TD n°3 : Réalisation d'instructions SIMD pour le processeur NIOS II

0. Introduction

On peut définir de nouvelles instructions (« customization ») pour le processeur NIOS II (cœur logiciel pour FPGA d'Altera), selon le schéma

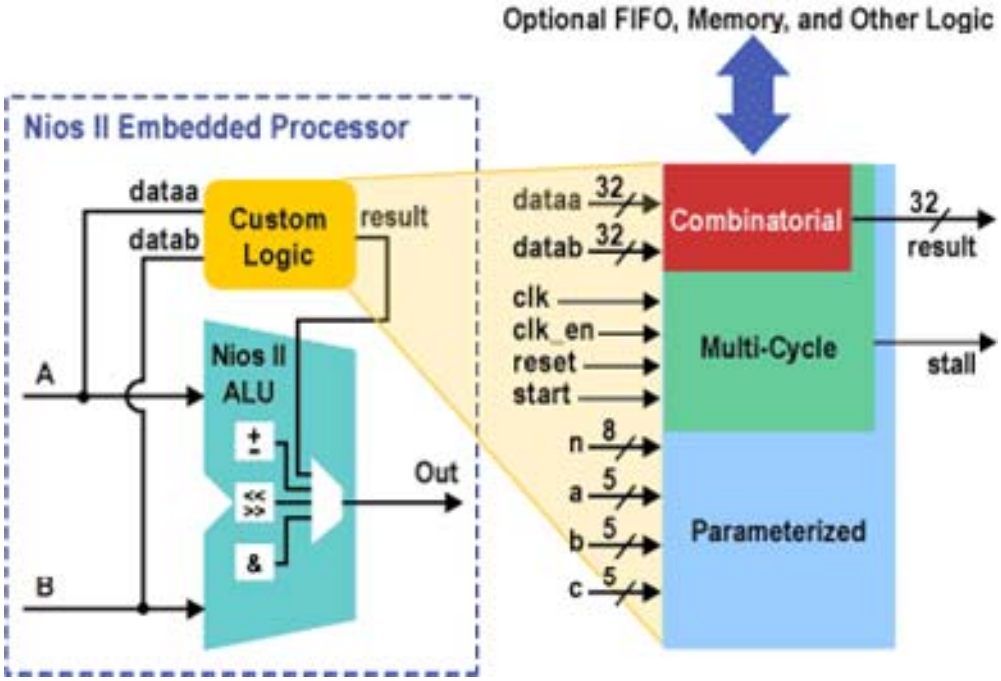

Figure 1 : Spécialisation d'instructions pour le processeurs NIOS II

1. Traitement d'images

Opérateur min

L'opérateur min est implanté en C à l'aide d'une fonction. Dans une première étape, on veut réaliser un opérateur MIN sur des entiers non signés 32 bits. Ecrire le code VHDL permettant d'implanter le matériel pour une instruction spécialisée de ce type.

Filtre MIN 3 x 3 pour le traitement d'images.

On veut réaliser les opérateurs matériels pour des instructions SIMD permettant de traiter simultanément 4 octets non signés (pixels) pour le filtre MIN. Les instructions à définir sont

- Décalage gauche d'un octet pour obtenir le mot de 32 bits pour les pixels X[i][j-1].
- Décalage droite d'un octet pour obtenir le mot de 32 bits pour les pixels X[i][j+1]
- Instruction SIMD de minimum sur des octets non signés.

Ecrire le code VHDL pour l'implantation de ces différentes instructions.

2. Cryptographie

Soit le code C implantant une version « octets » d'AES 256. Le code complet est fourni en annexe et accessible sur le site du cours.

Une présentation d'AES 128 est donnée dans l'article de Texas Instruments : « AES-128 – A C implementation for Encryption and Decryption ».

On veut accélérer l'exécution de certaines fonctions d'AES-256 « octets » avec des instructions spécialisées ajoutées au jeu d'instructions du processeur NIOS-II

Fonction aes_addRoundKey.

```
void aes_addRoundKey(uint8_t *buf, uint8_t *key)  //uint8_t = unsigned char
{
    register uint8_t i = 16;
    while (i--) buf[i] ^= key[i];
} /* aes_addRoundKey */
```

Définir le code VHDL pour une instruction spécialisée NIOS permettant d'accélérer la fonction aes_addRoundKey. L'instruction correspondante s'appellera ADDROUNDKEY (int a, int b).
Réécrire la fonction `aes_addRoundKeyM` qui utilise cette instruction.

Fonction aes_subBytes

```
void aes_subBytes(uint8_t *buf)
{
    register uint8_t i = 16;

    while (i--) buf[i] = rj_sbox(buf[i]);
} /* aes_subBytes */
```

Définir une instruction spécialisée SUBBYTES (int a) et réécrire la fonction `aes_subBytesM`

Fonction aes_mixcolumns

```
void aes_mixColumns(uint8_t *buf)
{
    register uint8_t i, a, b, c, d, e;
    for (i = 0; i < 16; i += 4)
    {
        a = buf[i]; b = buf[i + 1]; c = buf[i + 2]; d = buf[i + 3];
        e = a ^ b ^ c ^ d;
        buf[i] ^= e ^ rj_xtime(a^b);   buf[i+1] ^= e ^ rj_xtime(b^c);
        buf[i+2] ^= e ^ rj_xtime(c^d); buf[i+3] ^= e ^ rj_xtime(d^a);
    }
} /* aes_mixColumns */
```

Définir une instruction spécialisée MIXCOLUMNS (int a) et réécrire la fonction `aes_mixColumns`

Autres fonctions

Examiner les autres fonctions pour déterminer celles susceptibles d'être accélérées par des instructions spécialisées.

3. Annexe : code AES 256

```
/*
*   Byte-oriented AES-256 implementation.
*   All lookup tables replaced with 'on the fly' calculations.
*
*   Copyright (c) 2007-2009 Ilya O. Levin, http://www.literatecode.com
*   Other contributors: Hal Finney
*
*   Permission to use, copy, modify, and distribute this software for any
*   purpose with or without fee is hereby granted, provided that the above
*   copyright notice and this permission notice appear in all copies.
*
*   THE SOFTWARE IS PROVIDED "AS IS" AND THE AUTHOR DISCLAIMS ALL WARRANTIES
*   WITH REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED WARRANTIES OF
*   MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR
*   ANY SPECIAL, DIRECT, INDIRECT, OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
*   WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN
*   ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF
*   OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
*/

#include <stdlib.h>
#include <stdio.h>


#define F(x)   (((x)<<1) ^ ((((x)>>7) & 1) * 0x1b))
#define FD(x)  (((x) >> 1) ^ (((x) & 1) ? 0x8d : 0))

#define uint8_t  unsigned char


   typedef struct {
        uint8_t key[32];
        uint8_t enckey[32];
        uint8_t deckey[32];
    } aes256_context;
const uint8_t sbox[256] = {
    0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5,
    0x30, 0x01, 0x67, 0x2b, 0xfe, 0xd7, 0xab, 0x76,
    0xca, 0x82, 0xc9, 0x7d, 0xfa, 0x59, 0x47, 0xf0,
    0xad, 0xd4, 0xa2, 0xaf, 0x9c, 0xa4, 0x72, 0xc0,
    0xb7, 0xfd, 0x93, 0x26, 0x36, 0x3f, 0xf7, 0xcc,
    0x34, 0xa5, 0xe5, 0xf1, 0x71, 0xd8, 0x31, 0x15,
    0x04, 0xc7, 0x23, 0xc3, 0x18, 0x96, 0x05, 0x9a,
    0x07, 0x12, 0x80, 0xe2, 0xeb, 0x27, 0xb2, 0x75,
    0x09, 0x83, 0x2c, 0x1a, 0x1b, 0x6e, 0x5a, 0xa0,
    0x52, 0x3b, 0xd6, 0xb3, 0x29, 0xe3, 0x2f, 0x84,
    0x53, 0xd1, 0x00, 0xed, 0x20, 0xfc, 0xb1, 0x5b,
    0x6a, 0xcb, 0xbe, 0x39, 0x4a, 0x4c, 0x58, 0xcf,
    0xd0, 0xef, 0xaa, 0xfb, 0x43, 0x4d, 0x33, 0x85,
    0x45, 0xf9, 0x02, 0x7f, 0x50, 0x3c, 0x9f, 0xa8,
    0x51, 0xa3, 0x40, 0x8f, 0x92, 0x9d, 0x38, 0xf5,
    0xbc, 0xb6, 0xda, 0x21, 0x10, 0xff, 0xf3, 0xd2,
    0xcd, 0x0c, 0x13, 0xec, 0x5f, 0x97, 0x44, 0x17,
    0xc4, 0xa7, 0x7e, 0x3d, 0x64, 0x5d, 0x19, 0x73,
    0x60, 0x81, 0x4f, 0xdc, 0x22, 0x2a, 0x90, 0x88,
    0x46, 0xee, 0xb8, 0x14, 0xde, 0x5e, 0x0b, 0xdb,
```

```
    0xe0, 0x32, 0x3a, 0x0a, 0x49, 0x06, 0x24, 0x5c,
    0xc2, 0xd3, 0xac, 0x62, 0x91, 0x95, 0xe4, 0x79,
    0xe7, 0xc8, 0x37, 0x6d, 0x8d, 0xd5, 0x4e, 0xa9,
    0x6c, 0x56, 0xf4, 0xea, 0x65, 0x7a, 0xae, 0x08,
    0xba, 0x78, 0x25, 0x2e, 0x1c, 0xa6, 0xb4, 0xc6,
    0xe8, 0xdd, 0x74, 0x1f, 0x4b, 0xbd, 0x8b, 0x8a,
    0x70, 0x3e, 0xb5, 0x66, 0x48, 0x03, 0xf6, 0x0e,
    0x61, 0x35, 0x57, 0xb9, 0x86, 0xc1, 0x1d, 0x9e,
    0xe1, 0xf8, 0x98, 0x11, 0x69, 0xd9, 0x8e, 0x94,
    0x9b, 0x1e, 0x87, 0xe9, 0xce, 0x55, 0x28, 0xdf,
    0x8c, 0xa1, 0x89, 0x0d, 0xbf, 0xe6, 0x42, 0x68,
    0x41, 0x99, 0x2d, 0x0f, 0xb0, 0x54, 0xbb, 0x16
};
const uint8_t sboxinv[256] = {
    0x52, 0x09, 0x6a, 0xd5, 0x30, 0x36, 0xa5, 0x38,
    0xbf, 0x40, 0xa3, 0x9e, 0x81, 0xf3, 0xd7, 0xfb,
    0x7c, 0xe3, 0x39, 0x82, 0x9b, 0x2f, 0xff, 0x87,
    0x34, 0x8e, 0x43, 0x44, 0xc4, 0xde, 0xe9, 0xcb,
    0x54, 0x7b, 0x94, 0x32, 0xa6, 0xc2, 0x23, 0x3d,
    0xee, 0x4c, 0x95, 0x0b, 0x42, 0xfa, 0xc3, 0x4e,
    0x08, 0x2e, 0xa1, 0x66, 0x28, 0xd9, 0x24, 0xb2,
    0x76, 0x5b, 0xa2, 0x49, 0x6d, 0x8b, 0xd1, 0x25,
    0x72, 0xf8, 0xf6, 0x64, 0x86, 0x68, 0x98, 0x16,
    0xd4, 0xa4, 0x5c, 0xcc, 0x5d, 0x65, 0xb6, 0x92,
    0x6c, 0x70, 0x48, 0x50, 0xfd, 0xed, 0xb9, 0xda,
    0x5e, 0x15, 0x46, 0x57, 0xa7, 0x8d, 0x9d, 0x84,
    0x90, 0xd8, 0xab, 0x00, 0x8c, 0xbc, 0xd3, 0x0a,
    0xf7, 0xe4, 0x58, 0x05, 0xb8, 0xb3, 0x45, 0x06,
    0xd0, 0x2c, 0x1e, 0x8f, 0xca, 0x3f, 0x0f, 0x02,
    0xc1, 0xaf, 0xbd, 0x03, 0x01, 0x13, 0x8a, 0x6b,
    0x3a, 0x91, 0x11, 0x41, 0x4f, 0x67, 0xdc, 0xea,
    0x97, 0xf2, 0xcf, 0xce, 0xf0, 0xb4, 0xe6, 0x73,
    0x96, 0xac, 0x74, 0x22, 0xe7, 0xad, 0x35, 0x85,
    0xe2, 0xf9, 0x37, 0xe8, 0x1c, 0x75, 0xdf, 0x6e,
    0x47, 0xf1, 0x1a, 0x71, 0x1d, 0x29, 0xc5, 0x89,
    0x6f, 0xb7, 0x62, 0x0e, 0xaa, 0x18, 0xbe, 0x1b,
    0xfc, 0x56, 0x3e, 0x4b, 0xc6, 0xd2, 0x79, 0x20,
    0x9a, 0xdb, 0xc0, 0xfe, 0x78, 0xcd, 0x5a, 0xf4,
    0x1f, 0xdd, 0xa8, 0x33, 0x88, 0x07, 0xc7, 0x31,
    0xb1, 0x12, 0x10, 0x59, 0x27, 0x80, 0xec, 0x5f,
    0x60, 0x51, 0x7f, 0xa9, 0x19, 0xb5, 0x4a, 0x0d,
    0x2d, 0xe5, 0x7a, 0x9f, 0x93, 0xc9, 0x9c, 0xef,
    0xa0, 0xe0, 0x3b, 0x4d, 0xae, 0x2a, 0xf5, 0xb0,
    0xc8, 0xeb, 0xbb, 0x3c, 0x83, 0x53, 0x99, 0x61,
    0x17, 0x2b, 0x04, 0x7e, 0xba, 0x77, 0xd6, 0x26,
    0xe1, 0x69, 0x14, 0x63, 0x55, 0x21, 0x0c, 0x7d
};
#define rj_sbox(x)     sbox[(x)]
#define rj_sbox_inv(x) sboxinv[(x)]
#define DUMP(s, i, buf, sz)  {printf(s);                    \
                              for (i = 0; i < (sz);i++)     \
                                  printf("%02x ", buf[i]); \
                              printf("\n");}

 void aes256_init(aes256_context *, uint8_t * /* key */);
    void aes256_done(aes256_context *);
    void aes256_encrypt_ecb(aes256_context *, uint8_t * /* plaintext */);
    void aes256_decrypt_ecb(aes256_context *, uint8_t * /* cipertext */);

uint8_t rj_xtime(uint8_t x)
```

```
{
    return (x & 0x80) ? ((x << 1) ^ 0x1b) : (x << 1);
} /* rj_xtime */
/* -------------------------------------------------------------------------*/
void aes_subBytes(uint8_t *buf)
{
    register uint8_t i = 16;

    while (i--) buf[i] = rj_sbox(buf[i]);
} /* aes_subBytes */

/* -------------------------------------------------------------------------*/
void aes_subBytes_inv(uint8_t *buf)
{
    register uint8_t i = 16;

    while (i--) buf[i] = rj_sbox_inv(buf[i]);
} /* aes_subBytes_inv */

/* -------------------------------------------------------------------------*/
void aes_addRoundKey(uint8_t *buf, uint8_t *key)
{
    register uint8_t i = 16;

    while (i--) buf[i] ^= key[i];
} /* aes_addRoundKey */

/* -------------------------------------------------------------------------*/
void aes_addRoundKey_cpy(uint8_t *buf, uint8_t *key, uint8_t *cpk)
{
    register uint8_t i = 16;

    while (i--)  buf[i] ^= (cpk[i] = key[i]), cpk[16+i] = key[16 + i];
} /* aes_addRoundKey_cpy */
/* -------------------------------------------------------------------------*/
void aes_shiftRows(uint8_t *buf)
{
    register uint8_t i, j; /* to make it potentially parallelable :) */

    i = buf[1]; buf[1] = buf[5]; buf[5] = buf[9]; buf[9] = buf[13]; buf[13] =
i;
    i = buf[10]; buf[10] = buf[2]; buf[2] = i;
    j = buf[3]; buf[3] = buf[15]; buf[15] = buf[11]; buf[11] = buf[7]; buf[7]
= j;
    j = buf[14]; buf[14] = buf[6]; buf[6]  = j;
} /* aes_shiftRows */
/* -------------------------------------------------------------------------- */
void aes_shiftRows_inv(uint8_t *buf)
{
    register uint8_t i, j; /* same as above :) */

    i = buf[1]; buf[1] = buf[13]; buf[13] = buf[9]; buf[9] = buf[5]; buf[5] =
i;
    i = buf[2]; buf[2] = buf[10]; buf[10] = i;
    j = buf[3]; buf[3] = buf[7]; buf[7] = buf[11]; buf[11] = buf[15]; buf[15]
= j;
    j = buf[6]; buf[6] = buf[14]; buf[14] = j;
} /* aes_shiftRows_inv */
/* -------------------------------------------------------------------------- */
void aes_mixColumns(uint8_t *buf)
```

```
{
    register uint8_t i, a, b, c, d, e;
    for (i = 0; i < 16; i += 4)
    {
        a = buf[i]; b = buf[i + 1]; c = buf[i + 2]; d = buf[i + 3];
        e = a ^ b ^ c ^ d;
        buf[i] ^= e ^ rj_xtime(a^b);   buf[i+1] ^= e ^ rj_xtime(b^c);
        buf[i+2] ^= e ^ rj_xtime(c^d); buf[i+3] ^= e ^ rj_xtime(d^a);
    }
} /* aes_mixColumns */
/* ------------------------------------------------------------------------ */
void aes_mixColumns_inv(uint8_t *buf)
{
    register uint8_t i, a, b, c, d, e, x, y, z;

    for (i = 0; i < 16; i += 4)
    {
        a = buf[i]; b = buf[i + 1]; c = buf[i + 2]; d = buf[i + 3];
        e = a ^ b ^ c ^ d;
        z = rj_xtime(e);
        x = e ^ rj_xtime(rj_xtime(z^a^c));  y = e ^ rj_xtime(rj_xtime(z^b^d));
        buf[i] ^= x ^ rj_xtime(a^b);   buf[i+1] ^= y ^ rj_xtime(b^c);
        buf[i+2] ^= x ^ rj_xtime(c^d); buf[i+3] ^= y ^ rj_xtime(d^a);
    }
} /* aes_mixColumns_inv */
/* ------------------------------------------------------------------------ */
void aes_expandEncKey(uint8_t *k, uint8_t *rc)
{
    register uint8_t i;
    k[0] ^= rj_sbox(k[29]) ^ (*rc);
    k[1] ^= rj_sbox(k[30]);
    k[2] ^= rj_sbox(k[31]);
    k[3] ^= rj_sbox(k[28]);
    *rc = F( *rc);
    for(i = 4; i < 16; i += 4)  k[i] ^= k[i-4],   k[i+1] ^= k[i-3],
        k[i+2] ^= k[i-2], k[i+3] ^= k[i-1];
    k[16] ^= rj_sbox(k[12]);
    k[17] ^= rj_sbox(k[13]);
    k[18] ^= rj_sbox(k[14]);
    k[19] ^= rj_sbox(k[15]);
    for(i = 20; i < 32; i += 4) k[i] ^= k[i-4],   k[i+1] ^= k[i-3],
        k[i+2] ^= k[i-2], k[i+3] ^= k[i-1];
} /* aes_expandEncKey */
/* ------------------------------------------------------------------------ */
void aes_expandDecKey(uint8_t *k, uint8_t *rc)
{
    uint8_t i;
    for(i = 28; i > 16; i -= 4) k[i+0] ^= k[i-4], k[i+1] ^= k[i-3],
        k[i+2] ^= k[i-2], k[i+3] ^= k[i-1];
    k[16] ^= rj_sbox(k[12]);
    k[17] ^= rj_sbox(k[13]);
    k[18] ^= rj_sbox(k[14]);
    k[19] ^= rj_sbox(k[15]);
    for(i = 12; i > 0; i -= 4)  k[i+0] ^= k[i-4], k[i+1] ^= k[i-3],
        k[i+2] ^= k[i-2], k[i+3] ^= k[i-1];
    *rc = FD(*rc);
    k[0] ^= rj_sbox(k[29]) ^ (*rc);
    k[1] ^= rj_sbox(k[30]);
    k[2] ^= rj_sbox(k[31]);
    k[3] ^= rj_sbox(k[28]);
```

```
} /* aes_expandDecKey */
/* ----------------------------------------------------------------------- */
void aes256_init(aes256_context *ctx, uint8_t *k)
{
    uint8_t rcon = 1;
    register uint8_t i;
    for (i = 0; i < sizeof(ctx->key); i++) ctx->enckey[i] = ctx->deckey[i] =
k[i];
    for (i = 8;--i;) aes_expandEncKey(ctx->deckey, &rcon);
} /* aes256_init */
/* ----------------------------------------------------------------------- */
void aes256_done(aes256_context *ctx)
{
    register uint8_t i;
    for (i = 0; i < sizeof(ctx->key); i++)
        ctx->key[i] = ctx->enckey[i] = ctx->deckey[i] = 0;
} /* aes256_done */
/* ----------------------------------------------------------------------- */
void aes256_encrypt_ecb(aes256_context *ctx, uint8_t *buf)
{
    uint8_t i, rcon;
    aes_addRoundKey_cpy(buf, ctx->enckey, ctx->key);
    for(i = 1, rcon = 1; i < 14; ++i)
    {
        aes_subBytes(buf);
        aes_shiftRows(buf);
        aes_mixColumns(buf);
        if( i & 1 ) aes_addRoundKey( buf, &ctx->key[16]);
        else aes_expandEncKey(ctx->key, &rcon), aes_addRoundKey(buf, ctx-
>key);
    }
    aes_subBytes(buf);
    aes_shiftRows(buf);
    aes_expandEncKey(ctx->key, &rcon);
    aes_addRoundKey(buf, ctx->key);
} /* aes256_encrypt */
/* ----------------------------------------------------------------------- */
void aes256_decrypt_ecb(aes256_context *ctx, uint8_t *buf)
{
    uint8_t i, rcon;

    aes_addRoundKey_cpy(buf, ctx->deckey, ctx->key);
    aes_shiftRows_inv(buf);
    aes_subBytes_inv(buf);
    for (i = 14, rcon = 0x80; --i;)
    {
        if( ( i & 1 ) )
        {
            aes_expandDecKey(ctx->key, &rcon);
            aes_addRoundKey(buf, &ctx->key[16]);
        }
        else aes_addRoundKey(buf, ctx->key);
        aes_mixColumns_inv(buf);
        aes_shiftRows_inv(buf);
        aes_subBytes_inv(buf);
    }
    aes_addRoundKey( buf, ctx->key);
} /* aes256_decrypt */

int main (int argc, char *argv[])
```

```
{
      double t1, t2, texe, overhead;
    aes256_context ctx;
    uint8_t key[32];
    uint8_t buf[16], i;
    /* put a test vector */
    for (i = 0; i < sizeof(buf);i++) buf[i] = i * 16 + i;
    for (i = 0; i < sizeof(key);i++) key[i] = i;
    DUMP("txt: ", i, buf, sizeof(buf));
    DUMP("key: ", i, key, sizeof(key));
    printf("---\n");
    aes256_init(&ctx, key);
    aes256_encrypt_ecb(&ctx, buf);
    DUMP("enc: ", i, buf, sizeof(buf));
    printf("tst: 8e a2 b7 ca 51 67 45 bf ea fc 49 90 4b 49 60 89\n");
    aes256_init(&ctx, key);
    aes256_decrypt_ecb(&ctx, buf);
    DUMP("dec: ", i, buf, sizeof(buf));
    aes256_done(&ctx);
    return 0;
} /* main */
```