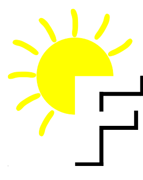
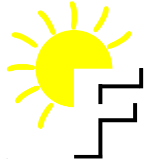


F-CPU: Year 4

Bail Cedric

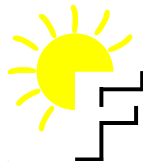
Boulay Nicolas

Yann Guidon

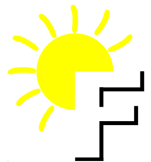


Plan

- F-CPU 4 dummies
- A simple SIMD character comparison
- Another example : arbitrary byte shuffling in one byte
- The hardware design flow
- TCPA
- Design
- Call convention

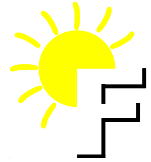


F-CPU 4 dummies
Yann Guidon



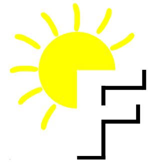
Introduction

- Goal : to design a microprocessor that can be used and modified by anyone without industrial pressure
- <RMS_beard=on> It’s all about freedom : This is ‘Freedom CPU’, not ‘Free CPU’
- ‘Year 4’ means 4th presentation to CCC and 4th year of existence



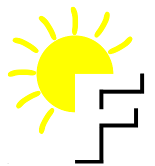
Architecture

- F-CPU is designed ‘from scratch’ and is not compatible with existing computers
- The architecture is aimed at high efficiency for computation intensive software
- RISC features and methods
  - Fixed-size 32 bits instructions
  - 64 x 64 bits registers
  - Load-store architecture
  - No stack
  - Register #0 is hardwired to 0
  - Conditional move and jump/call/return



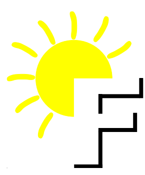
Data types

- Beware ! a register is not equivalent to a number !
- Registers are ‘at least’ 64-bit wide
- Registers can have more than 64 bits !
- It is simpler and more efficient to enlarge the registers than to decode more instructions per cycle (decoding and control logic would explode
- Register sizes can be any power of 2 : 128, 256, 512, or even 32768 bits (in theory)

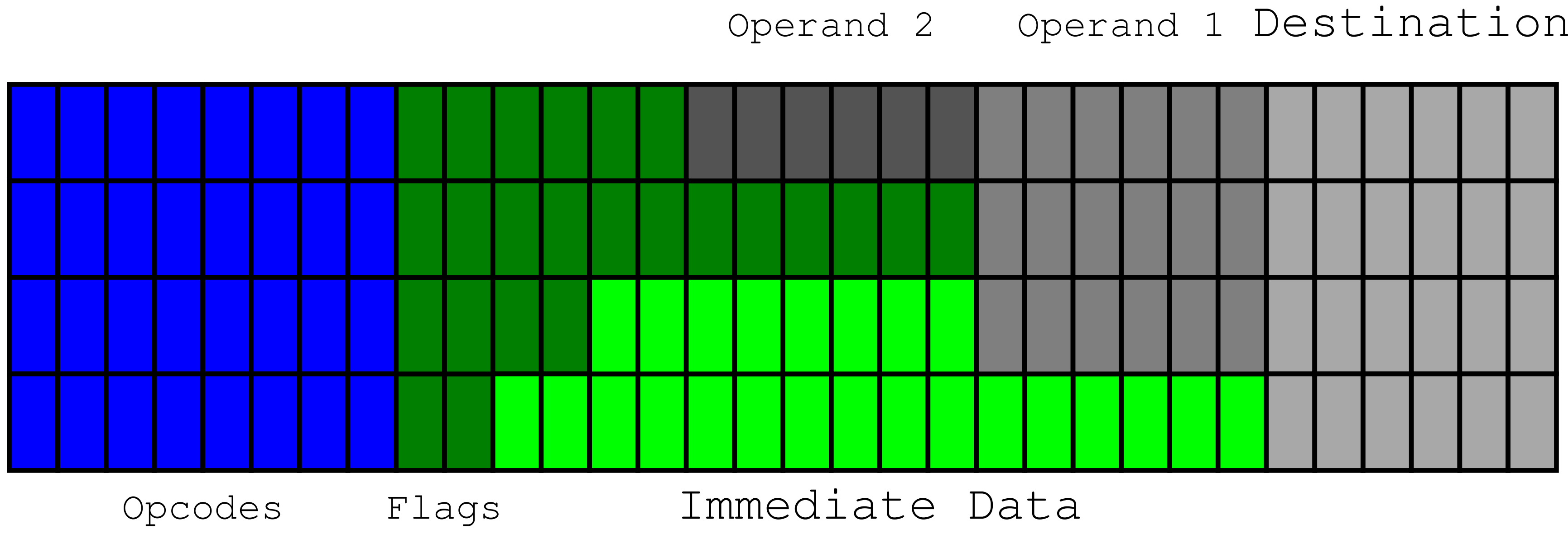


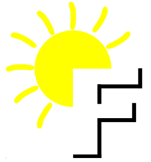
Data types (2)

- scalar data : aligned to the LSB, all MSB are cleared
  - 8, 16, 32 and 64 bit integers are supported
- pointers : like scalar data but the number of valid LSB is not known (depends on the implementation, could be 30 or 50)
- SIMD data : 2**N scalar data
  - 8x8, 4x16 and 2x32 bit integers are supported for 64 bit implementations



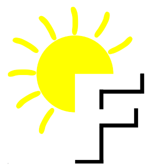
Instruction Format





FC0

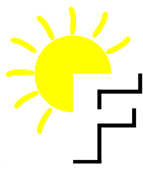
- 1st implementation: FC0
  - Statically scheduled (scoreboard-based)
  - Single-issue core
  - Out Of Order Completion
  - Many “Execution units” around a “Crossbar”
  - “Carpaccio” pipeline stages for higher frequency



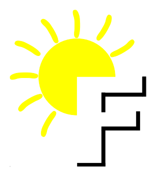
Ongoing work

(this is not complete or exhaustive)

- VHDL model
- C model
- Manual
- Boot monitor
- Gcc port
- Assembler
- Linker
- L4
- Linux



Simple SIMD character comparison



The ROP2 (logic) unit

rop2_unit.vhdl

3-level signal amplification tree (1->4->16->64) performed by fanout_tree

S S S S S S S S

partial_OR

partial_AND

partial_MUX

partial_ROP

A B C A B C A B C A B C A B C A B C A B C A B C

ROP2_mode

The fanout is higher than that : 16 for the 64-bit version. fanout_tree is used to compensate for this.

ROP2_function_bit3 2 1 0

rop2_xbar.vhdl

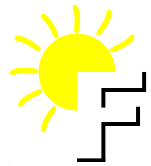
F-CPU Design Team
ROP2 unit : schematic view for one byte
(C) Yann Guidon 8/31/2001
version : dec. 2, 2001

This is only an indication of the equation complexity. The circuit will be synthesised from the parametised LUT.

LUT

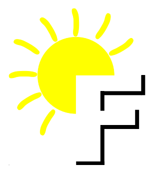
2 1 0

ROP2_function



C example

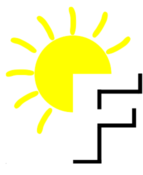
```
char a;

...

if (a == TAB || a == CR
    || a == ' ' || a == 0) {

    ...

}
```



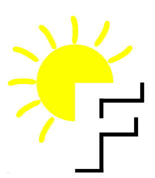
Assembler example

a in Ra, temporary result in Rtemp, mask in Rmask :

```
loadaddri end_if, Rjmp ; prefetch
sdup.8 Ra, Rtemp ; duplicate a
loadcons[0] 0x2000, Rmask ; load constants
loadconsx[1] 0x090A, Rmask
xorn.and.32 Rmask, Rtemp, Rtemp
bnz Rtemp, Rjmp

...

end_if:
```



Arbitrary byte shuffling in one byte

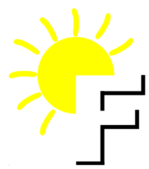


Random shuffling example

0 -> 3
1 -> 2
2 -> 4
3 -> 7
4 -> 5
5 -> 1
6 -> 0
7 -> 6

From this, we generate the following masks :

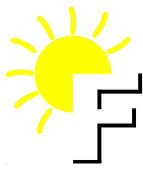
r3 = mask1 = 0x8040201008040201; // linear bit selection
r5 = maks2 = 0x4001028020100408; // permuted mask



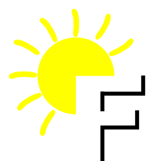
The assembly langage source

```
sdup.b r1, r2 ; duplicate r1 into r2
and.or r2, r3, r4 ; first mask and combine
and r4, r5, r6 ; second mask
shri 32, r6, r7 ; gather the bits in log2
or r6, r7
shri 16, r6, r7
or r6, r7
shri 8, r6, r7
or r6, r7
```

- 9 instructions for shuffling 8 bits : this yields almost 1 instruction per bit !



Powerup and BIST method



The FC0 pipeline

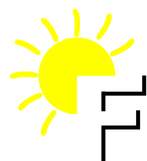
Register Set

ROP2

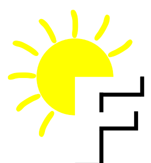
ASU

SHL

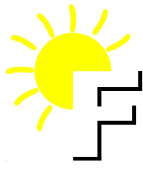
INC



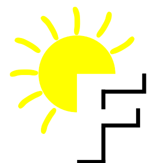
Popcount unit and LFSR



Popcount unit and LFSR



The hardware design flow
Nicolas Boulay



A transistor

Polysilizium

Oxid

Gate

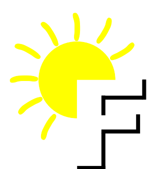
Drain

Source

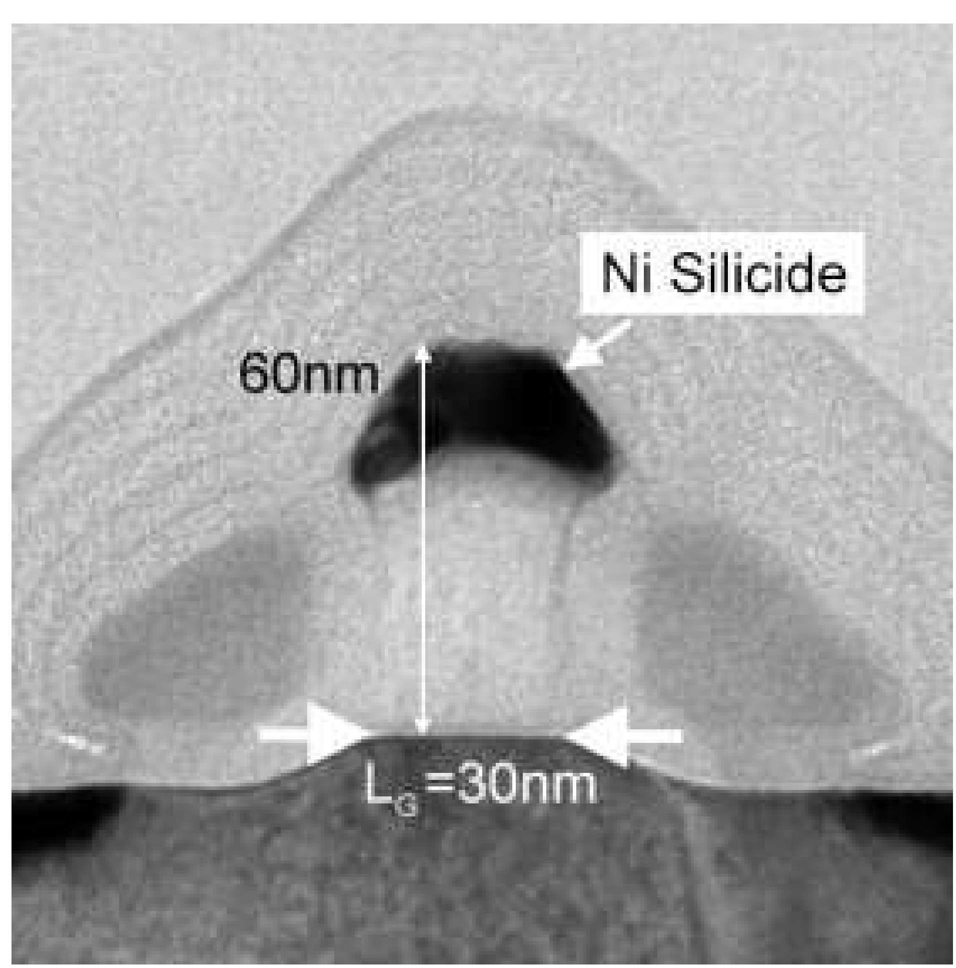
n^+

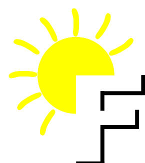
n^+

Silizium (p-dotiert)

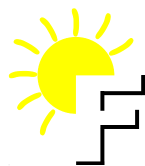


A real transistor

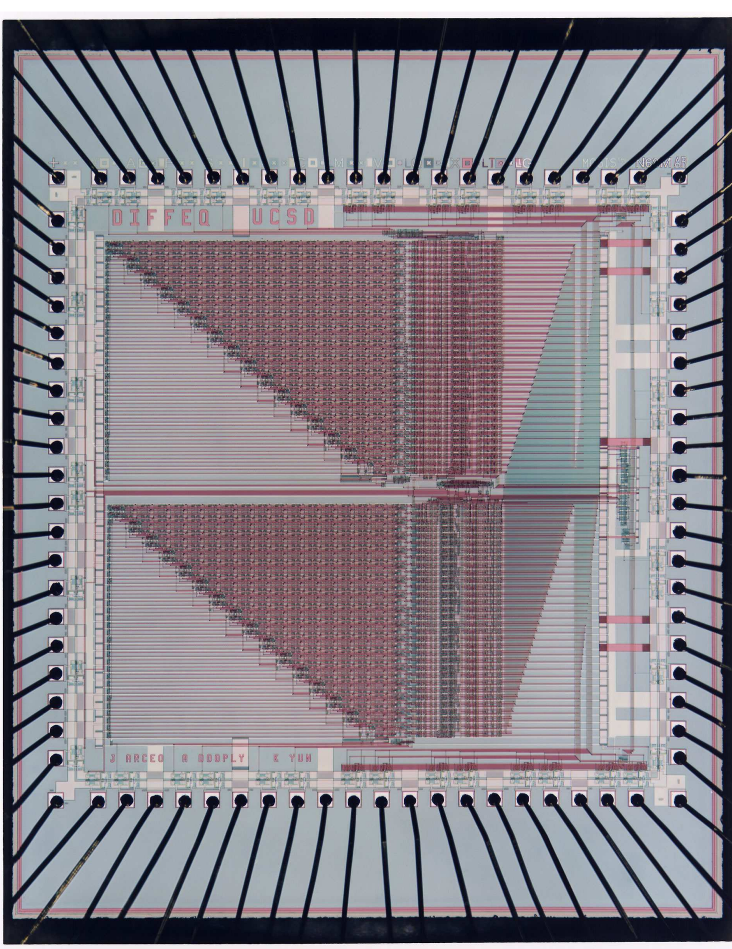


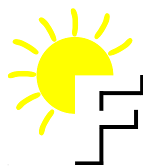


A wafer

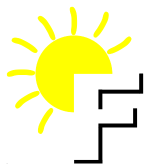


Some ASIC

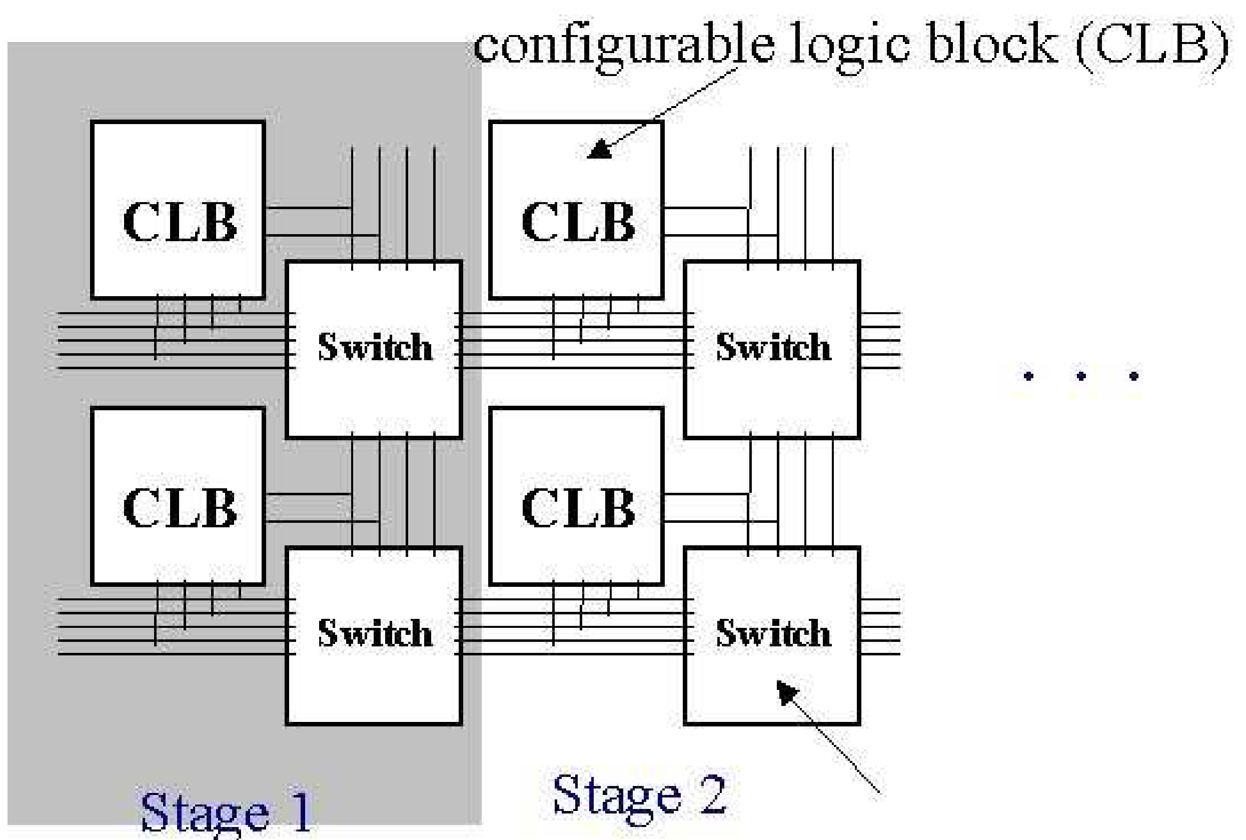


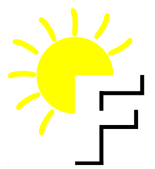


An other ASIC



FPGA principe

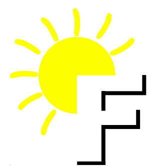




Making hardware

FPGA (field programable gate array)

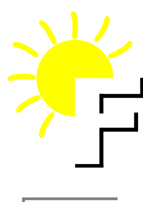
Semi-custom, full custom (ASIC, Application Specific Integrated Circuit).



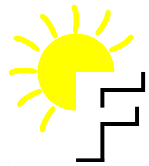
Design IP (or a core)

Nowdays what had been put in mainboard are put in the same die (piece of silicon). Componants are replace by core to create System-on-Chip (SoC).

F-cpu is a core. So a SoC could be maid of fritz chip + fcpu.

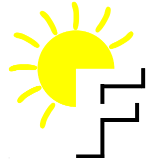


TCPA



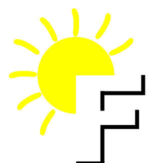
GPL

Depending of the licence, we could obliged to open all sources. But the cores risk to be not used (imagine that linux unallowed to run proprietary stuff). And seeing the code could not surely help to break the protection.



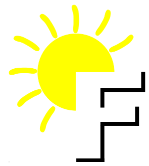
LGPL

Only the core is protected like the Leon is (Sparc V7 clone).



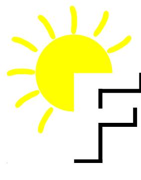
GPL+proprietary interface

Like linux kernel, we could choose to open certain interface (like the io bus but not the SDRAM bus).

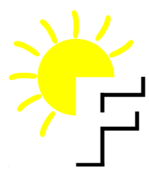


Licence

But the licence is a constant flameware on the mailing list. GPL is currently used, but is too much restrictive from my point of view. It's also hard to accept that GPL could cover hardware, too (something with sources and a "result").

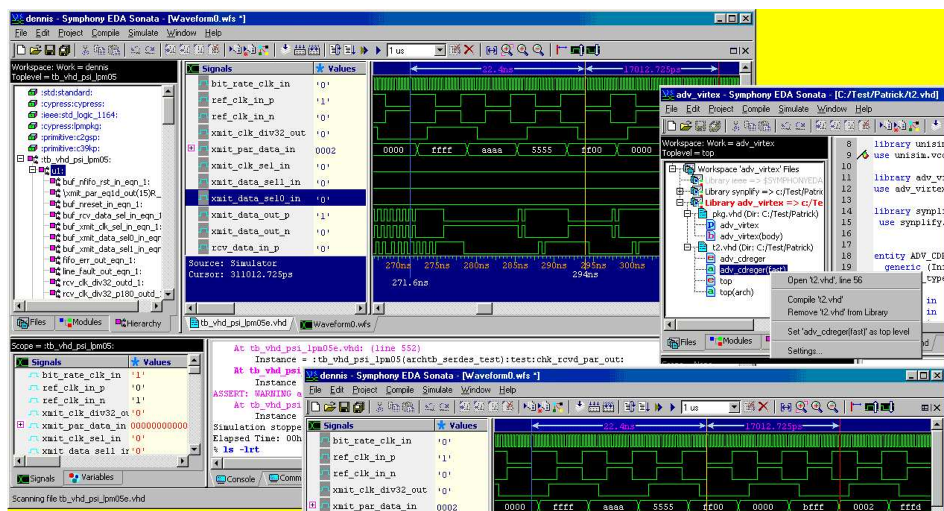


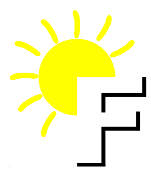
Design



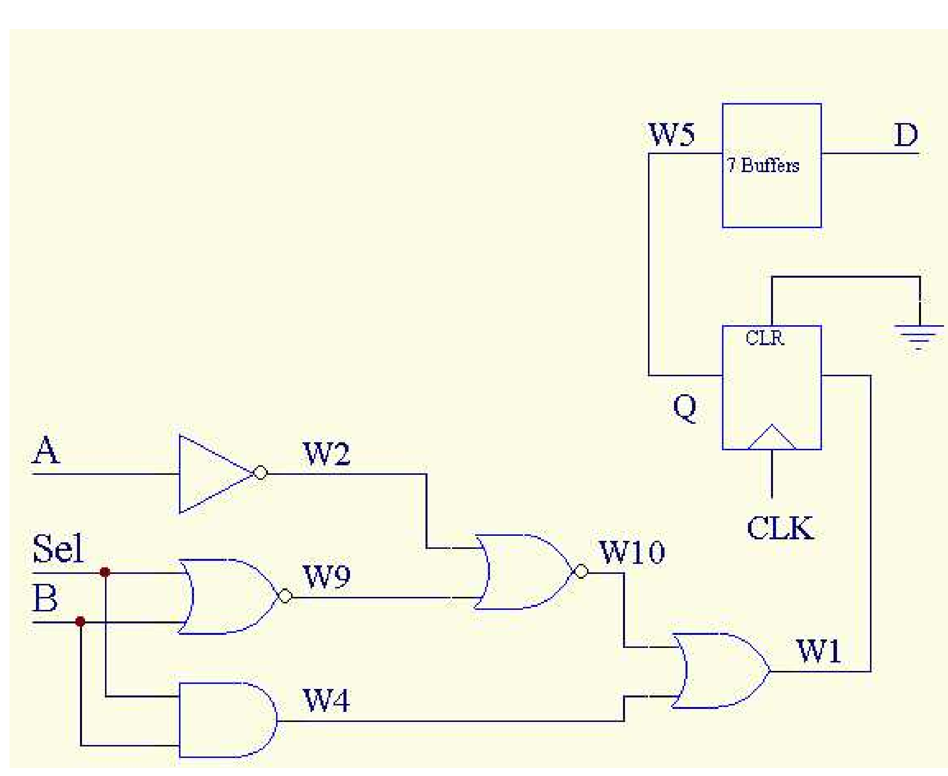
Design cycle

- Write HDL then Simulate RTL code (waveform)

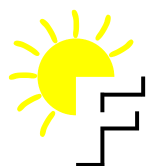




Design cycle

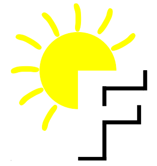


- Write HDL then Simulate RTL code (waveform)
- Synthesis it to have a netlist (timing result + number of gate used)



Design cycle

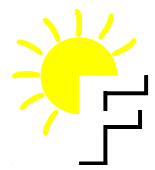
- Write HDL then Simulate RTL code (waveform)
- Synthesis it to have a netlist (timing result + number of gate used)
- Place and route to get plan (GDS2 files + more precise timing result + area used (wire))



Simulator

F-CPU sources are compatible with most compilers and have been tested with :

- ncsim (cadence, fastest of the market)
- modelsim
- Simili (freeware, slower that ncsim)
- ghdl (alpha version) (the story of a guy that wanted to learn Ada and VHDL so he wrote a VHDL gcc front end in Ada)
- ALDEC's Riviera (nice but proprietary)
- Vanilla VHDL (abandonware)

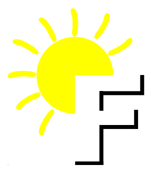


Synthetiser

Design Compiler (Synopsys, 100 Keur/year... for ASIC)

Synplify (Synplicity for FPGA)

_NO_ free software

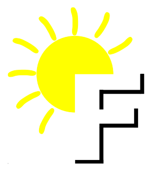


Place & Route

Cadence tools

Tendance of merged with synthesys tools (for <130 nm technology).

Also _NO_ free software



That's NOT all folks !

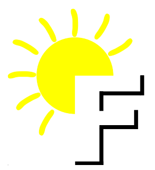
Static timing analysis tool to verify synthesis (primetime from synopsys : 100 Keur/year).

Equivalence checking between netlist and rtl code (avoid slooow simulation in gate level).

ATPG (automatic patern generator) to create input vectors to test the chip at the fab to cover the maximum stuck fault with the minimum of vectors.

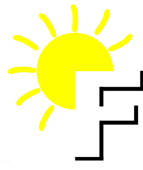
BIST generator to test memory.

Formal proofing tools to help finding bug in the rtl design.

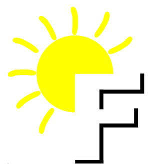


Tools conclusion

So it miss a lot of free tools !

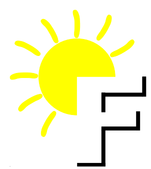


Call convention
Cedric Bail



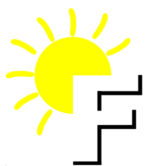
F-CPU call capacity

- No specialised register



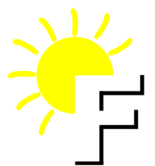
F-CPU call capacity

- No specialised register
  - No stack pointer



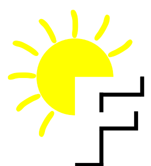
F-CPU call capacity

- No specialised register
  - No stack pointer
  - No specific address pointer



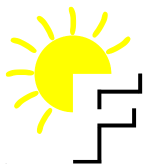
F-CPU call capacity

- No specialised register
  - No stack pointer
  - No specific address pointer
  - 63 Generals registers



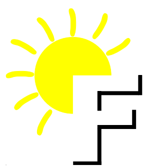
F-CPU call capacity

- No specialised register
  - No stack pointer
  - No specific address pointer
  - 63 Generals registers
- No call



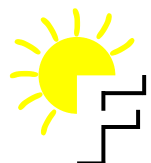
F-CPU call capacity

- No specialised register
  - No stack pointer
  - No specific address pointer
  - 63 Generals registers
- No call
- No stack



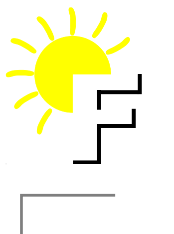
What we need to do a call

- Stack pointer
- Return address
- Return value
- Parameters



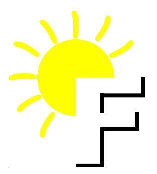
C source example

```
void hanoi(int N, char* D, char* B, char* I)
{
    if (N == 1)
        printf ("move %s to %s", D, B);
    else
       {
            hanoi (N-1, D, I, B);
            printf ("move %s to %s", D, B);
            hanoi (N-1, I, B, D);
       }
}
```



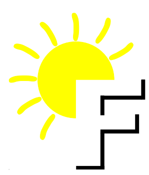
The first call convention

R0



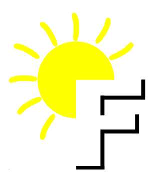
The first call convention

R0 = always zero



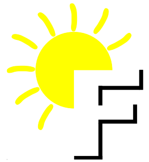
The first call convention

R0 = always zero
R1-R61
R62
R63



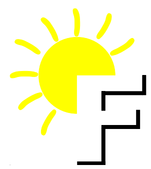
The first call convention

R0 = always zero
R1-R61 = preserved accross call
R62 = return address
R63 = stack pointer



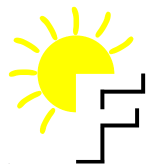
The cost

- Before using a register need to store it in memory
- Before doing a return you need to load them back from memory



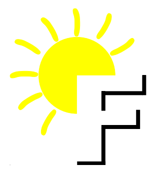
Prologue example

```
storei -8, [sp], r1
storei -8, [sp], r2
storei -8, [sp], r3
storei -8, [sp], r4
storei -8, [sp], r5
storei -8, [sp], r6
storei -8, [sp], r7
storei -8, [sp], r62
```



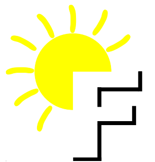
Prologue example

```
storei -8, [sp], r1
storei -8, [sp], r2
storei -8, [sp], r3
storei -8, [sp], r4
storei -8, [sp], r5
storei -8, [sp], r6
storei -8, [sp], r7
storei -8, [sp], r62

addi 6 * 8, sp, r1
loadi +8, [r1], r2 ; char* I
loadi +8, [r1], r3 ; char* B
loadi +8, [r1], r4 ; char* D
loadi +8, [r1], r5 ; int N
```



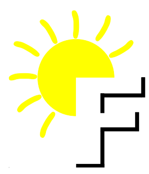
Epilogue example

```
loadi -8, [sp], r62
loadi -8, [sp], r7
loadi -8, [sp], r6
loadi -8, [sp], r5
loadi -8, [sp], r4
loadi -8, [sp], r3
loadi -8, [sp], r2
loadi -8, [sp], r1
```



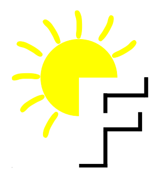
hanoi with first call convention

- 22 * 64 bits data are stored
- 20 * 64 bits data are loaded
- No tail recursive call



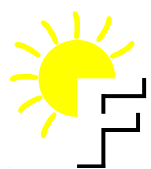
Second call convention

R1-R15 = Parameters
R16-R31 = Temporary (not preserved accross call)
R32-R61 = Saved temporary (preserved accross call)
R62 = Stack pointer
R63 = Return address



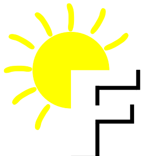
Prologue example

```
storei -8, [sp], r32
storei -8, [sp], r33
storei -8, [sp], r34
storei -8, [sp], r35
storei -8, [sp], r62
```



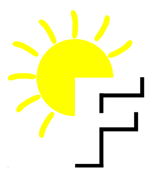
Epilogue example

```
loadi +8, [sp], r62
loadi +8, [sp], r35
loadi +8, [sp], r34
loadi +8, [sp], r33
loadi +8, [sp], r32
```



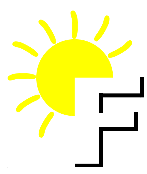
hanoi with second call convention

- 10 * 64 bits data are stored
- 10 * 64 bits data are loaded
- Tail recursive call
- Recursive prologue



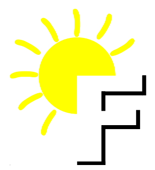
Recursive prologue example

```
storei -8, [sp], r36
storei -8, [sp], r37

loadcons printf, r36
loopentry r37


; Hanoi really start here
```



The maskload/store idea

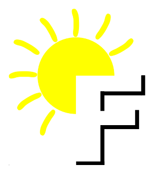
R1-R15 = Parameters
R16-R31 = Temporary (not preserved accross call)
R32-R57 = Saved temporary (preserved accross call)
R58 = Mask register
R59 = Pointer to Procedure Linkage Table
R60 = Pointer to Global Offset Table
R61 = Frame pointer
R62 = Stack pointer
R63 = Return address



Prologue example

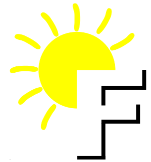
Will save r48-r52, mr (r58), sp (r62), ra (r63)
1100 0100 0001 1111

```
move r0, t2
loadcons.3 0xC82F, t2
and mr, t2, t3

maskstore t3, [sp]
move t2, r48
```



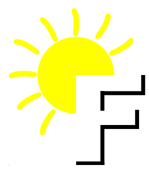
Epilogue example

```
maskload r48, [sp]
```



Problem

- Asynchronous
- Complex
- Faults
- Never the same binary with the same code



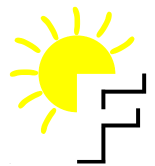
Solution

But we can do it with conditionnal load and store.

```
cstorel t3, [sp], r48
shiftli 1, t3, t3
msubi 8, sp, sp
```

The current accepted call convention

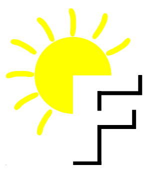
R1-R15 = Parameters
R16-R31 = Temporary (not preserved accross call)
R32-R58 = Saved temporary (preserved accross call)
R59 = Pointer to Procedure Linkage Table
R60 = Pointer to Global Offset Table
R61 = Frame pointer
R62 = Stack pointer
R63 = Return address



Linking solution

Use elf to put information on register used by function and call graph

- Clean address mode
- No hidden register
- Always the same result with the same code
- Always the best result for the binarie



Questions ?

Cedric BAIL : cedric.bail@free.fr
Nicolas BOULAY : nico@seul.org
Yann GUIDON : whygee@f-cpu.org