Système Digital : horloge virtuelle

Théophile Bastian, Noémie Cartier, Nathanaël Courant

26 janvier 2016

Vue d'ensemble

- Processeur généré en Python
- Netlist compilée vers C
- Sortie binaire envoyée sur une interface graphique
- Architecture inspirée d'ARM

Résultat :

- *Rapide* : $9.25 j.s^{-1}$; $11.5 j.s^{-1}$ en faisant fondre un processeur
- $1.6 MHz$; $2 MHz$ *EFFUP*

Compilateur de netlists

Fonctionnement :

- Nappes de fils : entiers 64 bits
- Tri topologique des équations
- Produire le code pour chaque équation

Optimisations :

- Opérations `SLICE`, `SELECT`, `CONCAT` quand on peut
- Opérations bit-à-bit sur les nappes si possible
- « – » unaire pour étendre un fil en une nappe ayant cette valeur

Opérations de l'assembleur

Les opérations supportées par notre processeur :

- opérations arithmétiques d'additions et de soustraction ;
- opérations binaires classiques ;
- déplacements d'une valeur ou de son opposé d'un registre vers un autre ;
- accès RAM (en lecture ou en écriture) ;
- saut vers un label `JMP`.

Chacune peut-être assortie d'une conditionnelle en fonctions des flags.

Les opcodes

Une opération assembleur $\rightsquigarrow$ un *opcode* (entier 64 bits)

Forme de l'opcode produit :

Bits	Longueur	Contenu
1 – 4	4	Condition d'exécution
5 – 8	4	Code de l'instruction
9	1	Écrire le résultat ?
10	1	Utiliser le *carry bit* ?
11	1	Écrire les flags ?
12 – 15	4	Registre de destination
16 – 19	4	op_1
20	1	Remplacer op_1 par $0 \cdots 0$?
21 – 46	25	op_2

Quartz

- *Pipe* sur le processeur
- Initialise au temps voulu
- Horloge synchrone : 1 une fois par seconde (incrément), 0 sinon
- Deux programmes distincts :
  - Synchrone : code python (plus simple, moins rapide)
  - Rapide : code C ; initialise, `while(1) { putchar(0); }`
- Gain Python $\rightarrow$ C : $\times 3$

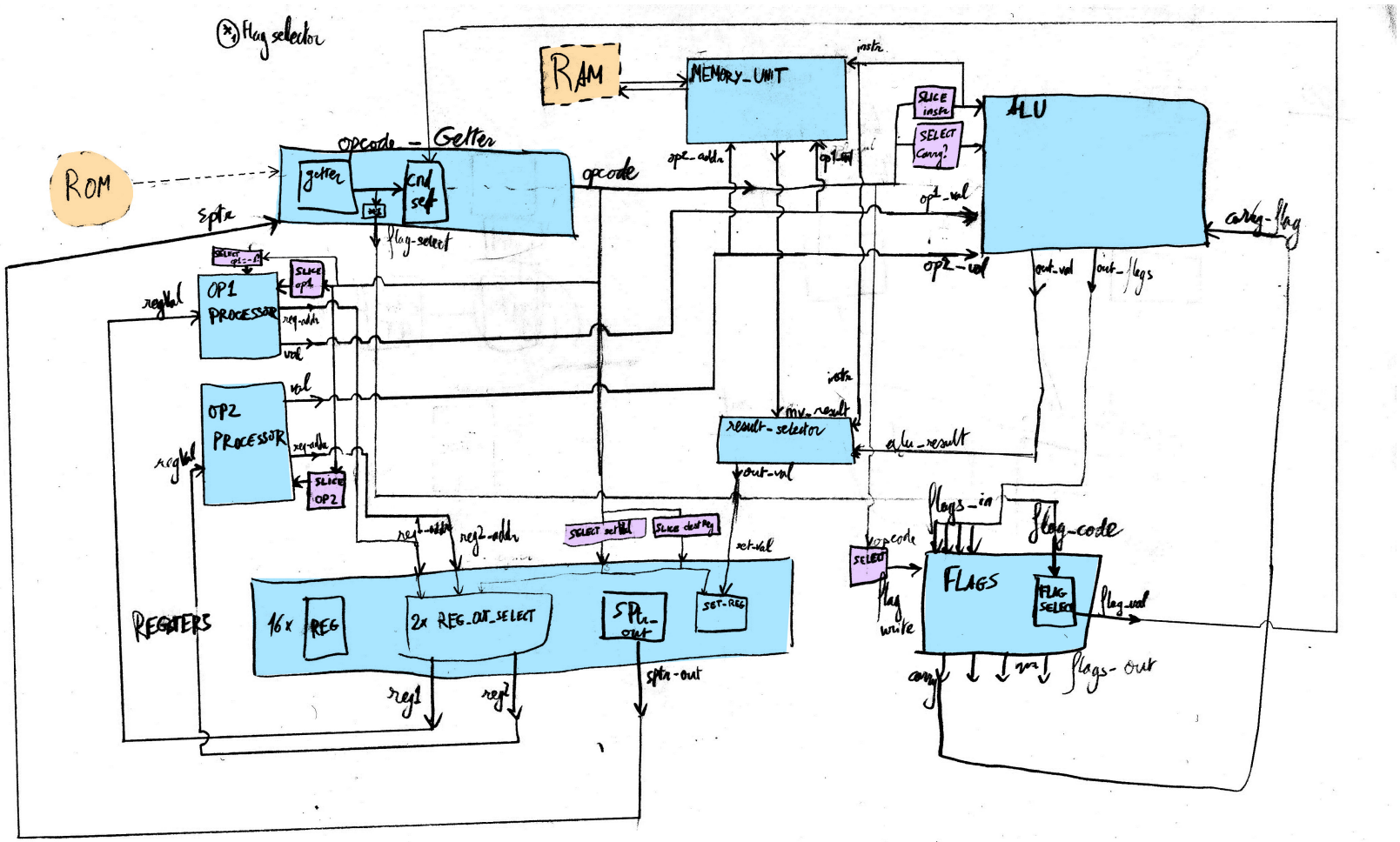
Programme

- Boucle des secondes déroulée
- Une instruction sur deux : mettre à jour l'affichage
- L'autre : calculer l'affichage de la minute suivante
- Exactement 2 instructions par seconde simulée (optimal)
- Gestion des années bissextiles, y compris exceptions
- Initialisation : difficile (début à une seconde quelconque)

Architecture du processeur

- basée sur ARM
- 16 registres
- registres spéciaux : `%r00`, `%r01`, `%r02`, `%r03`
- prend ses instructions en ROM
- RAM et ROM indexée par 16 bits
- 1699 portes logiques, $\sim$700 après optimisation

Schéma


Flag selector
RAM
MEMORY_UNIT
ROM
opcode_Getter
getter
cnd set
opcode
instr
flag_select
ALU
SLICE instr
SELECT Carry?
op1_addr
op1_val
op2_val
carry_flag
out_val
out_flags
OP1 PROCESSOR
OP2 PROCESSOR
reg_addr
val
SLICE op1
SLICE OP2
result_selector
mv_result
alu_result
out_val
REGISTERS
16x REG
2x REG_OUT_SELECT
SPtr_out
SET_REG
reg1_addr
reg2_addr
SELECT set_val
Slice destReg
set_val
reg1
reg2
sptr_out
flags_in
flag_code
SELECT
FLAGS
FLAG SELECT
flag_val
flag write
flags_out

Interface graphique

- C++, avec Qt
- Communication : lit 16 caractères par rafraîchissement (`stdin`)
- Un chiffre = un caractère ; un bit = un segment
- Affichage à 30 FPS

Optimisations :

- Deux threads : affichage, lecture de l'entrée
- Première idée : quand demandé, lire 16 caractères puis `fflush`
- Utilisé : lire en continu, donner les 16 derniers $\rightsquigarrow$ trop lent !
- Finalement, on ignore 12 cycles sur 13 ($\in \mathcal{P}$)

Conclusion

Merci de votre attention !