# Fonctionnement de fctools-emu:

## Intégration possible en tant qu'ISS dans un environnement de simulation systemC.

### *1) Fctools*

Fctools est un ensemble d'outil écrit par Michael Riepe <michael@stud.uni-hannover.de>. Il est capable d'assembler, de lier, et de lancer des programmes au format ELF pour le fcpu. On a ainsi un environnement pour tester des concepts de conventions d'appels, de compilation, etc. Le fichier README de l'archive explique rapidement les bases:

```
What's in here? Besides the set of
generic ELF tools -- ar, elfdump,mcs,
nm, ranlib, size, strings and strip --
there are an F-CPU assembler(as), a
disassembler (disasm), a linker (ld)
and two instruction-level emulators.
The emulators share the same emulation
engine, but while `emu' accepts only
fully-relocated, binary images that
start at physical address 0, `elfemu'
was designed to run `real' binaries,
i.e. Statically linked ELF files.
That enables us to do both `bare
metal' development and higher-level
software ports.
```

*Text 1fichier README*

Avec fctools et le port de gcc pour le fcpu, on est capable de générer et de tester du code pour le processeur fcpu.

### 1.1) Architecture des fctools.

2 parties distinctes: une partie basée sur libELF qui implémente les outils de base de gestion des fichiers binaires au format ELF. Une partie processeur, qui implémente un assembleur, un désassembleur, et un *émulateur*. Ces 3 derniers programmes sont ceux qui nous interressent.

Le jeu d'instruction de fcpu ne bougera normalement que très peu, mais bougera tout de même. Il conviens alors que les 3 outils soient synchronisés sur le même jeu d'instruction. Pour aider à cela, un répertoire **fcpu_opcodes** contient les définitions des intructions, en grande partie sous forme d'enumérations (constantes C) déclarées dans le fichier **fcpu_opcodes/fcpu_opcodes.h**:

```
/*
 * ROP2 functions
 */
enum ROP2_function {
        ROP2_FUNC_AND,
        ROP2_FUNC_ANDN,
        ROP2_FUNC_XOR,
        ROP2_FUNC_OR,
        ROP2_FUNC_NOR,
        ROP2_FUNC_XNOR,
        ROP2_FUNC_ORN,
        ROP2_FUNC_NAND,
        ROP2_FUNC_last
};

/*
 * The opcode list
 */
enum fcpu_opcodes {     /* XXX: subject to
change! */
    OP_NOP,
    OP_MOVE,
    /*
     * Opcodes 2...3 currently unassigned
     */
    OP_BITOP        = 4,    /* uses 4 opcodes
*/
    OP_BTST         = OP_BITOP + ROP2_FUNC_AND,
    OP_BCLR         = OP_BITOP +
ROP2_FUNC_ANDN,
    OP_BCHG         = OP_BITOP + ROP2_FUNC_XOR,
    OP_BSET         = OP_BITOP + ROP2_FUNC_OR,
..
```

*Text 2définition des opcodes*

### 1.2) Format .bin, format .elf

2 types de binaires sont supportés par ces outils.

- Le format .bin est en fait le programme brut, le programme est censé est recopié intégralement et sans changements dans la mémoire à l'adresse 0.
- Le format .elf est un format de binaire plus évolué.

http://www.cs.ucdavis.edu/~haungs/paper/node10.html:

The executable and linking format (ELF) was originally developed by Unix System Laboratories and is rapidly becoming the standard in file formats[8]. The ELF standard is growing in popularity because it has greater power and

flexibility than the a.out and COFF binary formats[3]. ELF now appears as the default binary format on operating systems such as Linux, Solaris 2.x, and SVR4. Some of the capabilities of ELF are dynamic linking, dynamic loading, imposing runtime control on a program, and an improved method for creating shared libraries[3]. The ELF representation of control data in an object file is platform independent, an additional improvement over previous binary formats. The ELF representation permits object files to be identified, parsed, and interpreted similarly, making the ELF object files compatible across multiple platforms and architectures of different size.

The three main types of ELF files are executable, relocatable, and shared object files. These file types hold the code, data, and information about the program that the operating system and/or link editor need to perform the appropriate actions on these files. The three types of files are summarized as follows:

- An *executable file* supplies information necessary for the operating system to create a process image suitable for executing the code and accessing the data contained within the file.
- A *relocatable file* describes how it should be linked with other object files to create an executable file or shared library.
- A *shared object file* contains information needed in both static and dynamic linking.

In the next section we overview the ELF file format including a detailed description of each of the five section types that an ELF file might include. These five types are (1) the ELF header, (2) the program header table, (3) the section header table, (4) the ELF sections, and (5) the ELF segments. In Section 2.4.2, we describe the representation of data in an ELF file. The interested reader may consult reference [8] for additional information about the ELF format.

En pratique, 2 émulateur sont fournis pour chaque format de binaire. Les 2 émulateurs ne diffèrent pas que par leur mode de chargement du binaire. L'émulateur elf est beaucoup plus complet, et gère par exemple le MMU ( le format bin ne définit pas de toute façon les limites du programme)

# *2) Fonctionnement de l'émulateur.*

Les sources de l'émulateur sont stoquées dans le répertoire emu.

## 2.1) Les différents fichiers

- Syscalls.[c|h|def]: contient les codes pour chaque syscall supporté par l'émulateur. Les syscalls sont les appels système émulés. Les syscalls de linux (POSIX) sont partiellement gérés. Cela permet de lancer des programmes en s'appuyant sur le système hote. On n'as ainsi pas à charger un OS complet. Ces fichiers contiennent une astuce C amusante: on inclut 2 fois le fichier *syscalls.h*. Une fois pour déclarer et définir les fonctions, et une fois pour déclarer le tableau de pointeur vers ces fonctions.
- Emu.h: Contients les structures de données et des macros. Une étude plus poussée de ce fichier suit.
- Emumain.c: Partie spécifique à l'émulateur de . bin. Contient l'interprétation de la ligne de commande, le chargement du programme, la boucle principale, la gestion de la mémoire, la gestion de syscalls, la gestion des exceptions logicielles. 2 syscalls sont gérés: read et write.
- Elfemu.c: Même chose, mais pour le format elf. Ce format permet d'être plus précis dans l'émulation: gestion de la MMU (protection de la moire), -> plus d'Exception gérées. le systeme de syscalls.[h|c|def] est utilisé.
- Emu.c: Contient le core de l'émulation, le comportement de tous les opcodes est ici implémenté, ainsi que le décodage. Pour celui ci, on utilise une table triée qui contient une liste d'opcodes implémentés ( valeur, masque, implémentation). Il y a un index sur la valeur de l'opcode ( 8bits) et ensuite, on parcours par dichotomie les différentes version de l'instruction (flags différents).

## 2.2) Structures de données

L'émulateur manipule 3 données: La mémoire, les registres, les registres spéciaux.

### *2.2.1) Accès mémoire*

```
#define LOAD(sz,ignore) \
do { \
  char *p = memmap(r(R2).C(o,0), BYTES(sz), BYTES(sz), 0); \
  union reg r1 = reginit(R1); \
  unsigned j; \
  if (excode) return; \
  if (R3 && R2) r(R2).C(o,0) += r(R3).C(o,0); \
  for (j = 0; j < BYTES(sz); j++) \
    r1.C(b,j) = p[j]; \
  if (R1) r(R1) = r1; \
} while (0)
```

On voit que l'accès à la mémoire est géré par memmap qui renvoie un pointeur vers la zone mémoire sur laquelle on travaille. Il est possible, d'intégrer cela dans du systemC en remplaçant le

char * par un type spécial (genre MemPtr) qui sera simplement *typedef char * MemPtr,* dans la version C, mais qui sera une classe spéciale en simulation SystemC. Notre classe MemPtr devra redéfinir les opérateurs [] et * pour que la classe agisse comme si c'était un *char* *. Ainsi, le code de emu.c est entièrement utilisable moyennant un “rechercher remplacer”.

### 2.2.2) Accès registres

Les regitres sont définis par une grosse union:

```
/* a single register */
union reg {
        U8              b[CHUNKS(b)];
        U16             d[CHUNKS(d)];
        U32             q[CHUNKS(q)];
        U64             o[CHUNKS(o)];
        I8              sb[CHUNKS(b)];
        I16             sd[CHUNKS(d)];
        I32             sq[CHUNKS(q)];
        I64             so[CHUNKS(o)];
        float   F[CHUNKS(F)];
        double  D[CHUNKS(D)];
};

/* the register set */
extern struct regs {
        union reg       r[64];
        union reg       r_pc;
} regs;

/* shortcuts */
#define r(x)
regs.r[x]
#define for_all_chunks(i,sz)  for ((i) = 0; (i)
< CHUNKS(sz); (i)++)
```

Ainsi on peut considérer le registre comme un ensemble de mots de 8 bits ou un ensemble de mots de 16 bits, etc. Le banc de registre est donc très simplement un tableau de 64 registres. A noter que dans tout le code emu.c, la macro r est utilisée, pas d'accès direct par regs.r[]. On note aussi que le nombre de registre est codé en dur, par contre, la taille des registre est configurable. La macro CHUNKS est chargée de calculer le nombre de mots qui tiennent dans le registre.

### 2.2.3) Registres spéciaux

Les registres spéciaux contiennent une variété d'informations sur le contexte. Il y a des permissions d'accès.

```
/* special registers */
enum {
        SR_NUMBERS,
        SR_FAMILY,
        SR_STEPPING,
        SR_MAX_SIZE,
        SR_SIZE_0,
        SR_SIZE_1,
        SR_SIZE_2,
        SR_SIZE_3,
        SR_MAX_CHUNK_SIZE,
        SR_CYCLE,
        SR_PAGING,
        SR_CONTROL,
        SR_IRQ_BASE,
        SR_IRQ_SIZE,
        SR_TRAP_BASE,
        SR_TRAP_SIZE,
        SR_SYSCALL_BASE,
        SR_SYSCALL_SIZE,
        SR_TLBMISS_BASE,
        SR_URL,
        SR_URL_last = SR_URL + 7,
        SR_LAST_SR
};

#define _SR_RD (1u << 0)
#define _SR_WR (1u << 1)
#define _SR_RW (_SR_RD|_SR_WR)

extern struct sreg {
        U32 p_super;
        U32 p_user;
        union {
                U64 v;          /* XXX: use
`union reg' instead?! */
                char s[8];
        } u;
} sregs[];
```

### 2.2.4) Exceptions

Les execptions sont levées grace à la macro ex (EX). Elle pourra facilement être redéfinie pour lever un évènement systemC. On parle ici d'exceptions Logicielles. Je n'ai pas vu de trace de gestion d'exceptions matérielles dans le code.