Snake: Aspectos Funcionais e Estruturais do Projeto

M.S/W.J.C 19

1. O Jogo

O jogo se dá sobre um tabuleiro de dimensão quadrada MxM, portanto apresentando M^2 células (exemplo de 8x8 na Figura 1). Os objetos do jogo que aparecem na tela do computador são:

- A cobra (*snake*) que ocupa N≥1 células adjacentes.
- Seu alimento que ocupa uma única célula.

A cobra consiste de sua cabeça (a célula dianteira), seguida por uma sequência de células adjacentes, correspondendo ao seu corpo. A célula mais afastada da cabeça corresponde à cauda.

O objetivo do jogo é mover a cobra em direção à sua comida.

O jogo começa com uma cobra de tamanho 1 (só a cabeça) posicionada na célula (0,0), no canto superior esquerdo, e sua comida, posicionada numa célula (I,J) selecionada aleatoriamente.

O jogador deve mover a cobra em direção à sua comida utilizando as flechas NORTE, SUL, LESTE, OESTE. Quando a cabeça da cobra atinge a comida, a cobra aumenta de tamanho (1 célula), a comida é recolocada numa nova posição (escolhida aleatoriamente) e o jogo recomeça.

Desta forma o jogo evolui em etapas, cada etapa sendo caracterizada por uma cobra de tamanho crescente. O objetivo do jogo é atingir uma cobra de maior dimensão possível.

O jogo chega ao fim quando a cabeça da cobra encostar em qualquer parte de seu corpo ou se o jogador decidir encerrá-lo.

Observação: Apesar do jogo transcorrer sobre um tabuleiro com aparência plana, na realidade os extremos do tabuleiro se tocam tanto na direção X como na Y. Por exemplo, se a cobra atingir a extremidade da direita do tabuleiro (posição (7,Y)), um movimento à direita faz com que a cobra reapareça na extremidade esquerda do tabuleiro (posição (0,Y)) e assim por diante.

2. A implementação do jogo

Este jogo pode ser desenvolvido tanto em software (para ser executado sobre um processador qualquer) ou em hardware.

Nesta disciplina o jogo será implementado todo em hardware a fim de ilustrar as diferentes etapas de um projeto de circuitos integrados semi-dedicados e da infraestrutura necessária para sua execução (ferramentas e bibliotecas de células e de macro-células).

O circuito final será designado pelo nome Circuito Snake e será prototipado sobre uma placa de desenvolvimento DE2 da Altera que contém um dispositivo FPGA da família Cyclone II. Recursos da placa como botões de pressão e saída para monitor no padrão VGA serão utilizadas para a interface com o usuário (veja a Figura 1).

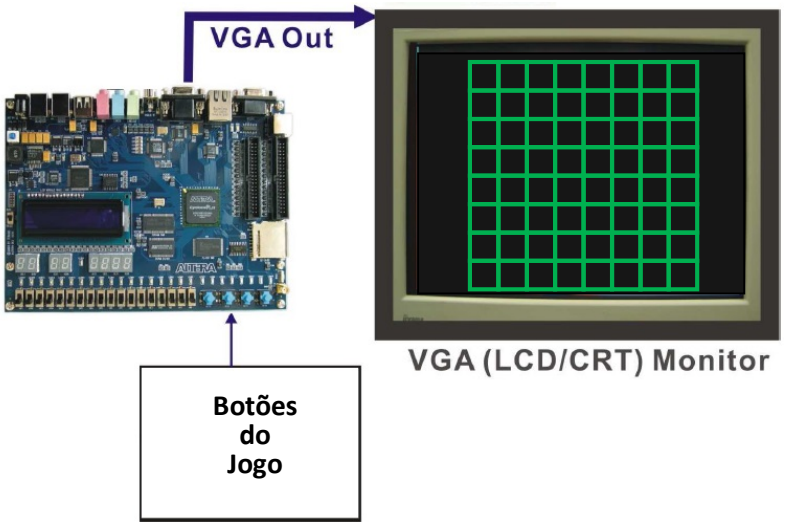

Figura 1: (Modificada da Figura original do Manual DE-2)

3. Modelo comportamental do circuito Snake em VHDL

A primeira tarefa de um projeto semi-dedicado consiste em criar um modelo comportamental em VHDL (ou outra linguagem de descrição de hardware). Este modelo é utilizado para duas finalidades: validar a funcionalidade do circuito a ser projetado e servir de ponto de partida para a etapa de síntese lógica.

O modelo costuma ser desenvolvido de forma modular e hierárquica.

Na seção 4 iremos descrever os modelos comportamentais criados para cada módulo nos quais o circuito foi decomposto, indicando também de que forma os diversos módulos se comunicam entre si durante a realização de um jogo.

4. A arquitetura do Circuito Snake

O circuito final será composto dos 4 módulos, como ilustrados na Figura 2: *Button Handler, Step Counter, Datapath* e *Control Snake*. Na Figura 2 também estão ilustrados alguns sinais que fazem parte da descrição da arquitetura em VHDL. Descrições detalhadas dos diversos blocos com cada um de seus portos serão providenciadas futuramente junto aos arquivos VHDL.

O módulo *Button Handler* executa a função de definir o movimento da cobra a partir dos botões existentes na placa.

O módulo *Step Counter* executa a função de definir a velocidade do movimento da cobra sobre o tabuleiro em direção à sua comida.

O módulo *Datapath* realiza todos os cálculos relativos ao movimento da cobra, ao seu tamanho (que cresce à medida que vai alcançando a comida) e ao reposicionamento do alimento da cobra a cada nova etapa do jogo.

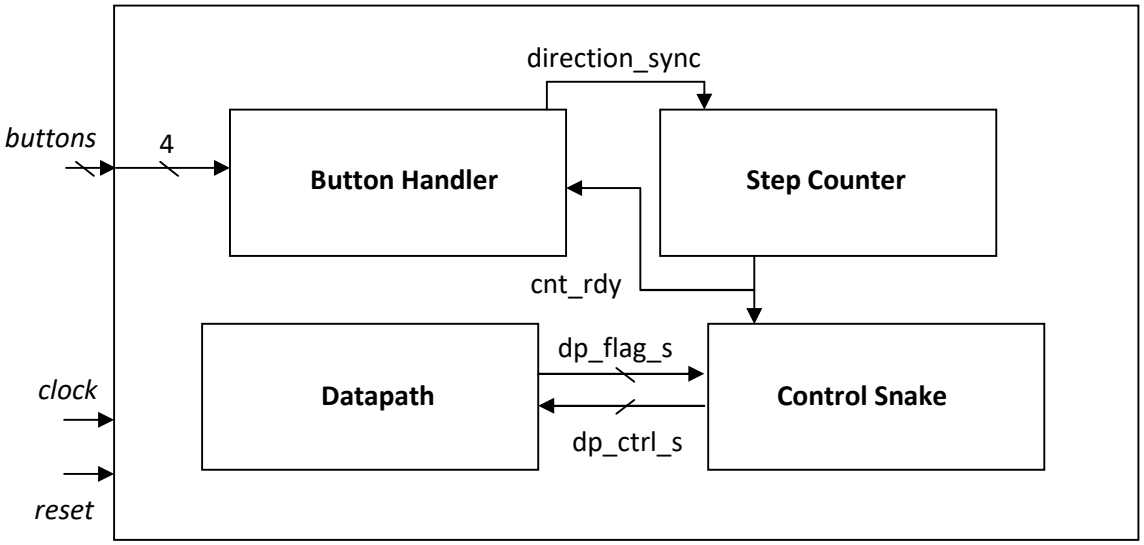

Figura 2: Arquitetura do Circuito Snake

O módulo *Control Snake* realiza o controle e sequenciamento das diferentes operações realizadas no *Datapath*, assim como das funções executadas nos dois outros módulos.

Todos os módulos tem os pinos de sinal de relógio (*clock*) e de (re)inicialização (*reset*).

Nas descrições a seguir serão feitas referências a alguns dos portos, sinais e parâmetros que estarão presentes na descrição funcional de cada módulo em VHDL. Os demais portos, sinais e parâmetros serão apresentados ao longo da realização do projeto.

4.1.Button Handler

A função deste módulo é definir a direção e sentido do movimento da cobra. O usuário tem um intervalo de tempo para decidir o próximo movimento através de botões (ver o sinal *buttons* da Figura 1). Este intervalo é definido pelo jogador (parâmetro de máxima contagem de *clocks* correspondendo ao nível de dificuldade do jogo). Se este intervalo de tempo for ultrapassado a cobra se moverá na mesma direção do movimento anterior. Se por outro lado o jogador ativar o botão de direção mais de uma vez durante o intervalo de tempo, a cobra se moverá na direção correspondente à última decisão tomada pelo usuário antes da sinalização de contagem máxima.

Há quatro botões de pressão sobre os quais o jogador atua, cada um indicando uma direção e sentido. Dependendo da opção escolhida pelo jogador, o sinal de saída *direction_sync* (para o *Control Snake*) poderá ter um dos quatro valores: UP, DOWN, RIGHT e LEFT.

Este módulo contém um registrador que armazena o último movimento da cobra escolhido pelo jogador. Este movimento permanecerá inalterado quando o jogador não

pressionar nenhum botão ou quando o jogador pressionar o botão com sentido oposto ao anterior (o jogo não permite que a cobra retorne em direção ao seu próprio corpo).

Em caso de *reset*, o jogo é reiniciado com *direction_sync* = RIGHT lembrando que a cabeça da cobra é sempre reposicionada na posição (0,0) no início de um novo jogo.

4.2.Step Counter

A função deste módulo é fazer uma contagem rotativa de 1 a COUNT_MAX, parâmetro inteiro definido pelo jogador. Em cada ciclo de contagem, uma indicação de finalização de contagem é colocada em forma de um pulso no sinal de saída *cnt_rdy*. Este sinal alimenta os módulos *Button Handler* e *Control Snake*. Quanto menor for o valor deste parâmetro, menos tempo o jogador terá para definir o próximo movimento, ou seja, o jogo se torna mais difícil.

4.3 Datapath

A Figura 3 apresenta a estrutura hierárquica deste módulo.

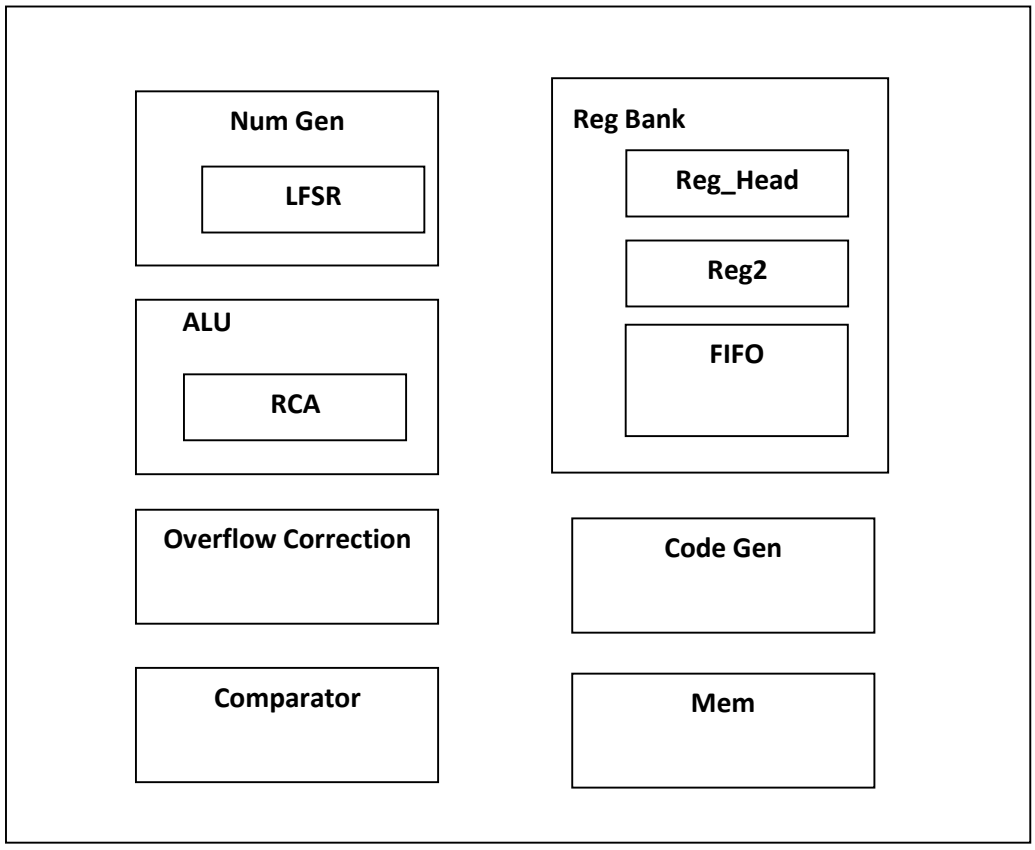

Figura 3: Estrutura hierárquica do módulo *Datapath*

A função deste módulo é realizar todos os cálculos para: iniciar os valores da memória (*reset*), efetuar o movimento da cobra, posicionar o alimento e comparar a posição da

cabeça da cobra com a de seu corpo e a do alimento. Os momentos da realização dos cálculos, assim como os recursos utilizados, são definidos por uma série de sinais oriundos do módulo Control Snake concentrados em um único sinal *dp_ctrl_s* de tipo *record* (ver na Figura 2). Da mesma forma, informações de overflow são enviadas para o módulo Control_Snake através do sinal (tipo record) *dp_flag_s* (ver na Figura 2).

Este módulo é composto de 8 sub-blocos, os quais serão detalhados a seguir.

4.3.1. Mem

Memória RAM estática de tamanho MxM (8x8 no projeto a ser desenvolvido), cada posição da memória correspondendo a uma posição do tabuleiro. Portanto, o barramento de endereço é de $2\log_2 M$ bits. Neste projeto serão 3 bits de endereço para X (coordenada horizontal de colunas) e 3 bits para Y (coordenada vertical de linhas). O tamanho dos dados é de 1 byte.

A memória armazena o status de cada uma das MxM posições do tabuleiro. São 7 status possíveis:

- BLANK (posição vazia)
- FOOD (posição da comida)
- BODY (posição ocupada pelo corpo da cobra, exceto sua cabeça)
- HEAD_UP (posição da cabeça se movimentando na direção UP)
- HEAD_DOWN (idem DOWN)
- HEAD_LEFT (idem LEFT)
- HEAD_RIGHT (idem RIGHT).

Os códigos correspondentes a cada status são gerados por Code Gen.

4.3.2. Code Gen

É um conversor de cada um dos 7 status acima listados (codificados como tipos enumerados) para seus correspondentes códigos em bytes (a serem armazenado em Mem). A conversão é a seguinte:

- BLANK 0000 0000
- FOOD 1000 0000
- BODY 0001 0000
- HEAD_UP 0001 0001
- HEAD_DOWN 0001 0010
- HEAD_LEFT 0001 0100
- HEAD_RIGHT 0001 1000

4.3.3. Num Gen

Trata-se de um gerador de números, os quais serão somados ao valor dos endereços de memória a fim de determinar a nova posição dos objetos sobre o tabuleiro após cada movimento. Estes números serão um dos operandos, *op-first* da ULA (ver seção da ALU). São 3 os possíveis valores, dependendo do objeto cuja posição está sendo recalculada e do estado em que o circuito se encontra: +1, -1, rand().

Os valores +1 ou -1 são utilizados em 2 casos. Para atribuir o valor BLANK aos respectivos endereços da memória quando esta é iniciada (usa-se o +1) e para a cobra avançar uma posição a cada passo (+1 quando o avanço é para baixo ou para a direita e -1 para cima ou para a esquerda).

O valor rand() é um número pseudo-aleatório gerado por um linear feedback shift register (LFSR) ou, em simulação, uma função que gera tal número. Estará entre 0 e (M^2-1) que define a nova posição do tabuleiro para onde o alimento será reposicionado quando este é comido pela cobra em lance anterior. Se esta posição atual estiver bloqueada, o cálculo de nova posição prossegue com nova geração de número aleatório a ser somado à posição atual (ver formatação dos endereços X e Y da memória em 4.3.5)

4.3.4. Reg Bank

Este bloco é composto de 2 registradores (REG_HEAD e REG_2) de 8 bits cada um e de uma FIFO (64x8) que servem para armazenar valores de endereços após cada movimento. Estes endereços são passados à ALU através de um mux de seleção de saída do Reg Bank..

a) REG_HEAD – este registrador armazena o valor do endereço para o qual a cabeça da cobra deve se mover, ou seja a próxima posição do tabuleiro que será ocupada pela cabeça da cobra, tal como definida pelo jogador. Este valor vem como resultado da ALU, corrigido no *Overflow Correction*, e a saída de REG_HEAD vai para o mux de seleção de saída do Reg Bank a ser somado ao +1 (avançando para a direita ou para baixo) ou -1 (para a esquerda ou para cima). A posição da cabeça também serve de referência para a geração da primeira posição do alimento. N primeira seleção de número aleatório, este será somado à posição da cabeça, que consiste da mesma posição do alimento que acabou de ser consumido (observar que esta nova posição é armazenada em reg_2 que passa a conter o valor a ser somado a outros números aleatórios em caso de conflito entre posição de alimento gerado e de cobra.

b) REG_2- este registrador tem dupla finalidade. A primeira é armazenar o valor do novo endereço do alimento cada vez que o jogo reinicia. Este endereço pode precisar ser recalculado algumas vezes caso o novo endereço corresponda a uma posição do tabuleiro bloqueada (por alguma parte do corpo da cobra), conforme explicado na descrição do bloco Num Gen. Este valor vem como resultado da ALU, corrigido no *Overflow Correction*, e a saída de REG_2 vai para o mux de seleção de saída do Reg Bank. A segunda finalidade de REG_2 é armazenar os endereços de memória na sua inicialização; neste caso, a soma do valor do valor armazenado será com +1, pois deve-se inicializar a memória, posição a posição.

c) FIFO- esta memória sequencial de 64 posições armazena, passo a passo, os endereços da memória (posições do tabuleiro) percorridos pela cobra ao longo de um jogo. Lembrando que o tamanho da cobra é variável, podendo ocupar (N ≥1) posições do tabuleiro. Existem 2 registradores especiais: o registrador *tail*

aponta para o endereço da FIFO (*first* - mais antigo) que armazena o endereço da cauda (na memória – Mem) e o registrador *head* aponta para o endereço da FIFO (*last* - mais recente) que armazena o endereço da cabeça (na memória - Mem). Durante o jogo, o tamanho da cobra permanece inalterado enquanto ela se movimenta até atingir seu alimento. O valor de *last* é atualizado com o novo endereço da cabeça e os ponteiros *tail* e *head* têm seus valores incrementados. No caso da cobra atingir o alimento, só o endereço da cabeça precisa ser atualizada (pois neste momento o tamanho da cobra aumenta). Vale observar que, no primeiro caso, o conteúdo inicial de FIFO (*first*) apontado pelo *tail* deve ser lido do Reg Bank e enviado a memória Mem para atualização de seu dado para BLANK (indicando que esta posição do tabuleiro ficou vazia, uma vez que a cobra andou).

Neste projeto, os registradores *tail*, *head* e a memória FIFO são todos de 8 bits servindo para armazenar os endereços de memória (Mem). Entretanto a memória necessita de apenas 6 bits de endereçamento (tabuleiro de 64 posições). Este aparente descompasso será explicado na seção de ALU.

4.3.5. ALU

O bloco ALU concentra todos os cálculos através de um somador de 8 bits (*ripple carry adder*, RCA), permitindo também a transferência de valores a partir do módulo REG BANK cujos valores não são alterados (equivale a somar com ZERO).

Inicialmente é importante entender o modelo de representação do endereço. Para simplificar o hardware a arquitetura contém apenas um somador (poderiam ser 2 somadores, um para calcular o endereço das colunas X e outro para o das linhas Y).
A fim de representar os endereços X e Y (3 bits cada um) em um único registrador de 8 bits, foi adotado o seguinte formato:

$$OV\_Y - Y_2 - Y_1 - Y_0 - OV\_X - X_2 - X_1 - X_0$$

Onde Yi corresponde aos 3 bits do endereço de linha (8 linhas) e Xi corresponde aos 3 bits do endereço de coluna (8 colunas). OV_y e OV_x correspondem ao overflow no cálculo do novo endereço através da soma caso o novo endereço ultrapasse os 3 bits.

4.3.5.1. Operandos Op_first

Os operandos Op_first, oriundos do bloco *Num Gen*, usados para calcular os novos endereços de Mem podem ser positivos ou negativos (representados em complemento de 2)(ver seção 4.3.3).

Na ULA, os operandos podem sofrer deslocamento de 4 bits para a esquerda no caso de a soma ser para Y (para cima ou para baixo no tabuleiro). No caso de

soma X (para esquerda ou direita no tabuleiro) e de rand(), o deslocamento não ocorre.
Os seus valores para recalcular os endereços X são:
+1 => 0000 0001
-1 => 1111 1111
Para recalcular os endereços Y, a ULA desloca os valores acima de 4 bits para a esquerda:
+1 => 0001 0000
-1 => 1111 0000

4.3.5.2. Operandos Rb_op

Os operandos Rb_op oriundos de Reg Bank são sempre números positivos (sendo $OV\__Y$ e $OV\__X$ também positivos; ver próxima seção).

Exemplo: Suponha que a cabeça está na posição (2,6) do tabuleiro (que corresponde ao endereço 010-110 de Mem) avança para a posição (2,7) (endereço 010-111). Na posição de origem o valor em REG_HEAD é 00100110. Para calcular a nova posição o valor de REG_HEAD será somado ao valor 0000 0001 (+1 de X como explicado acima). O resultado da soma será 00100111 que corresponde à posição 2,7 (que será armazenada em REG_HEAD).

O módulo ALU contém um MUX que permite uma opção de transferência direta do valor que vem de Reg Bank, sem passar pelo somador como seu operando Rb_op, quando o valor (endereço em Mem) que chega é o da cauda, oriundo da memória FIFO, para a operação de atualização da posição correspondente de Mem para BLANK (conferir com a descrição de 4.3.4c)

4.3.6. Overflow Correction

A fim de permitir que a cobra prossiga seu movimento após ter atingido alguma das extremidades do tabuleiro (da posição 8 para a posição ZERO tanto na direção X como na direção Y e vice-versa). Usa-se para isto uma operação módulo, fazendo o endereço retornar a 0 sempre que alcançar 8 (e vice-versa). Este bloco corrige os valores de $OV\__Y$ e $OV\__X$ (após uma soma na ALU) através de mascaramento, fazendo com que os bits correspondentes sejam sempre 0. O valor resultante será enviado ao Reg Bank . Desta forma, permite-se manter sempre os bits de overflow dos endereços em Reg Bank atualizados e consistentes com as diversas somas propostas na ULA (ver 4.3.5.2),

4.3.7. Comparador

Lembrando que cada jogo é interrompido quando a cabeça da cobra atinge a posição em que se encontra o alimento (caso em que a cobra aumenta de tamanho e o alimento é reposicionado antes do jogo ser reiniciado) e que o jogo é terminado quando a cabeça da cobra atinge uma posição ocupada por qualquer outra parte do seu corpo, o comparador serve para testar estas duas situações. Para verificar se há alimento na célula atingida pela cabeça basta verificar o bit 7

do código armazenado na correspondente posição de memória e para verificar se há alguma parte do corpo basta checar o bit 4 (ver Code Gen).

4.4 Control Snake

A Figura 4 apresenta a arquitetura deste módulo. A função deste módulo é sincronizar e sequenciar as ações que ocorrem durante o jogo. Este módulo é composto de 4 FSM´s concorrentes (máquinas de Moore):

1. FSM Main
2. FSM Init
3. FSM Food
4. FSM Step

As quatro máquinas estão ativas simultaneamente, porém a sincronização entre elas estabelece que, quando uma máquina está atuando sobre o restante do circuito, as outras estão em algum estado de espera. Os estados de espera podem ser as seguintes em cada máquina.

1. FSM Main : INIT_ACTIVATION; FOOD_ACTIVATION e STEP_ ACTIVATION
2. FSM Init : READY
3. FSM Food : READY
4. FSM Step : READY

Os estados iniciais de cada FSM (após um *reset*) estão representados por círculos de fundo cinza na Figura 4.

A seguir vamos descrever a função de cada FSM.

4.4.1. FSM Main

Esta máquina de estados é responsável pela organização geral das ações do jogo, sendo a coordenadora das outras três máquinas. A sequência de eventos de FSM Main é a seguinte:

1) Ao receber o sinal *reset*, a FSM é colocada no SEU ESTADO INICIAL INIT_ACTIVATION. O sinal fsm_i_start é ativado (0 ->1) ativando assim a FSM Init (que estava em READY). As duas outras FSM permanecem em READY enquanto o processo de iniciação estiver em curso.

2) A FSM Main permanece no estado INIT_ACTIVATION até que receba fsm_i_done=1 (indicando que FSM Init completou a iniciação da memória). Imediatamente esta muda para o estado FOOD_ACTIVATION.

3) No estado FOOD_ ACTIVATION o sinal fsm_f_start é ativado (=1) ativando a FSM Food (que estava em READY). As duas outras FSM permanecem em READY enquanto o processo FOOD estiver em curso.

4) A FSM Main permanece no estado FOOD_ ACTIVATION enquanto a FSM_Food estiver ativa. Quando o alimento atingir alguma célula livre do tabuleiro, o sinal fsm_f_done é ativado (=1) e então a FSM Main muda para o estado IDLE.

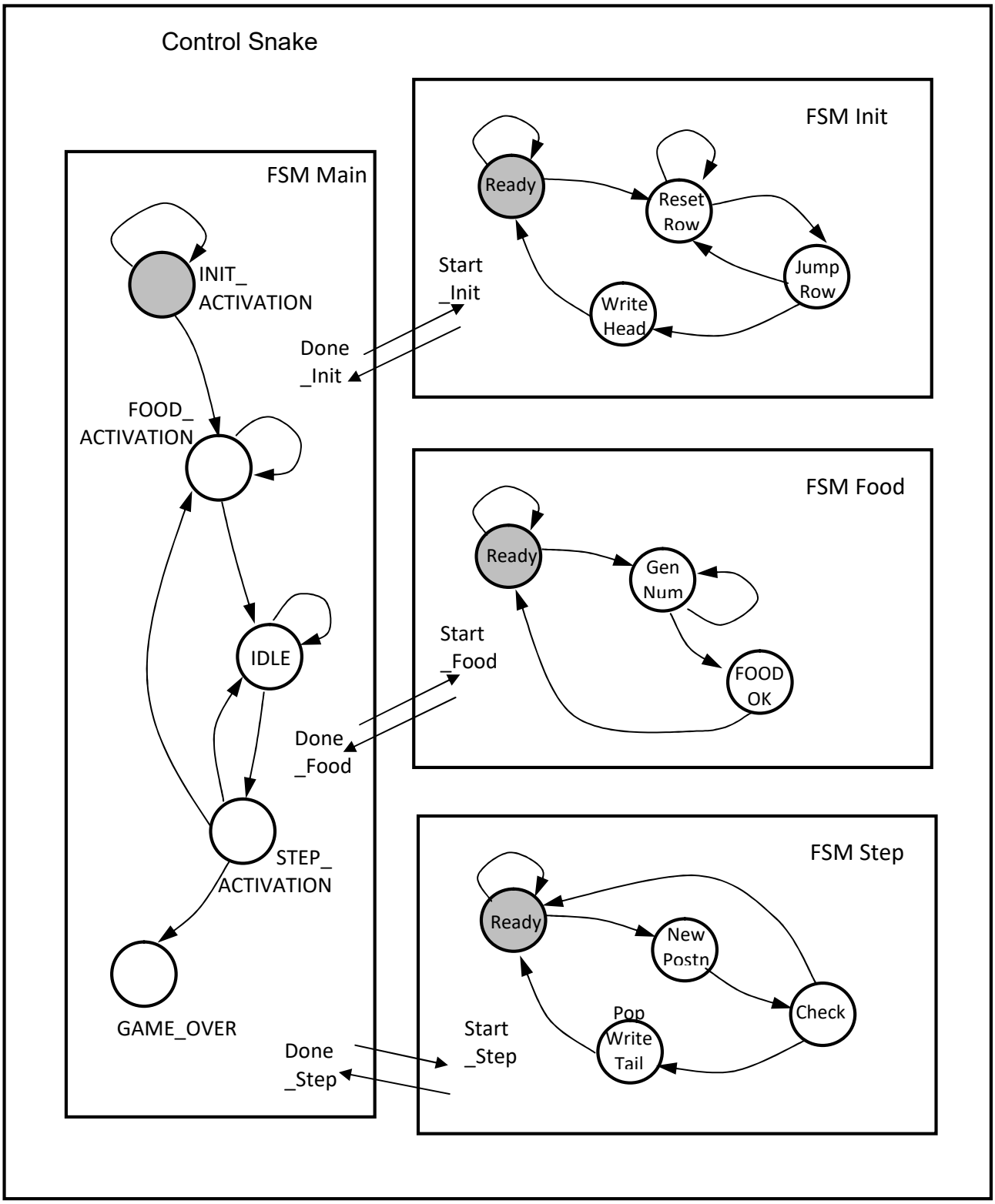

Figura 4: Arquitetura do módulo *Control Snake*.

5) Durante o estado IDLE, um contador é ativado. Durante o tempo de contagem o jogador deve decidir a direção do próximo movimento da cobra. Ao atingir o valor COUNT_MAX, estabelecido em Step Counter (seção 4.2), a FSM Main muda para o estado STEP_ACTIVATION.

6) No estado STEP_ ACTIVATION o sinal fsm_s_start é ativado (=1) ativando a FSM Step (que estava em READY). As duas outras FSM permanecem em READY enquanto o processo de movimentação da cobra estiver em curso.

7) A FSM Main permanece no estado STEP_ ACTIVATION enquanto a FSM Step estiver ativa. Quando a cobra avançou para a próxima posição definida pelo jogador, o sinal fsm_s_done é ativado (=1) e neste momento podem ocorrer um dos seguintes três casos: a) se o sinal gerado pelo comparador indicar que a cabeça encontrou o corpo, a FSM Main muda para o estado de GAME_OVER; b) se o sinal gerado pelo comparador indicar que a cabeça encontrou alimento, a FSM Main muda para o estado FOOD_ACTIVATION; c) se os casos a) ou b) não ocorrerem, isto significa que a nova posição da cobra é válida e a FSM Main muda para o estado IDLE.

8) Repetem-se as operações 5) 6) e 7).

4.4.2. FSM Init

FSM Init é a máquina responsável pela condição inicial da memória e o posicionamento inicial do alimento. A sequência de eventos de FSM Init é a seguinte:

1) O sinal *reset* coloca esta FSM no estado READY. A receber o sinal de ativação fsm_i_start=1, a FSM Init muda para o estado RESET_ROW;

2) No estado RESET_ROW ocorre um laço (loop) de 1 até M (número de células numa fileira do tabuleiro). Durante cada ciclo do loop ocorrem os seguintes eventos. O valor presente na saída de REG2/Reg Bank (iniciado em 0000 0000) é enviado ao porto de endereço da memória a fim de armazenar um BLANK naquela posição. Ao mesmo tempo este valor é enviado à ALU onde é incrementado (+1). O novo valor é armazenado em REG2/Reg Bank e o ciclo se repete até que todas as células da linha estejam preenchidas com o valor BLANK.

3) Ao atingir o final daquela fileira, o sinal ofc_of_x é ativado (=1) (seção 4.3.5) e a FSM muda para o estado JUMP_ROW.

4) Em JUMP_ROW existem duas opções. Se ofc_of_y=0 (seção 4.3.5) a FSM retorna ao estado RESET_ROW para escrever BLANK na nova linha. Caso contrário, o final do frame foi atingido e a máquina muda para o estado WRITE_HEAD.

5) Em WRITE_HEAD, o valor de endereço de memória (0000 0000) é armazenado no registrador REG_HEAD em Reg Bank e o valor HEAD_RIGHT é escrito nesta posição da memória (a cabeça da cobra começa na posição (0,0) e sua primeira direção é para a direita). A FIFO recebe o endereço 0000 0000 para o ponteiro HEAD (Last).

6) O sinal de saída fsm_i_done é ativado (=1) (indicando que a fase inicial foi completada) e a FSM Init retorna ao estado READY.

4.4.3. FSM Food

FSM Food é a máquina responsável pelo posicionamento do alimento no tabuleiro. Cada vez que o alimento deve ser reposicionado, esta FSM é responsável por buscar alguma célula que esteja desbloqueada (que não esteja ocupada pelo corpo da cobra). A sequência de eventos de FSM Food é a seguinte:

1) O sinal *reset* coloca esta FSM no estado READY. A FSM envia um sinal ao módulo Num Gen (ver seção 4.3.3) para que o valor aleatório da função rand() (seção 4.3.3) seja gerado e enviado à ALU onde seria somado a valor oriundo de REG_HEAD (em Rb_op) e retornado a REG2. No entanto, isto só ocorre quando os sinais con_sel e fsm_f_start são ativados no fsm_main, sendo que o primeiro seleciona o conjunto de sinalização para o *datapath*. Quando o sinal fsm_f_start é ativado (=1) o estado passa de READY para o estado GEN_NUM.

2) No estado GEN_NUM, a FSM continua enviando o sinal ao módulo Num Gen (ver seção 4.3.3) para que um novo valor aleatório da função rand() (seção 4.3.3) seja gerado e enviado à ALU onde é ***agora*** somado ao valor oriundo de REG_2 (em Rb_op), ou seja, o resultado da soma anterior. Ou seja, se for a primeira ocorrência de GEN_NUM, o valor em Rb_op deverá se o primeiro valor coletado de rand() somado à posição anterior do alimento. A valor da nova soma é enviado a REG2, enquanto o valor da soma anterior que foi utilizada como endereço para a obtenção do dado armazenado na memória, o qual é checado para tomada de decisão como no item abaixo.

3)Em GEN_NUM, existem duas opções: a) caso a posição de memória esteja desbloqueada (não existe cobra, ver comparador na seção 4.3.7), esta FSM muda para o estado FOOD_OK. b) Se a posição encontrada estiver bloqueada (contedo alguma parte da cobra), a FSM permanece no estado GEN_NUM e volta para o item 2).

4) No estado FOOD_OK, o valor FOOD é escrito na posição da memória definida antes da soma atualmente sendo executada, e a máquina retorna a READY. O sinal fsm_f_done é ativado (=1) indicando que o alimento encontra-se em sua nova posição.

4.4.4. FSM Step

FSM Step é a máquina responsável pela evolução da cobra a cada passo. A sequência de eventos de FSM Init é a seguinte:

1) O sinal *reset* coloca esta FSM no estado READY. Quando o sinal fsm_f_start é ativado (=1) esta FSM muda para o estado NEW_POSITION.

2) No estado NEW_POSITION, a FSM envia um sinal ao módulo Reg Bank para que o valor atual do endereço da cabeça (presente na saída de REG_HEAD em Reg Bank) seja selecionado e enviado à ALU. Ali é somado a +1 (movimento para a direita ou

para baixo) ou -1 (movimento para cima ou para a esquerda). A FIFO é atualizada com a nova posição em Last (HEAD pointer) e a FSM muda para o estado CHECK.

3) No estado CHECK é verificado o status da nova posição, ou seja o estado da ocupação desta célula. Há três casos: a) Se a nova posição estiver ocupada por alguma parte do corpo da cobra "BODY" (ver comparador da seção 4.3.7), o sinal fsm_s_game_over é ativado (=1), indicando a finalização de jogo. Esta FSM muda para o estado READY. b) Se a nova posição estiver ocupada pelo alimento, esta posição da memória é escrita com um dos valores HEAD_UP ou _DOWN ou _LEFT ou _RIGHT. O sinal fsm_s_done é ativado (=1) e a FSM retorna para o estado READY. c) Se a nova posição estiver vazia "BLANK" a FSM muda para o estado POP_WRITE_TAIL.

4) O estado POP_WRITE_TAIL corresponde à situação de avanço simples da cobra. A Fifo é atualizada no seu Tail (lembrar que Head já foi atualizado no item 2). O ponteiro de Tail (First futuro) é avançado com a retirada de dado (endereço do Tail) de First atual. Este endereço é utilizado para atualizar a memória com BLANK (antes era BODY). O sinal fsm_s_done é ativado (=1) e a máquina retorna para o estado READY.