Fonction buildAddress

La fonction *buildAddress* est un membre de la classe NNLayer. Cette fonction a pour but transférer les données d’une couche à celle qui la succède en considérant les interconnexions entre celles-ci. Afin de réaliser cette tâche, cette dernière utilise les données suivantes:

int n\_input\_per\_neuron // Entrée. Nombre d'entrées par neurone

int n\_neuron // Entrée. Nombre de neurones

float\* source // Entrée. Valeurs stimuli

const int\* current\_pos // Entrée. Tableau des interconnexions

int\* LUT\_Address // Sortie. Transcription des valeurs stimuli des LUT en ordre

Ci-dessous est la fonction originale en C, composée prioritairement de deux boucles *for* imbriqués. La première boucle fait le décompte des neurones qui ont été traitées en utilisant la variable *i* et la deuxième fait le décompte des « dendrites » de ces neurones avec la variable *j*. Les valeurs stimuli d’un neurone sont obtenues à partir des données à l’adresse pointée par *current\_pos* dans le tableau *source.* Ces stimuli sont ensuite transmis aux LUTs à travers le tableau *LUT\_Address*.

void NNLayer::buildAddress(float\* source, const int\* current\_pos, int\* LUT\_Address) {

for (int i = 0; i < n\_neuron; i++) {

for (int j = 0; j < n\_input\_per\_neuron; j++) {

if (source[\*(current\_pos++)] != 0) LUT\_Address[i] += (1 << j);

}

}

}

Le tableau ci-après donne un exemple des valeurs qui pourraient se retrouver dans l’un des octets du tableau *LUT\_Address* au fur et à mesure que le processeur itère dans la boucle « *j*». Le nombre d’entrées d’un LUT dans cet exemple est de 6.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **i** | **j** | **current\_pos** | **\*current\_pos** | **source[\*current\_pos]** | **LUT\_Address[i]** |
| 5 | 0 | 124 | 123 | 1 | 0b000001 |
| 5 | 1 | 125 | 234 | 0 | 0b000001 |
| 5 | 2 | 126 | 345 | 0 | 0b000001 |
| 5 | 3 | 127 | 456 | 1 | 0b001001 |
| 5 | 4 | 128 | 567 | 1 | 0b011001 |
| 5 | 5 | 129 | 678 | 0 | 0b011001 |
| 6 | 0 | 130 | 789 | 1 | 0b000001 |
| … | … | … | … | … | … |

À partir de cette fonction a été créé un ASM. Cet ASM sans optimisations comprend au total 11 états.

void NNLayer::buildAddress\_hard(float\* source, const int\* current\_pos, int\* LUT\_Address) {

//Control signals

int start = 1;

int done;

int data, read, write, waitrequest = 0;

void\* address;

//Local variables

int i, j;

int current\_pos\_index = 0;

//Initialization

INIT: if (start == 0) { goto INIT; }

else { write = 0; read = 0; done = 0; i = 0; goto S1; }

//Check if all neurons are done

S1: if (i < n\_neuron) {j = 0; goto S2; }

else { done = 1; return; }

// Check if all neuron inputs are connected; Read data in current\_pos

S2: if (j < n\_input\_per\_neuron) { address = (void\*)(current\_pos + current\_pos\_index); read = 1; goto RDRQ1; }

else { i++; goto S1; }

//Read data in source + \*current\_pos

S3: if (1) {address = source + \*(current\_pos + current\_pos\_index); read = 1; goto RDRQ2; }

//Check if input needs to be connected.

S4: if (source[current\_pos[current\_pos\_index]] != 0) { address = LUT\_Address + i; read = 1; goto RDRQ3; }

else { j++; current\_pos\_index++; goto S2; }

//Read data in LUT\_Address + i

S5: if (1) { address = LUT\_Address + i; data = \*(LUT\_Address + i) + (1 << j); write = 1; goto WRRQ; }

// Wait for memory

/\*Read \*current\_pos from data\*/

RDRQ1: if (waitrequest == 1) { goto RDRQ1; }

else { data = current\_pos[current\_pos\_index]; read = 0; goto S3; }

/\*Read \*(source + \*current\_pos) from data\*/

RDRQ2: if (waitrequest == 1) { goto RDRQ2; }

else { data = source[current\_pos[current\_pos\_index]]; read = 0; goto S4; }

/\*Read \*(LUT\_Address + i) from data\*/

RDRQ3: if (waitrequest == 1) { goto RDRQ3; }

else { data = \*(LUT\_Address + i); read = 0; goto S5; }

WRRQ: if (waitrequest == 1) { goto WRRQ; }

else { \*(LUT\_Address + i) += (1 << j); write = 0; j++; current\_pos\_index++; goto S2; }

}

Cet ASM a ensuite été optimisé. Le nombre d’états a diminué à 7 en exécutant les opérations à la mémoire une à la suite de l’autre plutôt que de transiter à travers un état intermédiaire pour changer les signaux de contrôle du bus de données. Les lectures et les écritures superflues dans le tableau LUT\_Address lors de la construction de l’octet à transcrire ont aussi été éliminées. Un registre à décalage est utilisé à la place.

void NNLayer::buildAddress\_hard\_optimise(float\* source, const int\* current\_pos, int\* LUT\_Address) {

//Control signals

int start = 1;

int done;

int data, read, write, waitrequest = 0;

void\* address;

//Local variables

int i, j;

int neuron\_stim = 0;

int current\_pos\_index = 0;

//Initialization

INIT: if (start == 0) { goto INIT; }

else { write = 0; read = 0; done = 0; i = 0; goto S1; }

//Check if all neurons are done

S1: if (i < n\_neuron) { neuron\_stim = 0; j = 0; goto S2; }

else { done = 1; return; }

// Check if all neuron inputs are connected; Read data in current\_pos

S2: if (j < n\_input\_per\_neuron) { address = (void\*)(current\_pos + current\_pos\_index); read = 1; goto RDRQ1; }

else { address = LUT\_Address + i; data = neuron\_stim; write = 1; goto WRRQ; }

//Check if input needs to be connected.

S4: if (source[current\_pos[current\_pos\_index]] != 0) { neuron\_stim += (1 << j);}

j++; current\_pos\_index++; goto S2;

// Wait for memory

/\*Read \*current\_pos from data\*/ //Read data in source + \*current\_pos

RDRQ1: if (waitrequest == 1) { goto RDRQ1; }

else { data = current\_pos[current\_pos\_index]; address = source + \*(current\_pos + current\_pos\_index); goto RDRQ2; }

/\*Read \*(source + \*current\_pos) from data\*/

RDRQ2: if (waitrequest == 1) { goto RDRQ2; }

else { data = source[current\_pos[current\_pos\_index]]; read = 0; goto S4; }

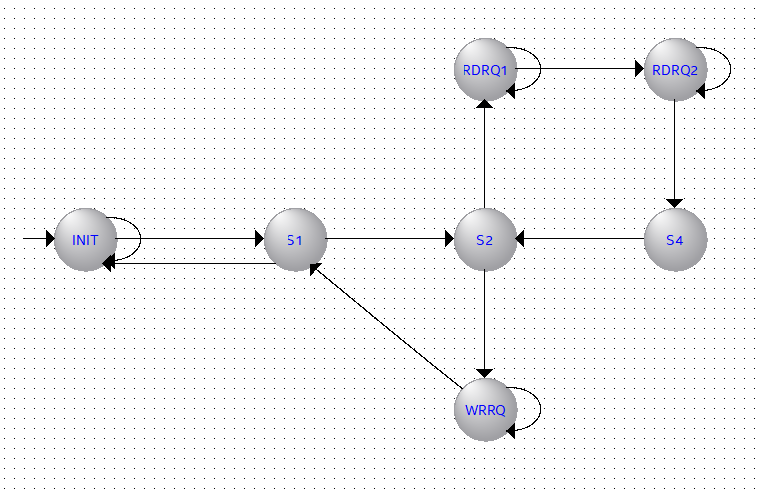
/\*Write LUT\_Address stimulus byte\*/

WRRQ: if (waitrequest == 1) { goto WRRQ; }

else { \*(LUT\_Address + i) = neuron\_stim; write = 0; i++; goto S1; }

}

Voici le diagramme d’état de cet ASM.



Les états S1, S2, et S4 servent à vérifier les différents branchements à réaliser selon l’état de la machine, tandis que les états RDRQX et WRRQ sont des états d’attente pour les lectures et les écritures à la mémoire. Afin de valider cet ASM, les données en sortie de cette dernière ont été comparées aux données originales grâce à un script Matlab.

L’implémentation de ce module dans le système FPGA s’est fait grâce à une interface de type « instruction spécialisé multicycle étendue ». Les paramètres de la fonction peuvent alors être changés en utilisant différents *opcodes*. Les paramètres associés aux différents opcodes sont décrit dans le tableau ci-dessous.

|  |  |  |
| --- | --- | --- |
| **Opcode (n)** | **dataa** | **datab** |
| 0 - 0b00 | n\_neuron | n\_input\_per\_neuron |
| 1 - 0b01 | source | LUT\_Address |
| 2 - 0b10 | current\_pos | - |
| 3 - 0b11 | - | - |

Normalement, lors de la réinitialisation du système, la machine est à l’état INIT. Cette dernière ne passe à l’état S1 que lorsque l’opcode 2 est envoyé. Afin de satisfaire les spécifications de l’interface, les signaux de contrôle *start* et *done* sont intégré dans l’algorithme du module, tel que visible dans le code ASM présenté antérieurement.

Pour compléter l’implémentation matérielle, une interface « Avalon Bus Master » est ajoutée. Cette interface permet de communiquer avec la mémoire et ainsi d’accéder aux données pertinentes et de construire le tableau *LUT\_Address*.

Le banc d’essai suivant a été utilisé afin de valider le bon fonctionnement de ce dernier. Un réseau contenant 4 neurones à 6 entrée est initialisé, donc 4 écritures et 24 lectures devraient être visible avant que le signal *done* soit levé. Les quelques premiers entrées sont branchés à un stimulus de 0 tandis et les reste est connecté à un stimulus de 1.

do\_check\_out\_result:process

begin

busy <= '0'; --Avalon bus never busy

clk\_en <= '1'; --Clock always enabled

rddata <= x"0000\_0000";

--Reset system

reset <= '1';

wait for 10 ns;

reset <= '0';

start <= '1'; --Custom instructions start signal asserted

--Write parameters

n <= "00";

dataa <= (2 = > '1', others = > '0'); --4 neurons

datab <= (2 downto 1 = > '1', others = > '0'); --6 inputs per neurons

wait for 40 ns;

n <= "01";

dataa <= x"0000\_00FF"; --Source address

datab <= x"0000\_FF00"; --LUT address

wait for 40 ns;

--Start work

n <= "10";

datab <= x"00FF\_0000"; --Currrent position address

wait for 40 ns;

start <= '1'; --Custom instructions start signal deasserted

start <= '0';

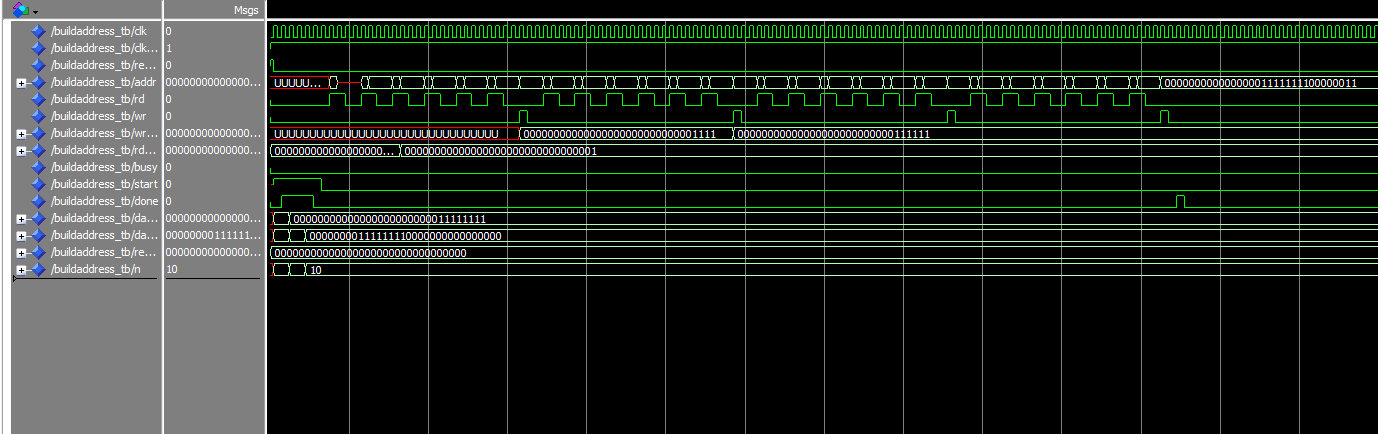
wait for 200 ns;

rddata <= x"0000\_0001";

wait for 2 us;

end process do\_check\_out\_result;

Ci-dessous sont les résultats de la simulation réalisée.



Dans les premiers instants après la réinitialisation, on peut voir l’injection des différents paramètres dans la machine à état. Après, il est facilement possible d’observer les 24 lectures, à *current\_pos* et à *source* respectivement, grâce au signal rd. Il en est de même pour les 4 écritures à *LUT\_Address*, grâce au signal *wr*. Tel qu’il est décrit dans le banc de test, la valeur de *rd\_data* change après les deux premières lectures dans *source*. Cela fait en sorte que les bits 4 et 5 dans la première écriture dans *LUT\_Address* sont à 0 plutôt qu’à 1 comme c’est le cas dans les écritures suivantes. Après que ces opérations soient complétées, le signal *done* est mis à 1 pour un cycle d’horloge.

Puisqu’il n’y a plus besoin de faire la lecture et l’écriture de *LUT\_Address* à chaque itération, le nombre de cycles nécessaires pour compléter ce processus diminue d’entre -1 à 11 opérations mémoire par neurone si les LUT ont 6 entrées. Le cas minimum survient lorsque tous les stimuli sont nuls, tandis que le cas maximum survient lorsque tous les stimuli sont actifs. Dans le programme original, il y a 24 opérations mémoire par neurones. Dans le meilleurs des cas, une amélioration de 45% peut être prévue, mais dans le pire des cas, ça sera un ralentissement de 4% qui sera observé. Une figure plus réaliste serait de prendre le facteur d’amélioration liée à une distribution uniforme de 0 et de 1 dans le tableau *source.* Dans ce cas, l’amélioration est d’environ 20%.