

Tutoriel : Prise en main du RISC-V CVA6



COMPAS 2024 —————
————— 2-5 juillet 2024, Nantes

Introduction

Juliette Pottier

Doctorante au laboratoire
IETR à Nantes


Antoine Bernabeu

Post-doctorant au laboratoire
IETR à Nantes

Plan



Objectifs du tutoriel

- 
- Prise en main d'outils pour le développement de cœurs RISC-V open-source
 - Optimisation du design décrit en SystemVerilog
 - Modification d'un compilateur pour l'ajout d'instructions customs
 - Implémentation sur carte FPGA

Contexte : le concours RISC-V

Edition 2024 : *Accélérateur matériel d'applications dédiées à l'IA pour un cœur de processeur RISC-V*

Application à accélérer : MNIST

Cœur cible : CV32A6



Contexte : l'application MNIST

→ Reconnaissance de chiffres de 0 à 9 écrits à la main

→ Base de données : 60 000 images

→ Les images sont traitées par un CNN (Convolutional Neural Network)



Contexte: le cœur CVA6

2017 ○ Création de **Ariane** par l'ETH Zürich et l'Université de Bologne

2020 ○ Première édition du concours RISC-V

2021 ○ Ariane devient **CVA6** et est transféré à l'OpenHW Group



THALES



Processeur **RISC-V** (extensions I, M, A, F, D, C & H)



6 étages de pipeline in-order (execute out-of-order)























Capable de booter **Linux** (Zephyr, FreeRTOS, ...)

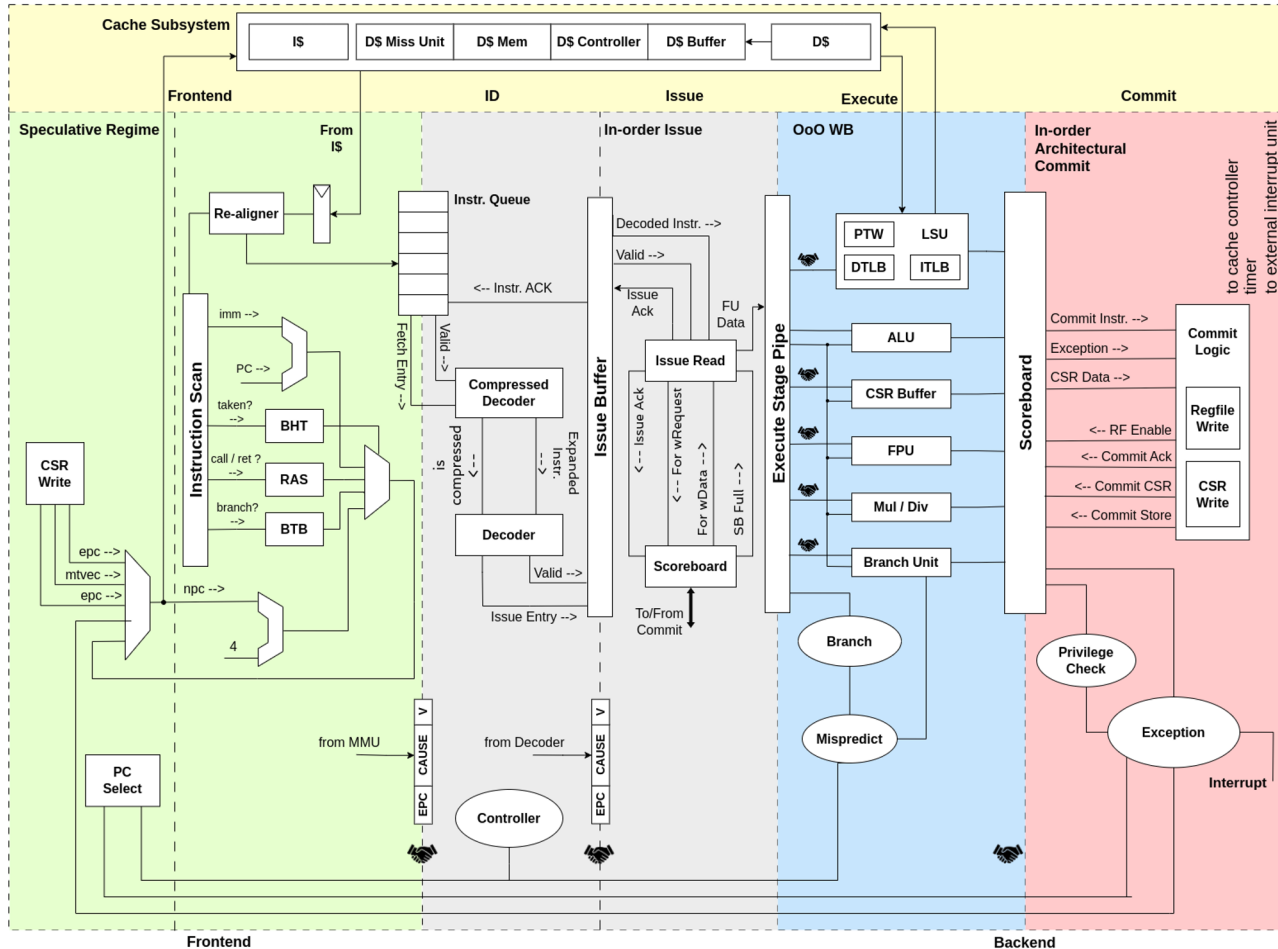
Prise en main!



La structure du projet

 cv32a6_contest_23_24	 5 Branches	 2 Tags	<input type="text" value="Go to file"/>	 Add file	 Code
 sjthales added reporting results doc  eab144e · 5 months ago  2,066 Commits					
 .github	updating cva6 sources and hierarchy				7 months ago
 .gitlab-ci	updating cva6 sources and hierarchy				7 months ago
 ci	updating cva6 sources and hierarchy				7 months ago
 common/local/util	updating cva6 sources and hierarchy				7 months ago
 core	update hpdcache to f2bc...				7 months ago
 corev_apu	update common_verification to a1e5...				7 months ago
 docs	docker image update				7 months ago
 pd/synth	updating cva6 sources and hierarchy				7 months ago
 sw	fix sw Makefile				7 months ago
 util	remove all jobs for toolchain compilation				7 months ago
 vendor	updating cva6 sources and hierarchy				7 months ago
 verif	restore submodules				7 months ago

Architecture de CVA6



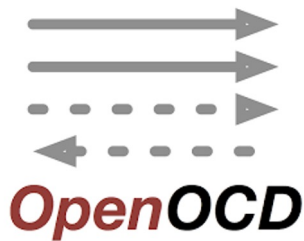
CV-X-IF

<https://github.com/openhwgroup/core-v-xif>

Les outils pour le concours

AMD
Vivado

- Synthèse
- Implémentation



- Debug (JTAG)



- Compilation (GCC)
- Debug (GDB)

QuestaSim

- Simulation



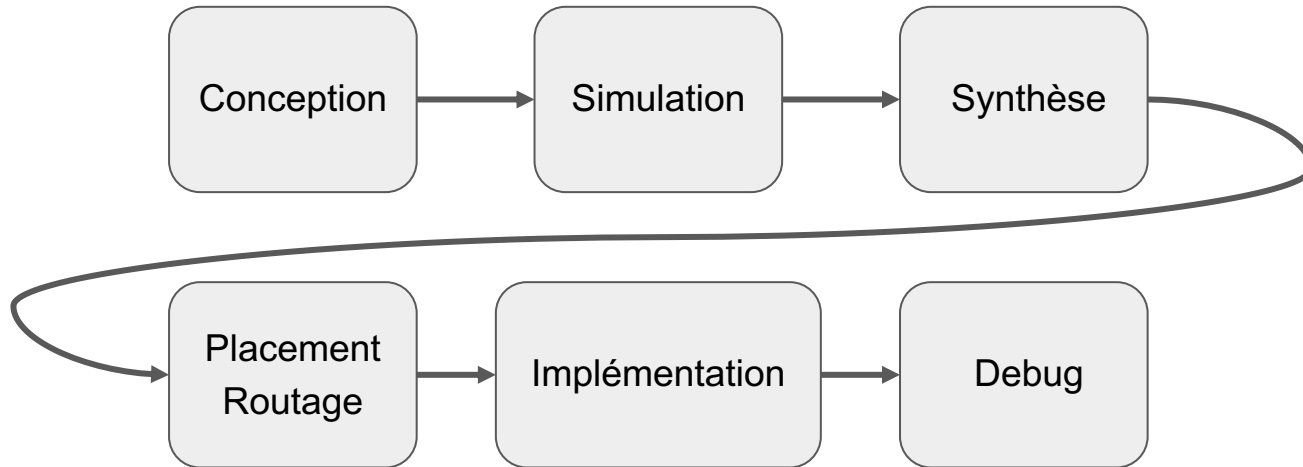
- Container pour le logiciel

Présentation Vivado

make fpga

À faire la première fois, pour les fois suivantes, utiliser l'interface graphique Vivado pour ne pas tout re-synthétiser.

Possibilité d'ajouter des sondes de debug qui permettent de voir les valeurs des signaux dans le FPGA : **ILA cores**



Prise en main du projet fourni par Thales (1/7)

Dépôt GitHub : <https://github.com/ThalesGroup/cva6-softcore-contest>

Contraintes logicielles :

- Le projet est prévu pour fonctionner avec Vivado/Vitis 2020.1
- Vivado 2020.1 nécessite Ubuntu 16.04.5, 16.04.6, 18.04.1, 18.04.2, 18.04.3, ou 18.04.4 LTS

Recommandations matérielles :

- La compilation du projet Vivado nécessite jusqu'à 10 Go de RAM (RAM + Swap) sinon la compilation échoue.
- Vivado exploite mal la compilation multithreads : il vaut mieux privilégier peu de cœurs CPU rapides que beaucoup de cœurs plus lents.

Prise en main du projet fourni par Thales (2/7)

Préparation et validation du projet (cf les étapes décrites dans le fichier [README.md](#)).

RQ: toutes ces étapes sont déjà réalisées dans la VM fournie

1. Récupération du projet

```
> git clone https://github.com/ThalesGroup/cva6-softcore-contest.git  
> cd cva6-softcore-contest  
> git submodule update --init --recursive
```

2. Compilation du projet Vivado (durée: environ 30 minutes)

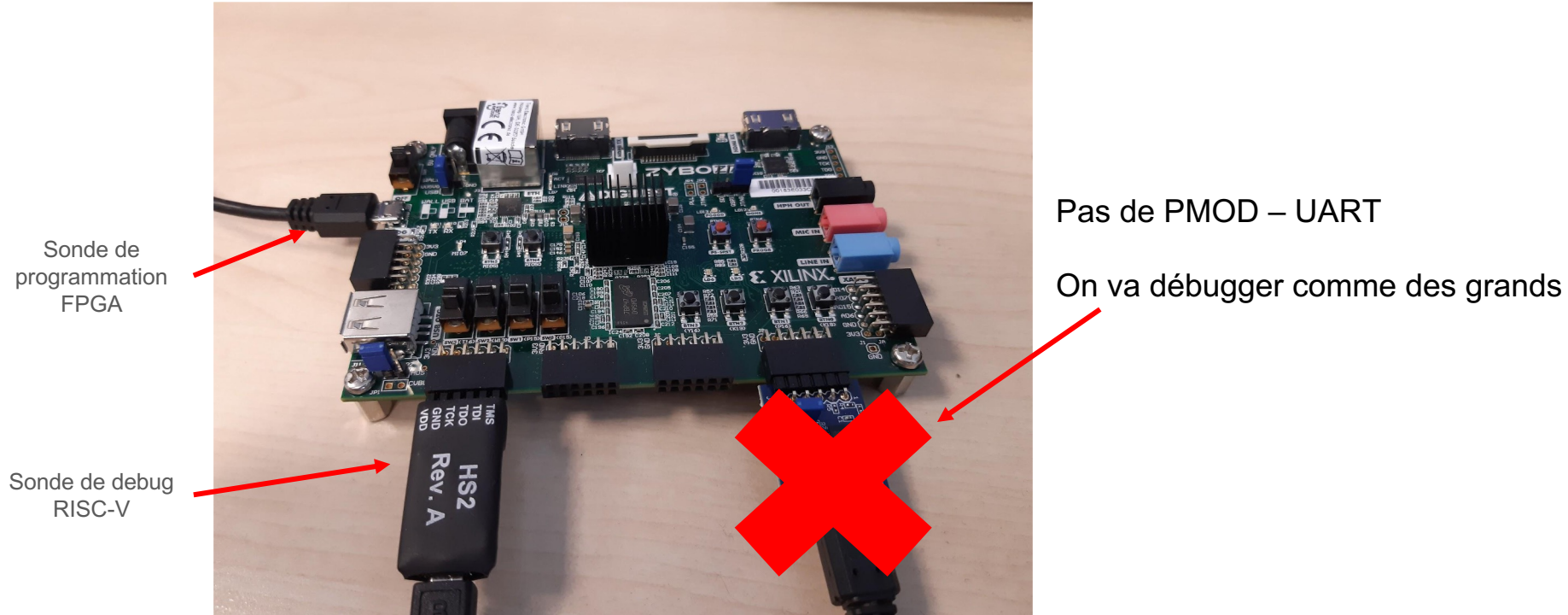
```
> make cva6_fpga
```

3. Création de l'image docker (compilation du compilateur GCC, interface de debug OpenOCD, programmes de test ...) (durée: environ 30 minutes)

```
> docker build -f Dockerfile --build-arg UID=$(id -u) --build-arg GID=$(id -g) -t sw-docker:v1 .
```

Prise en main du projet fourni par Thales (3/7)

Ordre de branchement de la carte ZYBO-Z2



Prise en main du projet fourni par Thales (4/7)

1. Programmation du FPGA

```
> make program_cva6_fpga
```

Une fois le FPGA programmé, les leds rouge et verte sont allumées → le cœur CVA6 a démarré.
Reset possible via le bouton BTN3 (au centre, à droite)

Ajout des options de compilation pour GDB

```
> modification du fichier cva6-softcore-contest/sw/app/Makefile
```

Ajouter -ggdb au variables CFLAGS et RISCVCFLAGS

```
CFLAGS ?=-march=rv32im_zicsr -mabi=ilp32 -DPREALLOCATE=1 -fvisibility=hidden -DSTDIO_THRU_UART -O3 -mcmodel=medany -static -Wall -pedantic -ggdb
RISCV_CFLAGS :=-DPERFORMANCE_RUN=1 \
    -DITERATIONS=3 \
    -DFLAGS_STR=\"$(FLAGS_STR)\" \
    -march=rv32im_zicsr -mabi=ilp32 -DPREALLOCATE=1 -fvisibility=hidden -DSTDIO_THRU_UART -O3 -mcmodel=medany -static -Wall -pedantic -ggdb
```

2. Lancement de l'image Docker

```
> sudo docker run -ti --privileged -v `realpath sw`:./workdir sw-docker:v1
```

Password : useruser

Prise en main du projet fourni par Thales (5/7)

Dans l'image Docker :

1. Compilation du programme de test (cible RISC-V)

```
> cd app  
> make mnist
```

2. Lancement d'OpenOCD (pour piloter la sonde JTAG RISC-V)

```
> openocd -f openocd_digilent_hs2.cfg &
```

Attention : le '&' est obligatoire car on va réutiliser le terminal pour GDB

3. Lancement de GDB (pour contrôler l'exécution du programme)

```
> riscv-none-elf-gdb mnist.riscv
```


Les bases de GDB

Pour lancer/continuer l'exécution d'un programme : `continue` (ou `c`)

Pour arrêter l'exécution d'un programme : `CTRL+C`

Ajout de points d'arrêt (breakpoints) : `b [*adresse|fonction]`

Avancer d'une instruction C : `step` (ou `s`)

Avancer d'une instruction en assembleur : `stepi` (ou `si`)

Pour afficher une variable/un registre : `print [variable|$registre]`
(ou `p`)

Pour afficher le contenu de la mémoire : `x adresse`

Possibilité de spécifier la quantité de mémoire : `x/[nombre][b|h|w]`

Modes d'affichage (pour les deux instructions précédentes) :

- `/a` : afficher une adresse
- `/x` : afficher en hexadécimal
- `/t` : afficher en binaire



Prise en main du projet fourni par Thales (6/7)

Dans l'interpréteur de commande GDB

1. Connexion à OpenOCD (port 3333)

```
(gdb) tar ext :3333
```

1. Chargement du programme

```
(gdb) load
```

2. Mise en place d'un point d'arrêt après l'inférence (ligne 71 dans le fichier main.c)

```
(gdb) b main.c:71
```

1. Lancement du programme

```
(gdb) c
```

Prise en main du projet fourni par Thales (7/7)

On peut à présent vérifier les résultats:

```
(gdb) p expectedOutputBuffer
```

```
{4}
```

```
(gdb) p predictedOutputBuffer
```

```
{4}
```

Input (image 28x28)

Résultat de l'inférence

Et les performances:

```
(gdb) p instret
```

```
1731593
```

```
(gdb) p cycles
```

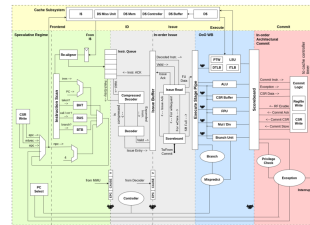
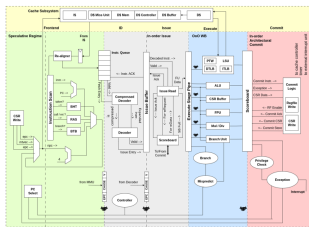
```
2353554
```

Nombre d'instructions exécutées

Nombre de cycles CPU

Et maintenant? A vous de jouer!


→ *Accélération de l'application MNIST*



Accélérateur
matériel

[illegible]

La solution des lauréats 2024 LATAM-V

Function	Number of calls	
macsOnRange	10274	
clamp	2480	
sat	2480	
saturate	2480	
read_mnist_input	4	
convcellPropagate1	2	
envRead	1	
fccellPropagateDATA _T	1	
fccellPropagateUDATA _T	1	
feof_mnist_input	1	
maxPropagate1	1	
processInput	1	
propagate	1	
readStimulus	1	

Optimisation de l'opération
« multiplier et accumuler »

La solution des lauréats 2024 LATAM-V

Opération « multiplier et accumuler »

```
1 lbu s2,0(s9) // We load 1 input
2 lb s3,0(a2) // We load 1 weight
3 mul s6,s2,s3 // Multiply both values
4 add a5,a5,s6 // Accumulate with previous results
```

Originellement : traitement octet par octet

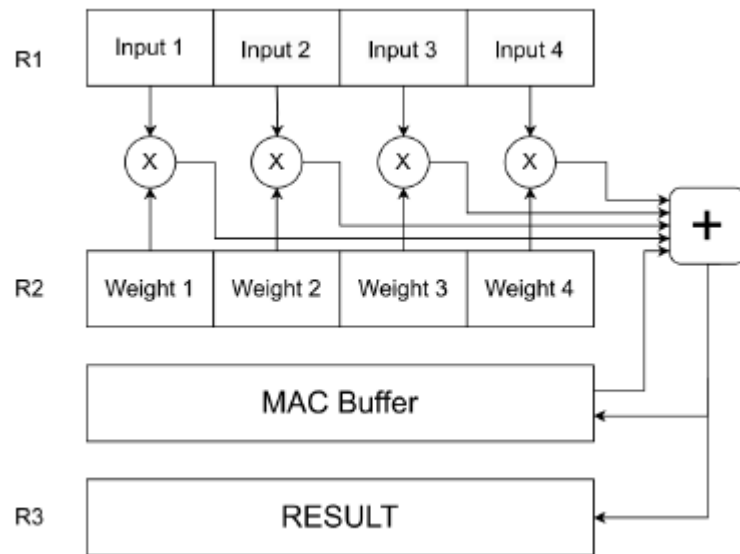
→ **Solution** : paralléliser les multiplications (tirer parti des registres du cœur de taille 32 bit)

La solution des lauréats 2024 LATAM-V

```
1 lbu s2,0(s9) // We load 1 input
2 lb s3,0(a2) // We load 1 weight
3 mul s6,s2,s3 // Multiply both values
4 add a5,a5,s6 // Accumulate with previous results
```



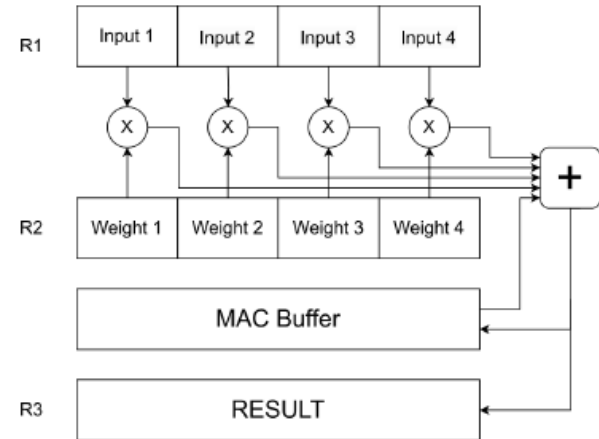
```
1 lw s2,0(s9) // We load 32 bits -> 4 inputs
2 lw s3,0(a2) // We load 32 bits -> 4 weights
3 customMAC s6,s2,s3 // 4 parallel Mac's
```




La solution des lauréats 2024 LATAM-V

Développement de deux instructions customs :

- **CustomMAC** : multiplications parallélisées et additions
- **RetrieveMAC** : récupération du résultat et remise à zéro



Implémenter la solution des lauréats 2024 LATAM-V...

- 
- Modification du design du cœur pour inclure une accélération matérielle
 - Modification d'un compilateur pour l'ajout d'instructions customs
 - Adaptation du code applicatif (utilisation des instructions customs)
 - Implémentation sur carte FPGA

Récupération du dépôt

On récupère le clone du dépôt des gagnants du concours 2024 (on en profite pour les féliciter):

```
git clone https://github.com/bernabeu-a/cva6-softcore-contest-compas2024
```



On se place dans le répertoire du dépôt:

```
cd ~/cva6-softwore-contest-compas2024
```

C'est la même structure que le dépôt du concours RISC-V.

Nous avons juste modifié la solution des gagnants pour vous faire travailler un peu

Allez, au boulot



Modification de l'architecture (1/3)

La première étape consiste à modifier les sources du design du cœur en SystemVerilog pour ajouter un accélérateur matériel pour l'application MNIST.

Afin de s'éviter un temps de synthèse du cœur, **les sources sont à compléter sous forme de code à trous et le design déjà synthétisé est fourni** pour la suite.

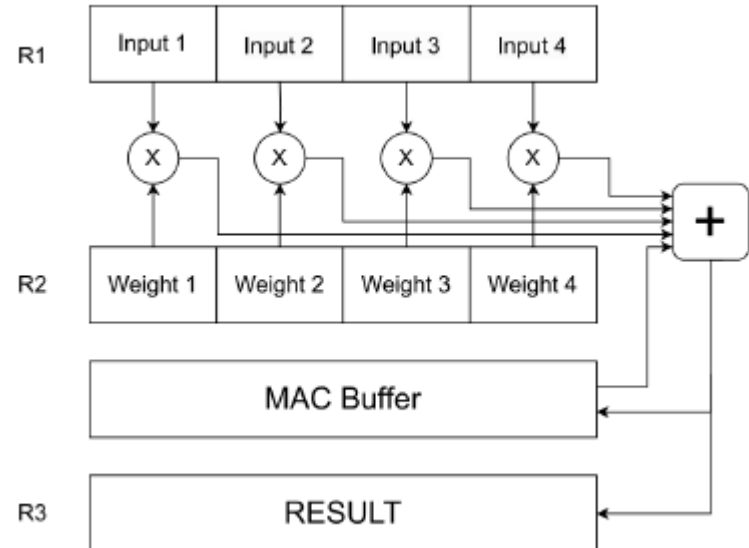
Ouvrir le fichier [/core/cvxif_example/cvxif_example_coprocessor.sv](#)

Compléter le code à trous l.150-153 et l.200-220 (voir TODO).

Modification de l'architecture (2/3)

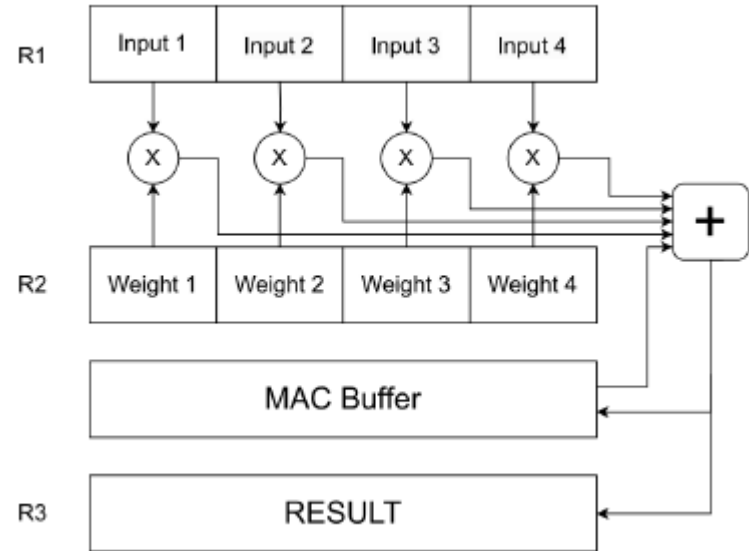
[/core/cvxif_example/cvxif_example_coprocessor.sv](#)

```
144 //TODO parallelization of MAC operations
145 end else begin
146     x_issue_ready_o <= x_issue_ready_q;
147     result_reg <= x_result_o.data;
148     custom_accumulator <= reset_acc? 32'd0: sum;
149     acc_reg <= acc;
150     //res_1 <= ??????;
151     //res_2 <= ??????;
152     //res_3 <= ??????;
153     //res_4 <= ??????;
154     unsigned_buffer[buffer_addr_reg] <= buffer_input;
155     buffer_output <= unsigned_buffer[buffer_addr];
156     buffer_addr_reg <= buffer_addr;
157     fill_buffer_reg <= fill_buffer;
158
159 end
160 end
```



Modification de l'architecture (2/3)

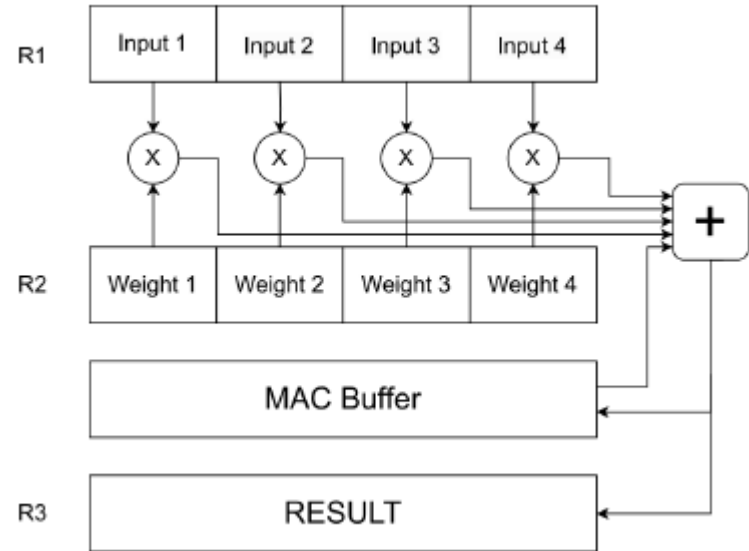
```
end else begin
  x_issue_ready_o <= x_issue_ready_q;
  result_reg <= x_result_o.data;
  custom_accumulator <= reset_acc? 32'd0: sum;
  acc_reg <= acc;
  res_1 <= unsigned_num_0*signed_num_0;
  res_2 <= unsigned_num_1*signed_num_1;
  res_3 <= unsigned_num_2*signed_num_2;
  res_4 <= unsigned_num_3*signed_num_3;
  unsigned_buffer[buffer_addr_reg] <= buffer_input;
  buffer_output <= unsigned_buffer[buffer_addr];
  buffer_addr_reg <= buffer_addr;
  fill_buffer_reg <= fill_buffer;
end
end
```



Modification de l'architecture (3/3)

[/core/cvxif_example/cvxif_example_coprocessor.sv](#)

```
198 //TODO parallelization of the multiplications (32-bit registers)
199 if (fill_buffer_reg==1) begin
200     /*unsigned_num_0 = req_o.req.rs[0][?:?];
201     unsigned_num_1 = req_o.req.rs[0][?:?];
202     unsigned_num_2 = req_o.req.rs[0][?:?];
203     unsigned_num_3 = req_o.req.rs[0][?:?];*/
204
205 end else begin
206     /*unsigned_num_0 = buffer_output[?:?];
207     unsigned_num_1 = buffer_output[?:?];
208     unsigned_num_2 = buffer_output[?:?];
209     unsigned_num_3 = buffer_output[?:?];*/
210
211 end
212
213
214 /*signed_num_0 = req_o.req.rs[1][?:?];
215 signed_num_1 = req_o.req.rs[1][?:?];
216 signed_num_2 = req_o.req.rs[1][?:?];
217 signed_num_3 = req_o.req.rs[1][?:?];*/
218
219 //TODO accumulation
220 // res = ??????;
```



Modification de l'architecture (3/3)

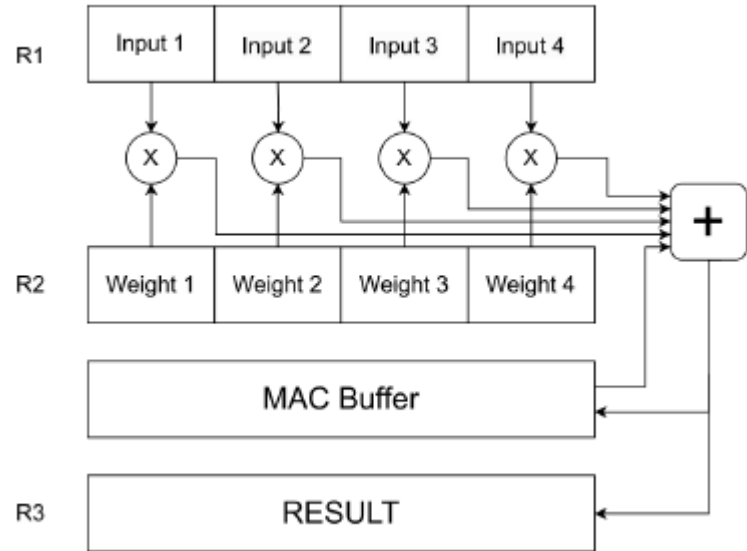
```
if (fill_buffer_reg==1) begin
    unsigned_num_0 = req_o.req.rs[0][7:0];
    unsigned_num_1 = req_o.req.rs[0][15:8];
    unsigned_num_2 = req_o.req.rs[0][23:16];
    unsigned_num_3 = req_o.req.rs[0][31:24];

end else begin
    unsigned_num_0 = buffer_output[7:0];
    unsigned_num_1 = buffer_output[15:8];
    unsigned_num_2 = buffer_output[23:16];
    unsigned_num_3 = buffer_output[31:24];

end

signed_num_0 = req_o.req.rs[1][7:0];
signed_num_1 = req_o.req.rs[1][15:8];
signed_num_2 = req_o.req.rs[1][23:16];
signed_num_3 = req_o.req.rs[1][31:24];

res = res_1+res_2+res_3+res_4;
```



Modification de la toolchain (1/5)

On a modifié l'architecture, il faut à présent flash sur le FPGA (le bitstream est fourni):

```
> make program_cva6_fpga
```

Après avoir modifié l'architecture,
il faut à présent pouvoir compiler un programme utilisant les modifications.

Lancement de l'image Docker:

```
> sudo docker run -ti --privileged -v `realpath sw`:/workdir sw-docker:v1
```

Password : useruser



Toute modification dans le docker est effacée si vous fermez le conteneur → il faudra recommencer

Modification de la toolchain (2/5)



Pas d'éditeur de texte dans le docker, installez votre préféré

```
> sudo apt install nano
```

On va modifier la toochain de compilation:

Ajout des OPCODE des instructions customs → On modifie `riscv-opc.c` et `riscv-opc.h`

Modification de la toolchain (3/5)

Le fichier riscv-opc.c:

```
> sudo nano /util/gcc-toolchain-builder/src/binutils-gdb/opcodes/riscv-opc.c
```

On ajoute les 2 instructions custom, ligne 321:

```
{"custom1", 0, INSN_CLASS_I, "d,s,t", MATCH_CUSTOM1, MASK_CUSTOM1, match_opcode, 0},  
{"custom2", 0, INSN_CLASS_I, "d,s,t", MATCH_CUSTOM2, MASK_CUSTOM2, match_opcode, 0},
```

```
/* Basic RVI instructions and aliases. */  
{ "custom1", 0, INSN_CLASS_I, "d,s,t", MATCH_CUSTOM1, MASK_CUSTOM1, match_opcode, 0 },  
{ "custom2", 0, INSN_CLASS_I, "d,s,t", MATCH_CUSTOM2, MASK_CUSTOM2, match_opcode, 0 },  
{ "unimp", 0, INSN_CLASS_C, "", 0, 0xffffU, match_opcode, INSN_ALIAS },  
{ "unimp", 0, INSN_CLASS_I, "", MATCH_CSRRW|(CSR_CYCLE << OP_SH_CSR), 0xffffffffU, match_opcode, 0 }, /* csr cycle, x0 */  
{ "ebreak", 0, INSN_CLASS_C, "", MATCH_C_EBREAK, MASK_C_EBREAK, match_opcode, INSN_ALIAS },  
{ "ebreak", 0, INSN_CLASS_I, "", MATCH_EBREAK, MASK_EBREAK, match_opcode, 0 },  
{ "sbreak", 0, INSN_CLASS_C, "", MATCH_C_EBREAK, MASK_C_EBREAK, match_opcode, INSN_ALIAS },
```

On oublie pas d'enregistrer !

Avec nano:

Enregistrer: ctrl+o

Quitter: ctrl+x

Modification de la toolchain (4/5)

Le fichier riscv-opc.h:

```
> sudo nano /util/gcc-toolchain-builder/src/binutils-gdb/include/opcode/riscv-opc.h
```

On ajoute les macros, ligne 24:

```
#define MATCH_CUSTOM1 0x2b  
#define MASK_CUSTOM1 0xfe00707f  
#define MATCH_CUSTOM2 0x5b  
#define MASK_CUSTOM2 0xfe00707f
```

```
#ifndef RISCV_ENCODING_H  
#define RISCV_ENCODING_H  
/* Instruction opcode macros. */  
#define MATCH_CUSTOM1 0x2b  
#define MASK_CUSTOM1 0xfe00707f  
#define MATCH_CUSTOM2 0x5b  
#define MASK_CUSTOM2 0xfe00707f  
  
#define MATCH_SLLI_RV32 0x1013
```

Et ligne 2790:

```
DECLARE_INSN(custom1, MATCH_CUSTOM1, MASK_CUSTOM1)  
DECLARE_INSN(custom2, MATCH_CUSTOM2, MASK_CUSTOM2)
```

```
#endif /* RISCV_ENCODING_H */  
DECLARE_INSN(custom1, MATCH_CUSTOM1, MASK_CUSTOM1)  
DECLARE_INSN(custom2, MATCH_CUSTOM2, MASK_CUSTOM2)  
  
DECLARE_INSN(slli_rv32, MATCH_SLLI_RV32, MASK_SLLI_RV32)  
DECLARE_INSN(srli_rv32, MATCH_SRLI_RV32, MASK_SRLI_RV32)  
DECLARE_INSN(srai_rv32, MATCH_SRAI_RV32, MASK_SRAI_RV32)  
DECLARE_INSN(frflags, MATCH_FRFLAGS, MASK_FRFLAGS)  
DECLARE_INSN(fsflags, MATCH_FSFLAGS, MASK_FSFLAGS)
```

Modification de la toolchain (5/5)

On peut compiler la toolchain avec les fichiers modifiés:

```
> cd /util/gcc-toolchain-builder
```

```
> export RISCv=riscv_toolchain
```

```
> sudo bash ./build-toolchain.sh $RISCv
```

```
> PATH="$PATH:/util/gcc-toolchain-build/riscv_toolchain/bin"
```

La toolchain est prête pour les instructions !

Modification de l'application (1/2)

On modifie uniquement les fonctions **CustomMacsOnRange1**, **CustomMacsOnRange2** et **MacsResultBufferVersion** dans l'application MNIST (./sw/app/mnist/NetworkPropagate.c)

Note, on peut modifier les fichiers de l'application en dehors du conteneur, le dossier ./sw est monté dans le docker

```
static inline void CustomMacsOnRange1(const UDATA_T* __restrict inputs,
                                     const WDATA_T* __restrict weights,
                                     int nb_iterations)
{
    /* ToDo:
       iterate over nb_iterations:
       input is now a 32 bit (instead of 8 bit)
       put 4 input of 8 bits in input_ 32bits
       call custom instruction to perform MAC operations
    */

    SUM_T res=0;
    int32_t input_;
    int32_t weights_;
    for (int iter = 0; iter < nb_iterations; ++iter) {
        /* CHANGE INPUT_ and WEIGHTS_ */
        input_ = CHANGE_ME;
        weights_ = CHANGE_ME;
        asm volatile
        (
            "custom1 %[z], %[x], %[y]\n\t"
            : [z] "=r" (res)
            : [x] "r" (input_), [y] "r" (weights_)
            );
    }
}
```

```
static inline void CustomMacsOnRange2(const UDATA_T* __restrict inputs,
                                     const WDATA_T* __restrict weights,
                                     int nb_iterations)
{
    /* ToDo:
       iterate over nb_iterations:
       input is now a 32 bit (instead of 8 bit)
       put 4 input of 8 bits in i1 32bits
       put 4 input of 8 bits in i2 32bits
       weights is now a 32 bit (instead of 8 bit)
       put 4 weights of 8 bits in w1 32bits
       put 4 weights of 8 bits in w2 32bits
       call custom instruction to perform MAC operations on i1/i2 and w1/w2
    */

    SUM_T res=0;
    for (int iter = 0; iter < nb_iterations/2; ++iter) {
        /* CHANGE HERE */

        int32_t w1 = CHANGE_ME;
        int32_t w2 = CHANGE_ME;
        uint32_t i1 = CHANGE_ME;
        uint32_t i2 = CHANGE_ME;

        asm volatile (
            "custom1 %[res1], %[i1_reg], %[w1_reg]\n\t"
            "custom1 %[res2], %[i2_reg], %[w2_reg]\n\t"
            : [res1] "=&r" (res), [res2] "=&r" (res)
            : [i1_reg] "r" (i1), [w1_reg] "r" (w1), [i2_reg] "r" (i2), [w2_reg] "r" (w2)
            );
    }
}
```

Modification de l'application (2/2)

On retourne dans les applications pour compiler MNIST

```
> cd /workdir/app
```

```
> make mnist
```

On en profite aussi pour compiler le MNIST original, sans les customisations

```
> make mnist_original
```

On peut debugger comme précédemment avec openocd + GDB (slide 13)

Il y a-t-il une amélioration ?

Pour conclure

Image de la VM VirtualBox disponible sur FileServer :

- <https://filesender.renater.fr/?s=download&token=d29b007a-3394-4318-a5f4-c6631f9a278c>
- <https://bit.ly/3xgbuJt>
- Login : user
- Mot de passe : useruser

Solutions fournies dans le dossier solution du dépôt <https://github.com/bernabeu-a/cva6-softcore-contest-compas2024>


```

144 //TODO parallelization of MAC operations
145 end else begin
146     x_issue_ready_o <= x_issue_ready_q;
147     result_reg <= x_result_o.data;
148     custom_accumulator <= reset_acc? 32'd0: sum;
149     acc_reg <= acc;
150     //res_1 <= ??????;
151     //res_2 <= ??????;
152     //res_3 <= ??????;
153     //res_4 <= ??????;
154     unsigned_buffer[buffer_addr_reg] <= buffer_input;
155     buffer_output <= unsigned_buffer[buffer_addr];
156     buffer_addr_reg <= buffer_addr;
157     fill_buffer_reg <= fill_buffer;
158
159 end
160 end

```

```

always_ff @(posedge clk_i or negedge rst_ni) begin : regs
    if (!rst_ni) begin
        x_issue_ready_o <= 1;
        custom_accumulator <= 32'd0;

        buffer_addr_reg <= 5'd0;
        for (int i = 0; i < 128; i++) begin
            unsigned_buffer[i] = 32'd0;
        end
        fill_buffer_reg <= 1;
    end else begin
        x_issue_ready_o <= x_issue_ready_q;
        result_reg <= x_result_o.data;
        custom_accumulator <= reset_acc? 32'd0: sum;
        acc_reg <= acc;
        res_1 <= unsigned_num_0*signed_num_0;
        res_2 <= unsigned_num_1*signed_num_1;
        res_3 <= unsigned_num_2*signed_num_2;
        res_4 <= unsigned_num_3*signed_num_3;
        unsigned_buffer[buffer_addr_reg] <= buffer_input;
        buffer_output <= unsigned_buffer[buffer_addr];
        buffer_addr_reg <= buffer_addr;
        fill_buffer_reg <= fill_buffer;
    end
end
end

```

```

198 //TODO parallelization of the multiplications (32-bit registers)
199 if (fill_buffer_reg==1) begin
200     /*unsigned_num_0 = req_o.req.rs[0][?:?];
201     unsigned_num_1 = req_o.req.rs[0][?:?];
202     unsigned_num_2 = req_o.req.rs[0][?:?];
203     unsigned_num_3 = req_o.req.rs[0][?:?];*/
204
205 end else begin
206     /*unsigned_num_0 = buffer_output[?:?];
207     unsigned_num_1 = buffer_output[?:?];
208     unsigned_num_2 = buffer_output[?:?];
209     unsigned_num_3 = buffer_output[?:?];*/
210
211 end
212
213
214 /*signed_num_0 = req_o.req.rs[1][?:?];
215 signed_num_1 = req_o.req.rs[1][?:?];
216 signed_num_2 = req_o.req.rs[1][?:?];
217 signed_num_3 = req_o.req.rs[1][?:?];*/
218
219 //TODO accumulation
220 // res = ??????;

```

```

if (fill_buffer_reg==1) begin
    unsigned_num_0 = req_o.req.rs[0][7:0];
    unsigned_num_1 = req_o.req.rs[0][15:8];
    unsigned_num_2 = req_o.req.rs[0][23:16];
    unsigned_num_3 = req_o.req.rs[0][31:24];

end else begin
    unsigned_num_0 = buffer_output[7:0];
    unsigned_num_1 = buffer_output[15:8];
    unsigned_num_2 = buffer_output[23:16];
    unsigned_num_3 = buffer_output[31:24];

end

signed_num_0 = req_o.req.rs[1][7:0];
signed_num_1 = req_o.req.rs[1][15:8];
signed_num_2 = req_o.req.rs[1][23:16];
signed_num_3 = req_o.req.rs[1][31:24];

res = res_1+res_2+res_3+res_4;

```