TÉCNICO LISBOA

Parte 1 - Gauss Jacobi
Co-Projecto Hw/Sw

João Ramiro - 81138 José Carlos Vieira - 90900

Prof. Horácio Vieira Sequeira

---

1 Introdução

O projeto escolhido, dentro dos propostos, foi a implementação de um programa cujo seu objetivo é encontrar iterativamente uma solução de um sistema (diagonal dominante) de equações lineares, utilizando o método Gauss-Jacobi. Nesta primeira fase, o sistema pode ter até 8 equações lineares, onde tanto os elementos da matriz, como dos vetores são inteiros. O sistema, nesta fase, apenas calcula uma iteração do método, não tendo assim uma condição de paragem associada tal como seria de esperar.

O objetivo desta primeira fase incide na familiarização da utilização do *hardware* conjuntamente com o *software*, como tal, deverá ser verificado e entendido o funcionamento do sistema implementado. Este sistema deve gerir, ler e escrever para a memória que guardará inicialmente os elementos input e no fim o resultado da primeira iteração. Será utilizado o multiplicador estudado durante as aulas teóricas e um FIFO *stream* para a passagem dos dados entre o *processing system* (PS) e a *programmable logic* (PL). O componente HW (multiplicador) processará apenas os elementos inteiros, por isso a divisão (operação de *floating point*) é posteriormente implementada em software.

2 Algoritmo

Na álgebra linear numérica, o método Gauss-Jacobi (ou método iterativo de Jacobi) é um algoritmo para determinar as soluções de um sistema diagonalmente dominante de equações lineares $Ax = b$. O processo é iterado até convergir, ou seja, até Ax ser próximo o suficiente de b. Este algoritmo é uma versão simplificada do *Jacobi eigenvalue algorithm* [1]. O método tem o nome de Carl Gustav Jacob Jacobi.

> "Técnicas iterativas são raramente utilizadas para solucionar sistemas lineares de pequenas dimensões, já que o tempo requerido para obter um mínimo de precisão ultrapassa o requerido pelas técnicas diretas como a eliminação gaussiana. Contudo, para sistemas grandes, com grande porcentagem de entradas de zero (sistemas esparsos), essas técnicas aparecem como alternativas mais eficientes. Sistemas esparsos de grande porte frequentemente surgem na análise de circuitos, na solução numérica de problemas de valor de limite e equações diferenciais parciais."[2]

Como referido, é considerado o sistema de equações: $Ax = b$, onde:

$$A = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix}, \quad x = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}, \quad b = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}$$

Então A pode ser decomposto num componente diagonal D e o resto R:

$$A = D + R \quad Onde \quad D = \begin{pmatrix} a_{11} & 0 & \cdots & 0 \\ 0 & a_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_{nn} \end{pmatrix}, \quad R = \begin{pmatrix} 0 & a_{12} & \cdots & a_{1n} \\ a_{21} & 0 & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & 0 \end{pmatrix}$$

O sistema de equações lineares pode ser reescrito como:

$$Dx = b - Rx \tag{1}$$

O método de Jacobi é um método iterativo que resolve o membro esquerdo da expressão em ordem a x ao usar o método resultante da iteração anterior no membro direito. Analiticamente, isto pode ser escrito como:

$$x^{(k+1)} = D^{-1}(b - Rx^{(k)}) \tag{2}$$

ou, equivalentemente:

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j \neq i} a_{ij} x_j^{(k)} \right) \quad , i = 1, 2, \ldots, n. \tag{3}$$

Este método apenas converge para uma solução se, e somente se, o raio espetral de $D^{-1}R$ for menor que 1, i.e.:

$$\rho(D^{-1}R) < 1 \tag{4}$$

Consequentemente, o método de Jacobi é convergente sempre que a matriz A for diagonal dominante.

3 *PS Only*

Como referido o primeiro passo é gerar e guardar em memória as matrizes necessárias para a execução do método. Tendo em conta o referido na secção 2, n pode tomar valores menores ou igual a 8 e a matriz A tem de ser diagonal dominante. Para este passo as matrizes são guardadas num ficheiro binário que posteriormente é utilizado como input para inicializar a memória.

Neste caso foi utilizado o endereço base 0x10000000, que corresponde à primeira posição de memória da *external memory* (DDR). Para endereçar cada um destes vetores/matrizes, basta ter em conta o endereço anterior (ou o endereço base no primeiro caso) e o tamanho de cada um dos dados anteriores. Caso $n = 8$, os endereços de memória, são os apresentados no código 1.

Tendo em conta a equação 3, o somatório da multiplicação (produto interno) entre a_{ij} e $x_j^{(k)}$ é apenas aplicado quando $j \neq i$, ou seja quando não se trata de um elemento da diagonal de A. De forma a não fazer esta verificação por *hardware*, foram colocados a 0 todos os valores da diagonal de A. Ao mesmo tempo, foi calculado o vetor correspondente ao inverso da diagonal de A via *software*, pois trata-se de um resultado *float* (ainda não implementado). Este vetor de 8 posições é guardado de igual forma na memória DDR. Para isso foi utilizado o código apresentado no código 2:

Assim, estão reunidas todas as condições para a implementação do método Gauss-Jacobi. Nesta fase, foi proposto aos alunos desenvolver apenas uma iteração do método na versão onde apenas é utilizada a parte de PS, para se poder comparar com a versão onde o PL é implementado. Como, para o cálculo de cada um dos elementos de x_i^{k+1} é necessária a multiplicação por $\frac{1}{a_{ii}}$, onde resultado é um float, então não é, nesta fase, possível realizar a segunda iteração. Desta forma não é utilizada nenhuma condição de paragem, tal como apresentado no código 3

```
#define MATSIZE     8
#define INT_SIZE    4
#define MATRIX_SIZE INT_SIZE*MATSIZE*MATSIZE
#define VECTOR_SIZE INT_SIZE*MATSIZE

#define A_START_ADD   0x10000000
#define B_START_ADD   (A_START_ADD + MATRIX_SIZE)
#define X_START_ADD   (B_START_ADD + VECTOR_SIZE)
#define A_D_INV_ADD   (X_START_ADD + VECTOR_SIZE)
#define RES_START_ADD (A_D_INV_ADD + VECTOR_SIZE)

memA       = (int *)(A_START_ADD);
memB       = (int *)(B_START_ADD);
memRes     = (int *)(RES_START_ADD);
memXi      = (int *)(X_START_ADD);
memX       = (float *)(X_START_ADD);
memA_D_Inv = (float *)(A_D_INV_ADD);
```

Código 1: Posições de memória de cada um dos dados.

```
for (i = 0; i < MATSIZE; i++){
        memA_D_Inv[i] = 1.0f/A(i,i);
        A(i,i) = 0;
}
```

Código 2: Inverso da diagonal de A.

```
for (i = 0; i < MATSIZE; i++){
        memRes[i] = 0;
        for (j = 0; j < MATSIZE; j++)
                memRes[i] += A(i,j)*memXi[j];
}

for (i = 0; i < MATSIZE; i++)
        memX[i] = (float)memA_D_Inv[i]*(memB[i]-memRes[i]);
```

Código 3: Gauss-Jacobi.

4 PS e PL

Na sequência do projeto desenvolvido na secção 3, neste projeto são implementadas as mesmas funcionalidades, com a diferença do produto interno entre A e o x (secção 2) ser calculado na PL. De forma a aplicar esta nova funcionalidade, foi necessária a introdução de alguns *Intellectual Propertys* (IPs). Como se pode observar pela figura 1, foi adicionado o IP Processor System Reset, AXI Interconnect, AXI-Stream FIFO e o my_axis_int_matp_cnt (disponibilizado na página da cadeira).

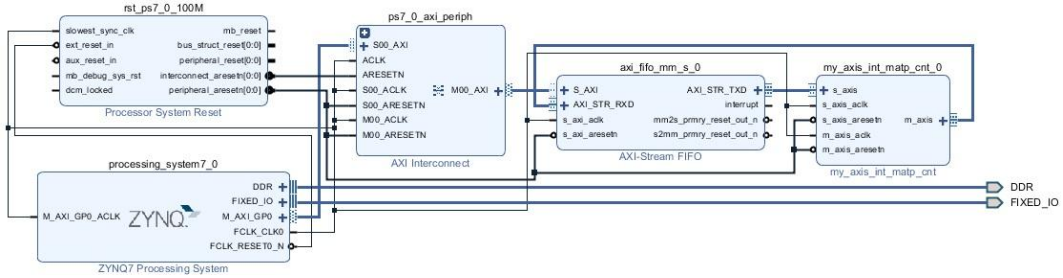

Figura 1: Diagrama PL.

Sendo necessário fazer uso da PL, é necessário ativar algumas funcionalidades na ZYNQ7 Processing System. Para que seja possível a transmissão de dados entre o PS e a PL é necessário ativar um porto AXI_M_GP (neste caso foi utilizado o 0). Para além disso, é necessário habilitar um dos *clock reset* (e.g., FCLK_RESET0_N) e ainda definir a frequência de relógio da PL, onde, neste caso foi habilitado o FCLK_CLK0 com uma frequência de 100MHz.

4.1 Multiplicador

Para o desenvolvimento do multiplicador, é necessário implementar o *datapath* que trata de toda a parte de registos, aritmética e operações lógicas, e a unidade de controlo (*Finite State Machine*) que faz uso dos *status* de saída do *datapath* para a escolha do estado atual e consequente controlo do *datapath*.

4.1.1 Datapath

O *datapath* do multiplicador implementado, é apresentado na figura 2. O seu objetivo é realizar o produto interno entre *data_A* e *data_B* e apresentar o resultado no *data_out*. Aplicando ao referido na secção 2, *data_A* será a_i e *data_B* será x_j.

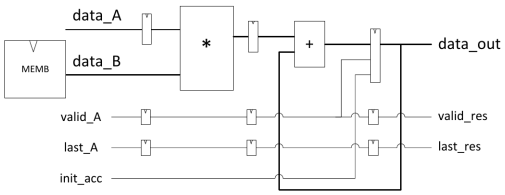

Figura 2: *Datapath* do multiplicador.

No produto interno é necessário realizar (em cada elemento) uma multiplicação e uma soma (acumulação). Para além destes, são também introduzidos os registos intermédios para reduzir o *critical path* e suportar *pipelining*. A sua utilização permite o aumento do *throughput*, isto é, resultados por unidade de tempo, apesar da sua implementação causar um atraso de um ciclo de relógio por cada registo introduzido. Isto permite que a frequência se mantenha alta, permitindo por isso, uma aceleração do *hardware*, e consequentemente aumento do *throughput*.

No *path* dos sinais *valid_A* e *last_A* é necessário implementar tantos registos intermédios quanto os que existem no *path* dos dados. Como o *valid_A* serve para o controlo do registo de saída (a montante do somador), os *paths* têm de ser síncronos. Os registos intermédios dos *paths valid_A* e *last_A*, implementam um *shift-register* e concatenação do novo valor de entrada, a cada flanco ascendente do relógio. Ao mesmo tempo, o valor de saída do multiplicador e o *data_A* são escritos nas respetivas memórias a montante. O valor de *data_B* está sempre disponível, pois este está escrito na memória MEMB.

O primeiro passo para realizar o produto interno é fazer o *reset* à memória de saída do somador, pois, esta é utilizada na sua entrada e no princípio tem guardado um valor indefinido. Para isso é utilizado o sinal *init_acc* que é verificado a cada flanco ascendente do *clock*, que quando for 1, é feita a escrita de 0s.

Na saída do multiplicador estará sempre o resultado da multiplicação entre *data_A* e *data_B* e na saída do somador estará sempre a soma entre a memória de saída e a memória a montante do multiplicador, consequentemente é necessário determinar quando é efetuada a escrita do resultado na memória de saída. Para isso é utilizado o valor presente no segundo registo do *valid_A*. Em cada flanco ascendente do relógio é verificado o seu valor, e caso este seja 1, dá-se a escrita.

4.1.2 *Control unit*

De modo a que o multiplicador funcione corretamente, é necessário fazer uso de uma unidade de controlo que gere sinais de controlo e defina quando é que os dados devem ser enviados. Esta unidade corresponde à *Finite State Machine* descrita nas aulas teóricas. Esta máquina de estados tem portanto 3 estados: *st_read_A*, *st_read_B*, e *st_write* tal como se pode observar na figura 3.

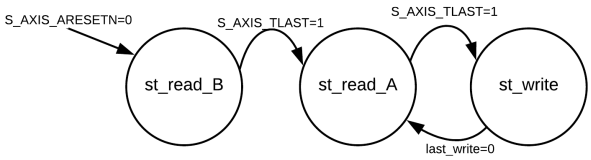

Figura 3: FSM do multiplicador.

A cada flanco ascendente do *S_AXIS_ACLK* é verificado o valor de *S_AXIS_TLAST*. Caso este tenha o nível lógico baixo, é colocada a máquina de estados no *st_read_B* onde, enquanto *S_AXIS_TVALID*=1 e *S_AXIS_TLAST*=0, todos os elementos recebidos (elementos de X), são escritos numa BRAM. Após a receção do sinal *S_AXIS_TLAST*=1, que indica que todos os elementos de X foram recebidos e escritos na memória, é passado para o estado st_read_A, onde são lidos os elementos de uma linha de A, sendo que a cada elemento lido, é feita multiplicação e a acumulação no *datapath*. Quando toda a linha é transmitida, passa-se ao estado de *st_write* onde o valor guardado no acumulador pode ser lido no *M_AXIS_TDATA*. Enquanto houver mais linhas de A, volta-se de novo ao estado *st_read_A* voltando se a fazer o produto interno dessa linha com X e posteriormente lido o resultado. O fim do produto interno das duas matrizes é dado pelo sinal *S_AXIS_TLAST*=1.

4.2 AXI-*Stream* FIFO

Foi introduzido o IP AXI-Stream FIFO de forma a tornar a comunicação entre o *software* e *hardware* possível. Todas as suas definições foram mantidas por *default* (512 palavras), pois é mais que suficiente para converter de *Axi-stream* para *Axi-lite* e vice versa sem que este fique totalmente ocupado.

4.3 Verificação do *Hardware*

Após a interligação de todos os IPs, foram executados os *Run Synthesis*, *Run Implementation* e o *Generate Bitstream*. De forma a verificar que a frequência de relógio implementada, poderia efetivamente ser utilizada, foi verificado o *Timing Summary Report*, onde foram obtidos os valores apresentados na tabela 1. Como se tratam de tempos positivos, esta frequência pode ser utilizada. Caso contrário teria de ser diminuída, pois, o WNS corresponde à margem entre o *slowest delay path* entre registos e o fim do ciclo de relógio.

Grandezas	Tempos[ns]
WNS	1.196
WHS	0.003
WPWS	3.750

Tabela 1: Timing Summary Report.

Foi também verificado o *Utilization Report*, onde foram obtidos os valores apresentados na tabela 2. Como se pode observar, e à semelhança do laboratório introdutório parte 2, são utilizados 3 DSPs, pois, como se trata de uma multiplicação de inteiros de 32-bit, então, estes são separados em dois números de 16-bit cada.

Considerando os inteiros $A = A_1 \times 2^{16} + A_0$ e $B = B_1 \times 2^{16} + B_0$, a sua multiplicação será:

$$A \times B = A_1 \times B_1 \times 2^{32} + A_1 \times B_0 \times 2^{16} + B_1 \times A_0 \times 2^{16} + A_0 \times B_0$$

O resultado é truncado, sendo igual aos 32-bit menos significativos, resultantes da multiplicação.

Grandezas	Quatntidade
Slice LUTs	1278
Slice Registers	1378
BRAMs	4
DSPs	3

Tabela 2: Utilization Report.

4.4 *Test bench*

Com o objetivo de confirmar o correto funcionamento do multiplicador (subsecção 4.1), foi alterado o *test-bench* fornecido, de forma a realizar as operações desejadas e verificar os resultados, figuras 4 e 5.

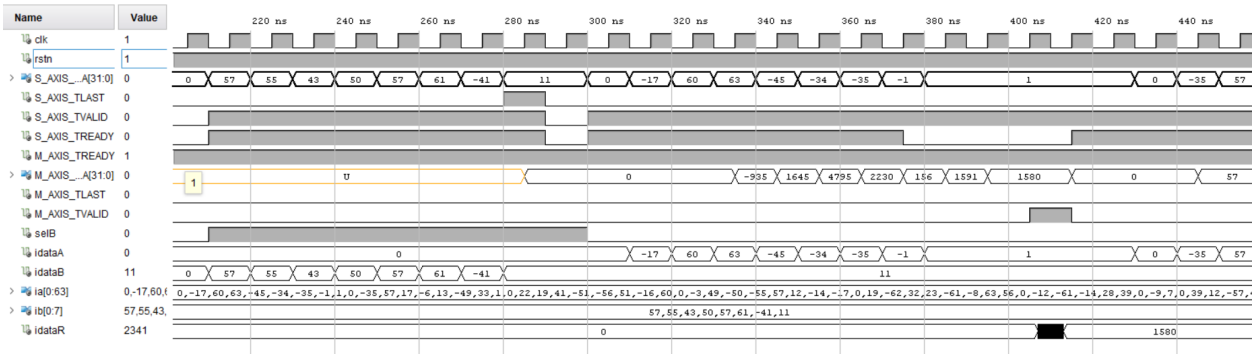

Figura 4: Cálculo de um valor.

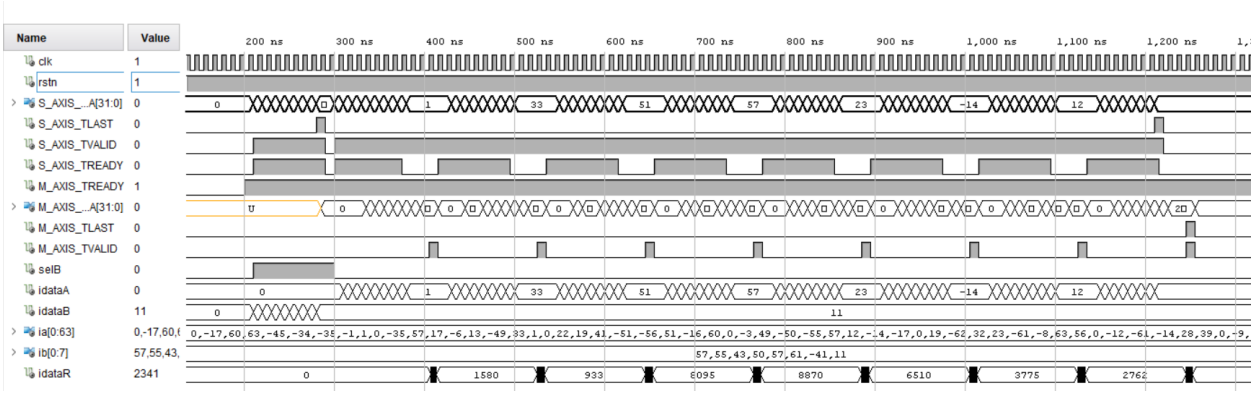

Figura 5: *Tesbench* Completo.

Pode ser observado, que inicialmente é enviando B (correspondente ao x_i) pois *selB* tem o valor lógico alto. De seguida é enviada cada linha de A separadamente, sendo que após a receção de cada elemento é feita a multiplicação e soma (linha *M_AXIS_TDATA*). No fim da receção de toda a coluna o resultado final é apresentado (*M_AXIS_TVALID* a 1). Este processo pode ser facilmente verificado na figura 4 onde, após a multiplicação entre $x_i = [57, 55, 43, 50, 57, 61, -41, 11]$ e a primeira linha de A $A(1,:) = [0, -17, 60, 63, -45, -34, -35, -1]$, é obtido 1580 e *M_AXIS_TVALID* a 1 aos 405ns. Para o produto com $A(n,:)$, x_i não será enviado novamente, pois este está guardado em memória.

4.5 *Software*

Após a verificação do bom funcionamento dos componentes implementados na PL, demonstrado na subsecção 4.4, foram realizadas as alterações necessárias para que o produto interno entre A e x, fosse calculado no *hardware*. Para

isso, o primeiro *for cycle*, apresentado no código 3, foi substituído pelo código 4.

```
my_axis_fifo_init();

my_send_to_fifo((void *)memXi, MATSIZE);

my_send_to_fifo((void *)memA, MATSIZE*MATSIZE);

my_receive_from_fifo((void *)memRes, MATSIZE);
```

Código 4: Utilização da FIFO *stream*.

À semelhança do apresentado no código 3, a após a receção do resultado (memRes), é calculado o resultado da primeira iteração (memX), onde este substitui os valores presentes na memória inicial (memXi). Note que memXi e memX apontam para a mesma zona de memória, embora com tipos de dados diferentes, int e float respetivamente (código 1).

5 Conclusão

Foram medidos os tempos, na *board*, de ambas as implementações, apresentados na tabela 3. Como se pode verificar, para apenas uma iteração e para n=8 elementos, o *PS e PL* é $\frac{26.56}{6.37} = 4.1$ vezes mais demorado que o *PS only*. Este

Implementação	Tempos[μs]
PS Only	6,37
PS e PL	25,56

Tabela 3: Tempos de computação.

atraso é consequência dos tempos de comunicação. Nas condições referidas, têm se $N^2 + N$ transferências sw-hw e N transferências hw-sw. Como n é muito pequeno, a aceleração que o *hardware* consegue oferecer em termos do calculo (produto interno), ainda não compensa em relação ao tempo das comunicações. Espera-se que quanto maior n, menos significativa seja esta diferença, pois o tempo de computação por *software* irá aumentar de forma mais rápida que o tempo despendido nas comunicações.

Além disso, de forma a conseguir *hardware acceleration*, na segunda parte do projeto, vão ser implementadas técnicas como DMA, guardar a matriz A em memória, pois esta nunca é alterada, e tirar partido dos dois processadores disponíveis na *board*, aplicando técnicas de paralelismo em *software*.

Concluí-se, portanto, que os objetivos incluindo a compreensão e implementação de um sistema básico de implementação em *Hardware* e *Software*, foram cumpridos.

References

[1] `https://en.wikipedia.org/wiki/Jacobi_eigenvalue_algorithm`

[2] `https://pt.wikipedia.org/wiki/Metodo_de_Jacobi`