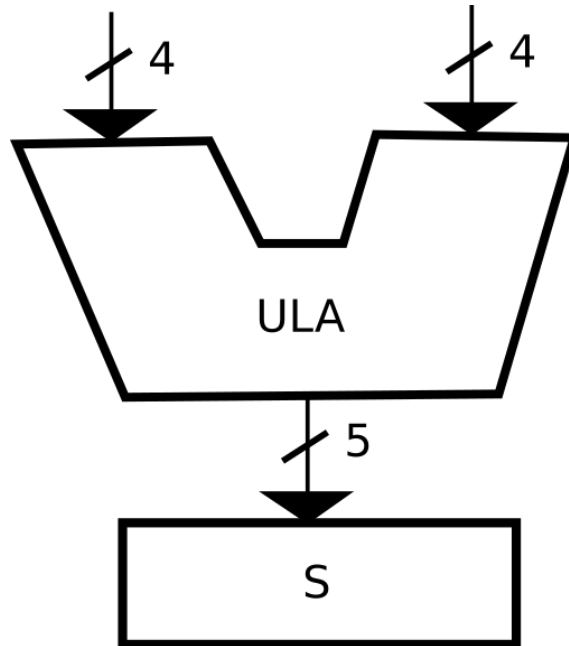
SystemVerilog

- Contador de 8 bits

```
module counter8(clk, rst_n, cont);  
    // Mesma definição de inputs outputs do anterior  
    reg [24:0] cont2;  
    always@(posedge clk)  
        begin  
            if (!rst_n)  
                begin  
                    cont <= 0;  
                    cont2 <= 0; // Divisor de frequencia  
                end  
            else  
                begin  
                    // Conta 50 milhoes antes de incrementar  
                    if (cont2 == 25'd50000000)  
                        begin  
                            cont <= cont + 1;  
                            cont2 <= 0;  
                        end  
                    else  
                        cont2 <= cont2 + 1;  
                    end  
                end  
        end  
end  
endmodule
```

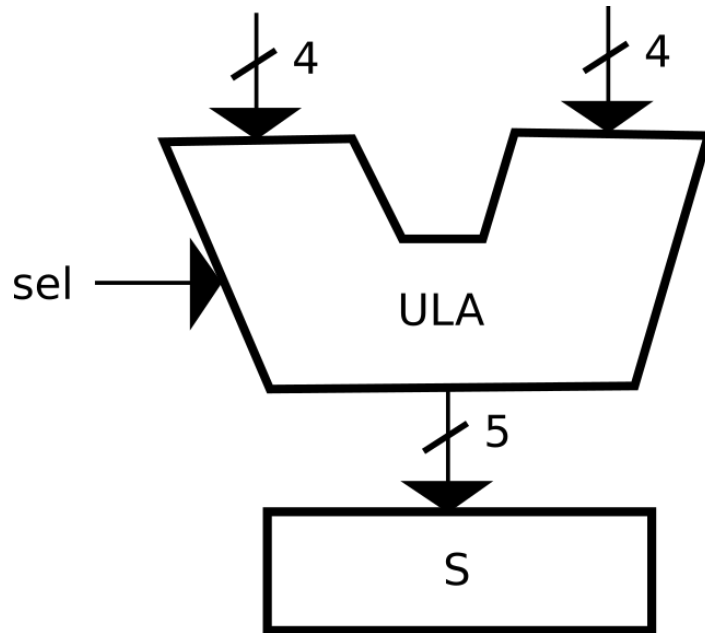
SystemVerilog

- ULA com saída registrada
 - Primeiro faremos um somador com saída registrada



SystemVerilog

- ULA com saída registrada
 - O que precisamos fazer para adicionar a operação de subtração?



SystemVerilog

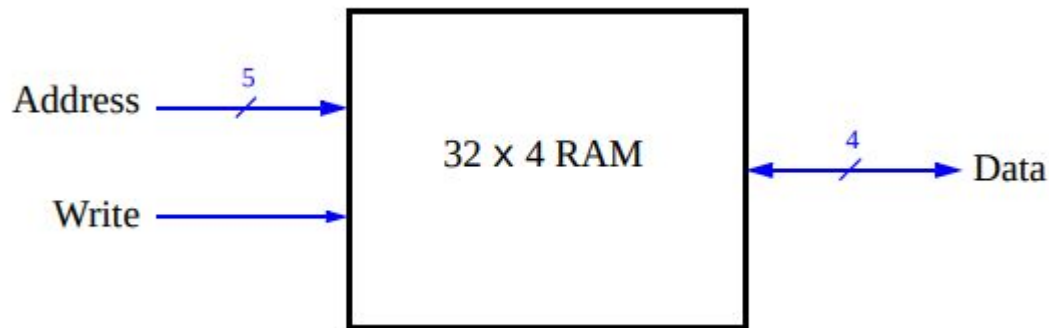
- O tipo enum
 - É possível declarar um tipo de dados enumerável
 - Serve para restringir um tipo de dados a valores conhecidos e nomeáveis
 - Exemplos:
 - `enum {RED, GREEN, BLUE} color;`
 - Nesse caso RED terá valor 0, GREEN terá valor 1 e BLUE terá valor 2
 - `enum reg [1:0] {idle=2'b00, start=2'b01, wait=2'b10, stop = 2'b11} states;`
 - `typedef enum {red, yellow, green} colors_t;`
 - `colors_t lightsRoadA, lightsRoadB;`

SystemVerilog

- Máquinas de estados finitas
 - Loops não tem o mesmo uso em HDL em relação a linguagens de programação imperativas
 - Além disso, algumas operações devem ser divididas em vários ciclos de clock
 - Isso pode deixar as operações mais ineficientes
 - Porém economiza área do chip
 - Usando dispositivos altera, é possível inferir vários tipos de FSM
 - [Mealy state machine](#)
 - [Moore state machine](#)
 - [Safe state machine](#)
 - [User encoded machine](#)
 - [All synchronous safe state machine](#)
- Atividade:
 - Ver como cada máquina de estados funciona e alterar seus estados para usar Enum

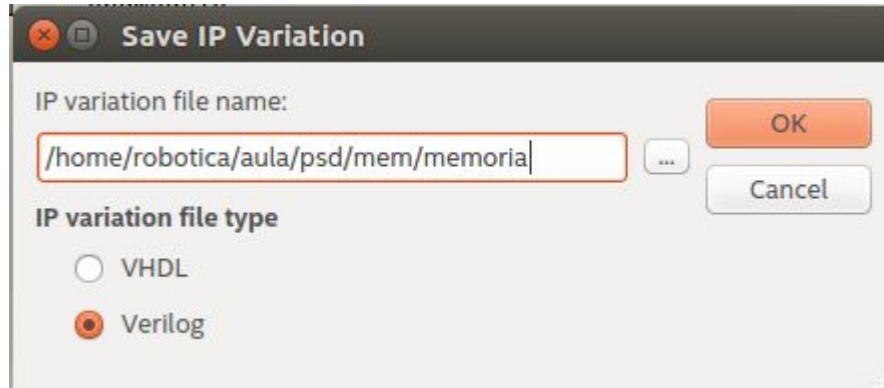
SystemVerilog

- Como implementar memória?
 - A placa que dispomos inclui blocos M10K, onde cada um contém 10240 bits de memória
 - Existem outros tipos de blocos em outros tipos de placa. Consulte antes de usar!
 - Um termo comum para uso de memória é o seu *aspect ratio*: largura (em bits) x profundidade (em palavras)

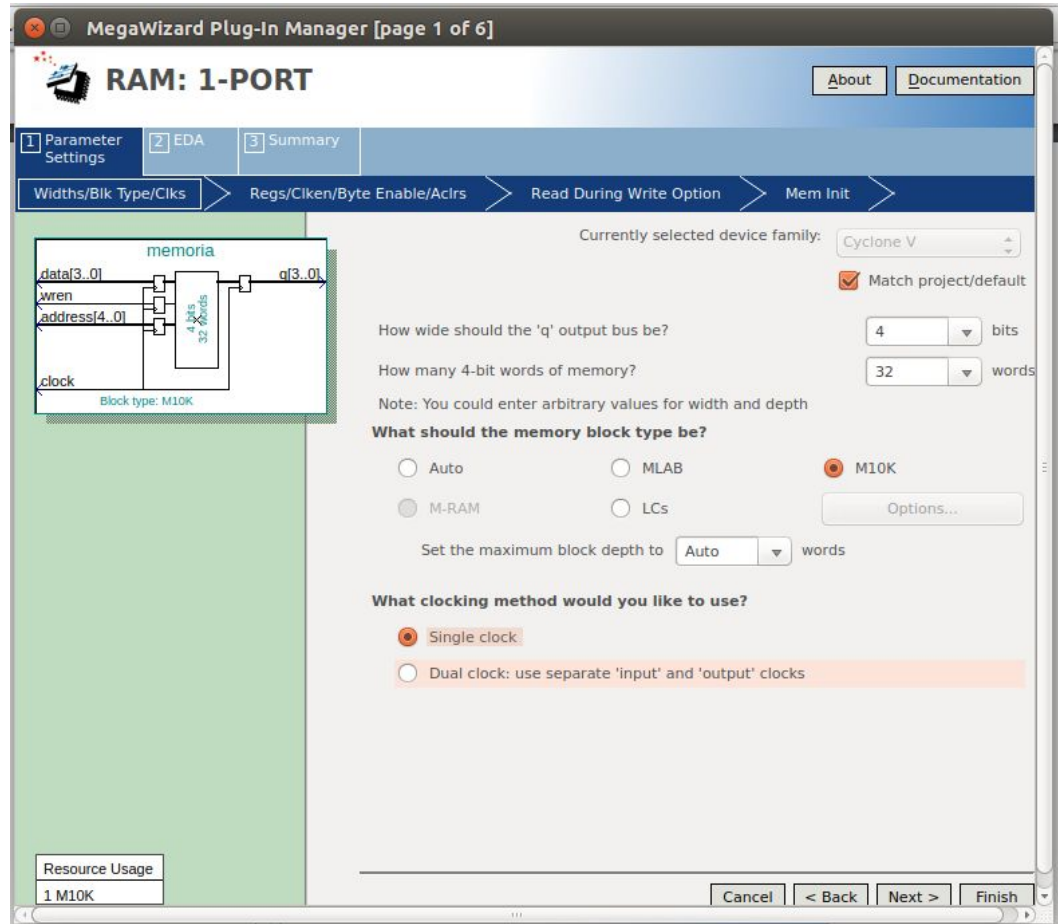


SystemVerilog

- Como implementar memória?
 - Podemos inferir a memória
 - Ou podemos criá-la na mão, usando o catálogo de IPs da Altera
 - Tools>IP Catalog
 - Ele deve abrir uma janela no canto direito (se já não estiver aberta)
 - Basic Functions>On Chip Memory>RAM: 1-PORT



SystemVerilog



SystemVerilog

MegaWizard Plug-In Manager [page 2 of 6]

RAM: 1-PORT

About Documentation

1 Parameter Settings 2 EDA 3 Summary

Widths/Bik Type/Ckls > Regs/Cken/Byte Enable/Aclrs > Read During Write Option > Mem Init >

data[3..0]

wren

address[4..0]

clock

q[3..0]

memoria

4 bits

32 words

Block type: M10K

Which ports should be registered?

- ☒ 'data' and 'wren' input ports
- ☒ 'address' input port
- ☒ 'q' output port

Create one clock enable signal for each clock signal.
☐ Note: All registered ports are controlled by the enable signal(s) More Options...

Create byte enable for port A
☐

What is the width of a byte for byte enables? 8 bits

Create an 'aclr' asynchronous clear for the registered ports
☐ More Options...

Create a 'rden' read enable signal
☐

Resource Usage

1 M10K

Cancel < Back Next > Finish

SystemVerilog

MegaWizard Plug-In Manager [page 3 of 6]

RAM: 1-PORT

[About](#) [Documentation](#)

1 Parameter Settings 2 EDA 3 Summary

Widths/Blk Type/Clocks > Regs/Clock/Byte Enable/Aclrs > **Read During Write Option** > Mem Init >

Diagram illustrating the RAM block configuration:

Block type: M10K

Single Port Read-During-Write Option

What should the q output be when reading from a memory location being written to?

▼

☒ Get x's for write masked bytes instead of old data when byte enable is used

Resource Usage

1 M10K

SystemVerilog

MegaWizard Plug-In Manager [page 4 of 6]

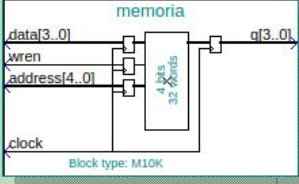
RAM: 1-PORT

[About](#) [Documentation](#)

1 Parameter Settings 2 EDA 3 Summary

Widths/Blk Type/Cls > Regs/Clk/Byte Enable/Aclrs > Read During Write Option > **Mem Init** >

Diagram showing a RAM block (M10K) with inputs: data[3..0], wren, address[4..0], and clock. The output is q[3..0]. The block is labeled "memoria" and "Block type: M10K".



Do you want to specify the initial content of the memory?

- ☒ No, leave it blank
- ☐ Initialize memory content data to XX..X on power-up in simulation
- ☐ Yes, use this file for the memory content data
(You can use a Hexadecimal (Intel-format) File [.hex] or a Memory Initialization File [.mif])

[Browse...](#)

File name:

The initial content file should conform to which port's dimensions? PORT_A ▼

☐ Allow In-System Memory Content Editor to capture and update content independently of the system clock

The 'Instance ID' of this RAM is: NONE

Resource Usage

1 M10K

[Cancel](#) [< Back](#) [Next >](#) [Finish](#)

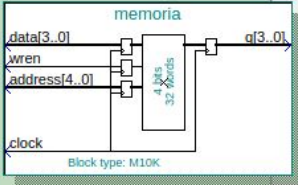
SystemVerilog

MegaWizard Plug-In Manager [page 5 of 6]

RAM: 1-PORT

[About](#) [Documentation](#)

1 Parameter Settings 2 EDA 3 Summary



Block type: M10K

Resource Usage

1 M10K

Simulation Libraries

To properly simulate the generated design files, the following simulation model file(s) are needed

File	Description
altera_mf	Altera megafunction simulation library

Timing and resource estimation

Generates a netlist for timing and resource estimation for this megafunction. If you are synthesizing your design with a third-party EDA synthesis tool, using a timing and resource estimation netlist can allow for better design optimization.

Not all third-party synthesis tools support this feature - check with the tool vendor for complete support information.


Note: Netlist generation can be a time-intensive process. The size of the design and the speed of your system affect the time it takes for netlist generation to complete.

☐ Generate netlist

Cancel < Back Next > Finish

SystemVerilog

MegaWizard Plug-In Manager [page 6 of 6]



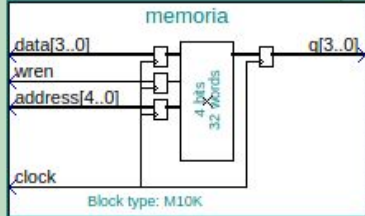
RAM: 1-PORT

[About](#)[Documentation](#)

1 Parameter Settings

2 EDA

3 Summary



Block type: M10K

Turn on the files you wish to generate. A gray checkmark indicates a file that is automatically generated, and a green checkmark indicates an optional file. Click Finish to generate the selected files. The state of each checkbox is maintained in subsequent MegaWizard Plug-In Manager sessions.

The MegaWizard Plug-In Manager creates the selected files in the following directory:
/home/robotica/aula/psd/mem/

File	Description
<input checked="" type="checkbox"/> memoria.v	Variation file
<input type="checkbox"/> memoria.inc	AHDL Include file
<input type="checkbox"/> memoria.cmp	VHDL component declaration file
<input type="checkbox"/> memoria.bsf	Quartus Prime symbol file
<input type="checkbox"/> memoria_inst.v	Instantiation template file
<input checked="" type="checkbox"/> memoria_bb.v	Verilog HDL black-box file

Resource Usage

1 M10K

Cancel

< Back

Next >

Finish

SystemVerilog

- Depois disso seu módulo de memória está pronto para ser instanciado e usado.
- Vamos fazer isso agora!
- Faça uma memória como descrito no tutorial
- Adicione um conteúdo nela com um arquivo .mif
- Crie um módulo top-level que instancia essa memória
- Leia o conteúdo da memória e escreva na saída do seu módulo
- Para colocar na placa, associe a pinagem correta para endereçar a memória com as chaves
- a cada ciclo de clock escreva o conteúdo daquele endereço nos LEDs ou no display de sete segmentos

SystemVerilog

- Como inferir memória?
 - O Quartus hoje oferece templates para vários tipos de circuitos que podem ser inferidos
 - Edit>Insert Template
 - Verilog HDL>Full Designs>RAMs and ROMs>Single Port RAM
 - No mesmo menu de templates dá pra ver como inicializar memória com origem em inferência
- Tarefa: Inferir, inicializar e simular memória RAM