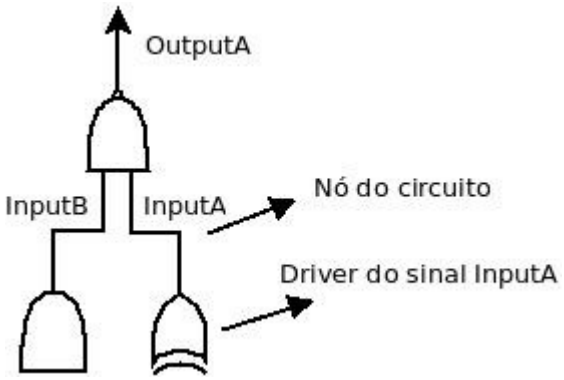
Pensando em Sinais ao Invés de Variáveis

Variáveis representam um estado que pode ser alterado em um algoritmo sequencial. O algoritmo sequencial, neste caso, diz respeito àquele que pode ser executado, passo a passo, por uma máquina de Turing. No mundo real, isto implica a presença de um dispositivo como um processador para executar tal algoritmo.

Já sinais se baseiam em um conceito usado em ferramentas EDA (*Eletronic Design Automation*) e em HDLs (*Hardware Description Languages*) que necessariamente representa um nó do circuito.



Neste contexto, dizemos que o *driver* de um sinal é a porta lógica responsável por controlar o valor lógico deste sinal (no CMOS, fisicamente, isto se traduz em controlar a tensão no fio). Na figura acima, identificamos 3 sinais se omitirmos as entradas das duas portas lógicas iniciais: InputA, InputB e OutputA. Note que é essencial que todo sinal tenha um, e apenas um *driver*.

Note que a diferença de pensar na lógica da solução de um problema em hardware e em software é bastante acentuada. A existência de linguagens de descrição de hardware, como VHDL, não diminui esta diferença, e o modo como se projeta hardware ou software possui fundamentos diferentes. Se é muito mais fácil resolver o problema com um algoritmo sequencial (usando variáveis e explorando sequencialidade) e é difícil projetar uma máquina de estados que realize esta sequencialidade, você deveria instanciar um processador[1] (memória, barramento e demais apetrechos) e passar o problema para a camada de software. Em hardware, nós temos que ter sempre em mente os nós do circuito (pelo menos dos sinais de entrada e saída) e os registradores envolvidos. Se você não tem uma idéia sólida de onde estão os registradores (flipflops) do seu projeto de hardware, é hora de repensar seu projeto. Quando for projetar o circuito (seja escrevendo o VHDL ou o esquemático), tenha sempre uma idéia clara destes elementos.

Guia para Escrever Processos VHDL RTL (para síntese de hardware)

RTL quer dizer *Register Transfer Level*. Isto significa, especificamente, que o nível de abstração que você deve trabalhar é aquele em que deixa explícito os registradores utilizados (através da criação de *process'* que inferem circuitos sequenciais, como veremos adiante) e a lógica que calcula valores entre

1 Após o curso de arquitetura de computadores, deverá ficar claro que um processador em si pode ser visto como uma máquina de estados (controle) e suas unidades funcionais (como a ALU). Escrever um programa de computador é precisamente a tarefa de ordenar uma sequencia de instruções para operar esta máquina.

registradores (através da criação de process que inferem circuitos combinacionais, como veremos abaixo). Este nível de abstração é alto o suficiente para que você não se preocupe em descrever o circuito digital com transistores ou portas lógicas. Por outro lado, é baixo o suficiente para que seja possível inferir hardware a partir da descrição.

EXEMPLO: Um nível mais alto que RTL é a modelagem do comportamento de um sistema, como por exemplo um emulador de videogame. O emulador imita perfeitamente o comportamento da plataforma de um console. Entretanto, não é possível inferir qual o hardware do console, porque o programador não organizou a descrição da plataforma de forma a deixar explícito todos os registradores e lógica intermediária. Ele escolheu uma linguagem de programação como C, pois não precisava dos formalismos de uma HDL, e implementou da maneira mais rápida possível, sem se preocupar também modelar detalhes da implementação em hardware.

Toda HDL oferece uma maneira de se descrever uma entidade em maneira algorítmica, como o *process* em VHDL, que pode ser interpretada pelo sintetizador e gerar hardware a partir da descrição. Chamamos este subconjunto da linguagem HDL que é sintetizável de *synthesizable subset*. Como o seu objetivo é descrever hardware em RTL, você deve se limitar a estas construções (note que uma HDL normalmente inclui um número muito maior de construções além do synthesizable subset, como alocação dinâmica, manipulação de arquivos, etc.).

Circuitos Combinacionais

Quando for escrever um processo, se o circuito é combinacional, especifique na *sensitivity list* todos os sinais que são usados como entrada. Qualquer sinal de saída que é definido dentro desse process (ou seja, você atribui um *driver* a ele) deve ser definido em todo caminho que a execução tomar. Por exemplo, se você utilizar um desvio condicional, criando um outro possível fluxo de execução, você deverá se assegurar que neste fluxo de execução os sinais de saída estão bem definidos. Exemplo:

```
-- Neste process os sinais saida1 e saida2 são definidos (driven)
exemplo1: process (a, b, c)
begin
  saida1 <= a and b;
  if (c = '0') then
    saida2 <= a;
  else
    saida2 <= b;
  endif;
end process exemplo1;
```

No exemplo acima, note que existem dois fluxos possíveis, criados pela construção IF. Seguindo o primeiro fluxo, temos:

```
saida1 <= a and b;
saida2 <= a;
```

E no segundo fluxo possível, temos:

```
saida1 <= a and b;
saida2 <= b;
```

Portanto, em todos os fluxos possíveis, meus sinais de saída são atribuídos a alguma expressão. Em termos de circuitos, garantimos que o sinal será sempre *driven* por alguém. Se você não fizer isso, a interpretação do sintetizador será que, para algum fluxo possível (ou seja, para alguns valores de entrada), existe algum sinal de saída que não recebe valor. Logo, será necessário armazenar o valor que este sinal recebeu em algum fluxo passado. Se o circuito foi combinacional, isto leva à famosa (e indesejada) *inferência de latches* (o latch é usado para armazenar valor).

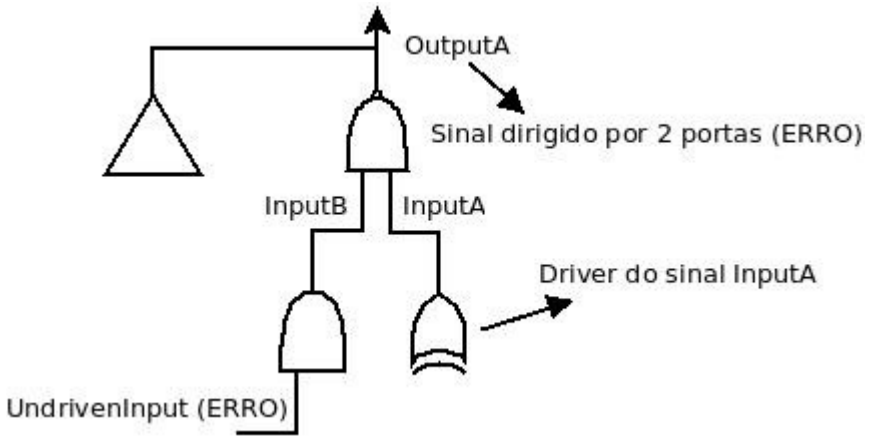
Relembrando o comportamento dos sinais: Durante a execução de um *process*, o valor de um sinal *nunca* muda, independente de suas atribuições. Uma atribuição ao sinal apenas irá escalonar uma mudança de valor para um momento futuro (atraso *delta*). Portanto, não faz sentido tentar ler o valor de um sinal como se fosse uma variável, na esperança de que ele esteja atualizado. Exemplo:

```
ERRADO: process (a, b, c)
begin
  if c = '0' then
    temp <= a and b;
  else
    temp <= a or b;
  endif;
  temp <= not temp;      -- Erro: temp não é uma variável. Isto irá
                         -- inferir um laço combinacional (ligar a
                         -- saída na entrada com uma porta NOT).
end process ERRADO;
```

Outro erro comum é atribuir duas vezes a um sinal no *mesmo* fluxo de execução, novamente como se fosse uma variável. O efeito disso é criar dois drivers para o mesmo sinal, o que é proibido para a síntese de circuitos combinacionais.

```
ERRADO: process (a, b, c)
begin
  if c = '0' then
    temp <= a and b;
  else
    temp <= a or b;
  endif;
  if c = '0' then
    temp <= '1';  -- segunda atribuição de TEMP no fluxo de execução
                  -- para C = '0'
  endif;
end process ERRADO;
```

A figura abaixo mostra o que isto representa no circuito. Percebemos que o nó/sinal OutputA é dirigido (ou *driven*) por duas portas, o que é errado para expressar um circuito combinacional.



DICA: Mantenha seus process' simples e modulares. Por exemplo, ao invés de criar uma lógica para definir o sinalA, e depois em seguida criar outra, no mesmo process, para o sinalB, crie dois process distintos, um para definir o sinalA e outro para o sinalB. Desse modo fica mais fácil de entender sua descrição.

DICA: Repare que usar process para descrever lógica de hardware dificilmente pode ser visto como um algoritmo sequencial, como aquele que ocorre quando se usa variáveis e descrito no início desse documento. Você deve enxergar os process' como uma maneira de atribuir sinais condicionalmente (criando Ifs e Cases), e, se for construir um algoritmo, use variáveis internas ao process e tenha certeza de que o algoirtmo é simples o suficiente para ser possível inferir hardware pelo sintetizador (por exemplo, usar variáveis para fatorar uma expressão é factível de ser usado e melhora a leitura do código).

Circuitos Sequenciais

Em termos de síntese, a diferença na construção do circuito sequencial é que:

- Sempre é iniciado com um IF testando se estamos na subida(descida) do clock. Se for adicionar mais atribuições após esse IF, tome cuidado e pense se não é melhor colocar esta lógica em outro *process* separado (afinal, uma vez fora do IF, pode ser vista como lógica combinacional). O *process* que infere a lógica sequencial deve preferencialmente conter apenas este IF e a lógica interna que sucede no caso de estarmos na subida(descida) do clock.

- Diferentemente do circuito combinacional, é possível ler o valor de sinais que são atribuídos neste mesmo process (sinais de saída), dentro do IF. Note também que, se o sinal é atribuído dentro do IF, o sintetizador irá necessariamente inferir um flipflop para este sinal (pois seu valor só se altera quando a borda do clock é acionada). Lembre que em um circuito combinacional você não pode atribuir um sinal apenas em um dos fluxos de execução, mas, em contrapartida, para expressar o circuito sequencial você deve necessariamente expressar apenas um fluxo de execução: aquele que acontece quando a borda de subida(ou descida) é acionada. Tome cuidado, entretanto, ao lembrar que o valor de um sinal nunca muda durante a execução do process. Esta é a regra do *delta time*: Um sinal é alterado após um tempo delta, e não imediatamente. Isto implica em: *O sinal de saída lido (que infere um flipflop) terá sempre o valor recebido na borda de subida(descida) de clock passada*.

- A *sensitivity list* do process normalmente contém apenas o clock. Outros sinais possíveis são os chamados sinais assíncronos (reset, enable). Se eles existirem, eles devem ser colocados na sensitivity list e é necessário criar um IF para testar se o sinal assíncrono está ligado *antes* do IF tetando pela borda de subida (ou descida).

Exemplo:

```
sequencial: process (clk, reset)
begin
  if reset = '1' then -- reset active high
    q <= '0';
  elsif clk'event and clk = '1' then
    q <= not q;   -- sinal "q" irá inferir um flipflop com reset
                  -- assíncrono. A cada subida do clock, o sinal
                  -- é invertido em relação ao passado (ler "q"
                  -- significa ler seu valor anterior e atribuir
                  -- "q" significa calcular seu novo valor).
  endif;
end process sequencial;
```

Melhorando seu Código HDL

Além dos conceitos básicos de descrição VHDL que você precisa conhecer, costuma-se definir *Coding Styles ou Guidelines*. São nada mais do que diretrizes para que você escreva código robusto e não ambíguo. Geralmente, um fabricante de ferramenta de síntese (como a Altera, Xilinx, Cadence, etc.) costuma publicar sua própria lista de *Coding Style* para que você faça código que garantidamente a ferramenta irá entender e gerar o hardware desejado. Entretanto, para um iniciante em HDL, independentemente da ferramenta usada, é sempre benéfico ler esses guias porque, acima de tudo, eles dão bons exemplos de como expressar uma determinada lógica de maneira clara e concisa. Ainda, costuma-se explicar como criar alguns circuitos básicos com a HDL (seja VHDL ou Verilog), como flipflops, máquinas de estado, etc.

Neste momento, como um projetista profissional, com conceitos sólidos de circuitos lógicos mas iniciante em HDLs, sua tarefa é olhar os Coding Styles da Actel, uma outra empresa de FPGAs:

http://www.ic.unicamp.br/~cortes/mc613/arquivos/hdlcode_ug.pdf

O motivo é que este guia tem uma seção muito boa chamada “Technology Independent Coding Styles”. O que isto significa? Como disse, cada empresa costuma publicar seu *Coding Style*, para que sua ferramenta específica entenda muito bem suas construções. Entretanto, nós somos ambiciosos e estamos interessados em trabalhar genericamente, para que qualquer ferramenta (ou colega) nos entenda. *Technology Independent* significa que os conceitos desta seção são universais, independente de qual tecnologia você vai utilizar (seja uma FPGA “xyz” ou na criação de um circuito integrado customizado de verdade).

Repare que o guia se preocupa sempre em dar exemplos utilizando as duas HDLs mais utilizadas no mundo: VHDL e Verilog. Vale a pena também dar uma olhada no guia da Altera:

http://www.altera.com/literature/hb/qts/qts_qii51007.pdf

Neste guia, na página 53 há uma ótima seção detalhando como você expressa uma máquina de estados. Na página 58, um exemplo em VHDL é dado.

Após se tornar um expert em *Guidelines* para síntese de circuitos lógicos, você poderá checar todos os seus designs no próprio Quartus. Escolha no menu “Processing” a opção do verificador de design, e o seu circuito será analisado completamente. Caso alguma *guideline* esteja sendo ferida, um aviso será emitido. Se o seu circuito passar desta etapa sem avisos, você pode ter certeza da robustez de seu design e que o hardware implementado fará exatamente o que você escreveu.