0101 - Calculs

Structures de contrôle, arithmétique, et qualité

Jean Privat

Université du Québec à Montréal

INF2171 - Organisation des ordinateurs et assembleur
v241

Rappels

Arithmétique de base

- Entiers signés et non signés (complément à deux)
- Addition et soustraction: `add`, `sub`...
- Débordements

Programmation assembleur

- Directives, instructions, registres, étiquettes
- Branchements conditionnels `b*` et inconditionnels `j*`

Plan

1. Structures de contrôle
2. Instructions logiques
3. Instructions de décalage
4. Multiplication et division
5. Développement et logiciels
6. Conclusion

Structures de contrôle

Structures de contrôles

Les langages procéduraux (et supérieurs)

Structures de contrôles explicites dans le langage

- `if(cond) {instrs}`
- `if(cond) {instrs} else {instrs}`
- `while(cond) {instrs}`
- `do {instr} while(cond)`
- `for(instr; cond; instr) {instrs}`
- Opérateurs booléens (`&&` `||` `!`)

En assembleur

- Branchement inconditionnel (`j`...)
- Branchement conditionnel (`beq`, `blt`...)

Goto vs. programmation structurée

Structures de contrôles

- Pour les humains
- Pour écrire des algorithmes
  (et des recettes de cuisine)

Gotos

- Pour les machines
- Pour écrire des livres dont vous êtres le héros

Programmer en assembleur

Principe

- Simuler les structures de contrôles

Comment faire

- Transformer les structures de contrôles en `j*` et `b*`
- Éviter d'utiliser des `j*` et `b*` autrement

Objectif

- Le code écrit doit être le plus linéaire possible
- Éviter le pire (code spaghetti)

Conditions

- Note: on *inverse* la condition

Condition if

```
      bne s1, s2, fin # if (s1 == s2) {
      nop             #    // corps du then
fin:                  # }
```

Condition if else

```
      bne s1, s2, else # if (s1 == s2) {
      nop              #    // corps du then
      j fin            # } else {
else: nop              #    // coprs du else
fin:                   # } // fin du if
```

Boucles while

Boucle while

```
loop: bge s1, s2, fin # while (s1 < s2) {
      nop             #   // corps du while
      j loop          # } // fin du while
fin:
```

Boucle do while

```
loop:                 # do {
    nop               #   // corps du do-while
    blt s1, s2, loop # } while (s1 < s2);
```

Boucle for

Boucle for

```
      li s0, 0            # variable de boucle i
      li s1, 100          # valeur maximum
loop: bge s0, s1, fin     # for(i=0; i<100; i++) {
      nop                 #   // corps du for
      addi s0, s0, 1      # // incrément du i
      j loop              # } // fin du for i
fin:
```

Opérations booléennes

En Java, C, C++, etc.

- `||` et `&&` sont paresseux (*lazy*)
- Exemple: `if(a && b) c;` est équivalent à
  `if(a) { if(b) { c; } }`

En Assembleur

- Pas d'instruction dédiée pour les Booléens
- On combine branchements conditionnels et inconditionnels

Exercice

Écrire un programme `compare.s`

- Demande (poliment) un nombre ; indiquer s'il est strictement négatif, compris entre 0 et 100 ou plus strictement grand que 100 ; puis dire au revoir.

Écrire un programme `spam.s`

- Écrire un programme qui lit un ligne et transforme les 'a' en '4', les 'e' en '3' et les 'i' en '1'
  (afin d'espérer déjouer les détecteurs de spam)

Instructions logiques

Opérations logiques (ou bit-à-bit) de base

- Non, *not* (~ en Java, C, C++...)
- Et, *and* (& en Java, C, C++...)
- Ou, *or* (| en Java, C, C++...)
- Ou exclusif, *xor* (^ en Java, C, C++...)

On traite les bits des registres un-à-un (*bitwise*)

Exemples 4 bits

a	1100
~a	0011

a	1100
b	1010
a&b	1000

a	1100
b	1010
a|b	1110

a	1100
b	1010
a^b	0110

Attention

- Ne pas confondre avec les **opérateurs booléens** !, && ou ||

Exercices

- `~0xEF01`, `0xEF01&0x7FAF`, `0xEF01|0x7FAF`, `0xEF01^0x7FAF`

Instructions logiques RISC-V

Type R (registre)

- `and rd, rs1, rs2`
- `or  rd, rs1, rs2`
- `xor rd, rs1, rs2`

Type I (immédiat)

- `andi rd, rs1, imm`
- `ori  rd, rs1, imm`
- `xori rd, rs1, imm`
- Attention: `imm` sur 12 bits (signé)
  -2^{11} à $2^{11} - 1$, soit -2048 à 2047
  Le signe de `imm` est **étendu** sur 32 ou 64 bits

Pseudoinstruction

- `not rd, rs1` équivalent à `xori rd, rs1, -1`

Exercice

Écrire un programme `capital.s`

- Qui transforme les minuscules en majuscules
- Les autres caractères restent inchangés
- ASCII 7 bits uniquement

Écrire un programme `capital2.s`

À la maison.

- Qui transforme les minuscules en majuscules
- Les majuscules en minuscules
- Les autres caractères restent inchangés
- ASCII 7 bits uniquement

Code ASCII (rappel)

Bits b_4	b_3	b_2	b_1	Row \ Column	0 (000)	1 (001)	2 (010)	3 (011)	4 (100)	5 (101)	6 (110)	7 (111)
0	0	0	0	0	NUL	DLE	SP	0	@	P	`	p
0	0	0	1	1	SOH	DC1	!	1	A	Q	a	q
0	0	1	0	2	STX	DC2	"	2	B	R	b	r
0	0	1	1	3	ETX	DC3	#	3	C	S	c	s
0	1	0	0	4	EOT	DC4	$	4	D	T	d	t
0	1	0	1	5	ENQ	NAK	%	5	E	U	e	u
0	1	1	0	6	ACK	SYN	&	6	F	V	f	v
0	1	1	1	7	BEL	ETB	'	7	G	W	g	w
1	0	0	0	8	BS	CAN	(	8	H	X	h	x
1	0	0	1	9	HT	EM	)	9	I	Y	i	y
1	0	1	0	10	LF	SUB	*	:	J	Z	j	z
1	0	1	1	11	VT	ESC	+	;	K	[	k	{
1	1	0	0	12	FF	FS	,	<	L	\	l	|
1	1	0	1	13	CR	GS	—	=	M	]	m	}
1	1	1	0	14	SO	RS	.	>	N	^	n	~
1	1	1	1	15	SI	US	/	?	O	_	o	DEL

Instructions de décalage

Décalage **logique à gauche**

- Décalent tout les bits d'un ou plusieurs crans à gauche
- Des bits sont **perdus** à gauche
- Des **zéros** s'insèrent à droite
- Opérateur << en Java, C, C++...

Exemples 8 bits

a	0	0	1	1	0	0	1	0
a<<1	0	1	1	0	0	1	0	0
a<<2	1	1	0	0	1	0	0	0
a<<3	1	0	0	1	0	0	0	0
a<<4	0	0	1	0	0	0	0	0

Décalage **logique à droite**

- Décalent tout les bits d'un ou plusieurs crans à droite
- Des bits sont **perdus** à droite
- Des **zéros** s'insèrent à gauche
- Opérateur >> en Java (et C, C++...)

Exemples 8 bits

a	0	0	1	1	0	0	1	0
a>>1	0	0	0	1	1	0	0	1
a>>2	0	0	0	0	1	1	0	0
a>>3	0	0	0	0	0	1	1	0
a>>4	0	0	0	0	0	0	1	1

Décalage **arithmétique à droite**

- Décalent tout les bits d'un ou plusieurs crans à droite
- Des bits sont **perdus** à droite
- Le bit de gauche est **répliqué**
- Opérateur >>> en Java (C et C++ utilisent aussi >>)

Exemples 8 bits

a	0	0	1	1	0	0	1	0
a>>>1	0	0	0	1	1	0	0	1
a>>>2	0	0	0	0	1	1	0	0
a>>>3	0	0	0	0	0	1	1	0

b	1	1	0	0	1	1	0	1
b>>>1	1	1	1	0	0	1	1	0
b>>>2	1	1	1	1	0	0	1	1
b>>>3	1	1	1	1	1	0	0	1

Logique ? Arithmétique ?

Décaler, c'est multiplier ou diviser

- En base 10: ajouter ou enlever à droite des 0 c'est multiplier ou diviser par 10, 100, 1000...
- En base 2: ajouter ou enlever à droite des 0 c'est multiplier ou diviser par 2, 4, 8...

Exemples 8 bits

a	0 0 1 1 0 0 1 0	→50
a>>1	0 0 0 1 1 0 0 1	→25
a>>2	0 0 0 0 1 1 0 0	→12
a<<1	0 1 1 0 0 1 0 0	→100

Débordement en décalant à gauche

- Une multiplication par deux, quatre, huit, etc. peut **déborder**

Exemples 8 bits non signé (de 0 à 255)

a	0 1 1 0 0 1 0 0	→	100
b=a<<1	1 1 0 0 1 0 0 0	→	200
b<<1	1 0 0 1 0 0 0 0	→	144

Exemples 8 bit signé (de -128 à +127)

a	0 1 1 0 0 1 0 0	→	100
b=a<<1	1 1 0 0 1 0 0 0	→	-56
b<<1	1 0 0 1 0 0 0 0	→	-112

Respect du signe en décalant à droite

Pour diviser par deux (ou plus)

- En non-signé: utiliser le **décalage logique** (>>)
- En signé: utiliser le **décalage arithmétique** (>>>)

La terminologie *logique* vs. *arithmétique* porte ici à confusion

Exemple 8 bits non signé (de 0 à 255)

a	1 1 1 1 0 0 0 0	→ 240
a>>1	0 1 1 1 1 0 0 0	→ 120
a>>>1	1 1 1 1 1 0 0 0	→ 248

Exemple 8 bits signé (de -128 à +127)

a	1 1 1 1 0 0 0 0	→ -16
a>>>1	1 1 1 1 1 0 0 0	→ -8
a>>1	0 1 1 1 1 0 0 0	→ 120

Instructions de décalage RISC-V

Type R (registre)

- `sll rd, rs1, rs2` (*shift left logical*)
- `srl rd, rs1, rs2` (*shift right logical*)
- `sra rd, rs1, rs2` (*shift right arithmetic*)
  - `rs1` est le registre à décaler
  - `rs2` est le décalage

Type I (immédiat), codage spécial

- `slli rd, rs1, imm` (*shift left logical immediate*)
- `srli rd, rs1, imm` (*shift right logical immediate*)
- `srai rd, rs1, imm` (*shift right arithmetic immediate*)
- Attention: `imm` sur 5 bits non signé
  de 0 à $2^5 - 1$, soit de 0 à 31

Exercice

Écrire un programme `decuple.s`

- Lire un nombre, le décupler, puis l'afficher
- Contrainte: utiliser `slli`

Écrire un programme `nbbits.s`

À la maison.

- Lire un nombre
- Afficher le nombre de bits à 1 dans sa représentation binaire
- Tester avec des nombre positifs et négatifs
- Tester avec RV32 et RV64

Multiplication et division

Multiplication

- Exercice: calculer 21×30 (décimal)

Multiplication

- Exercice: calculer 21×30 (décimal)
- Exercice: calculer 21×30 (8 bits non-signé)

Multiplication

- Exercice: calculer 21×30 (décimal)
- Exercice: calculer 21×30 (8 bits non-signé)

```
        00010101
       *00011110
        --------
        00000000
+      00010101
+     00010101
+    00010101
+   00010101
+  00000000
+ 00000000
+00000000
=000001001110110
```

Débordements

- Multiplier deux nombres de n bits donne un nombre de $2n$ bits

Débordements

- Pas de débordement possible si on a $2n$ bits pour le résultat

Multiplication signée

- On ne peut pas complètement **réutiliser** la micro-architecture
- On doit **distinguer** les opérations signées et non signées

Exemple 8 bits

255 (non signé) et −1 (signé) se codent pareil: 11111111

- $-1 \times -1 = 1$
  0000000000000001
- $-1 \times 255 = -255$
  1111111100000001
- $255 \times 255 = 65025$
  1111111000000001

Remarque

- Quelques soient les signes des opérandes,
- la moitié de **poids faible** du résultat reste **identique**
- 00000001 dans l'exemple

Multiplication en RISC-V

Extension "M" (*Integer Multiplication and Division*)

- Un registre insuffisant pour contenir le produit de deux registres
- Solution: deux opérations
  - Une pour les 32 bits de poids faible du résultat
  - Une pour les 32 bits de poids forts du résultat

Instructions `mul` (*multiply*, type R)

- `mul rd, rs1, rs2`
  La moitié faible de `rs1` $\times$ `rs2`
- `mulh rd, rs1, rs2` (*multiply high*)
  La moitié forte de `rs1` (signé) $\times$ `rs2` (signé)
- `mulhu rd, rs1, rs2` (*multiply high unsigned*)
  La moitié forte de `rs1` (non signé) $\times$ `rs2` (non signé)
- `mulhsu rd, rs1, rs2` (*multiply high signed unsigned*)
  La moitié forte de `rs1` (signé) $\times$ `rs2` (non signé)

Note: Pas de version immédiate ni de pseudoinstruction

Exercice

Écrire un programme `fact.s`

- Lire un nombre m
- Afficher les factorielles de 1 à m
- Rappel: $n! = 1 \times 2 \times ... \times (n-1) \times n$
- Note: utiliser seulement `mul` (ignorer `mulh*`).
- Question: pour quels n obtient-on un débordement?
  - En RV32 et en RV64?

Extension “M”

Pourquoi une extension

- Multiplication compliquée
  - Micro-architecture plus grosse/chère
- Pas nécessaire dans plusieurs situations
  - Donc exclu du socle de base RV32I
  - Mais présent dans le socle général RV32G (= RV32IMAFD)

Et sans l’extension “M” ?

- On programme des boucles
  $12 \times 4 = 12 + 12 + 12 + 12$
- On utilise des décalages si on peut
  Décalages plus performants que la multiplication
  On préfère les décalages même si “M” est présent

Division entière

Division entière non signée

- Diviser c'est encore plus compliqué
  - Microarchitecture encore plus grosse/chère
  - 2, 10, voire 20 fois plus lente qu'une addition
- Diviser deux nombres de n bits donne deux nombres de n bits
  - le **quotient** et le **reste**
  - $dividende = diviseur \times quotient + reste$
  - $reste < diviseur$

Exercice

- Calculer $235/11$ (décimal division posée à deux chiffres)

Division entière

Division entière non signée

- Diviser c'est encore plus compliqué
  - Microarchitecture encore plus grosse/chère
  - 2, 10, voire 20 fois plus lente qu'une addition
- Diviser deux nombres de n bits donne deux nombres de n bits
  - le **quotient** et le **reste**
  - $dividende = diviseur \times quotient + reste$
  - $reste < diviseur$

Exercice

- Calculer $235/11$ (décimal division posée à deux chiffres)
- Et en binaire ?

Division entière

Division entière non signée

- Diviser c'est encore plus compliqué
  - Microarchitecture encore plus grosse/chère
  - 2, 10, voire 20 fois plus lente qu'une addition
- Diviser deux nombres de n bits donne deux nombres de n bits
  - le **quotient** et le **reste**
  - $dividende = diviseur \times quotient + reste$
  - $reste < diviseur$

Exercice

- Calculer $235/11$ (décimal division posée à deux chiffres)
- Et en binaire ? Faites-le à la main à maison...

Division signée

Moins naturelle

- $7/3 \rightarrow 2$ reste 1, facile
- $-7/3$? $7/-3$? $-7/-3$? Cela demande réflexion

Plusieurs **conventions** possibles

- Quotient arrondi vers 0
  - $quotient = trunc(dividende/diviseur)$
  - Le reste et le dividende ont le même signe
  - x86, ARM, RISC-V, Java, JavaScript, C, C++...
- Quotient arrondi vers $-\infty$
  - $quotient = floor(dividende/diviseur)$
  - Le reste et le diviseur ont le même signe
  - Python, Ruby, Excel...
- Mathématique classique
  - Le reste est toujours positif

Division par zéro et débordement

Division par zéro

- Opération non définie
- Mème classique...

Débordement

- Un seul cas possible: calcul de l'opposé
  - $-2^{n-1}/-1$ avec n bits signé
- Exemple 8 bits signé (de -128 à 127)
  - $-128/-1 \rightarrow 128$ reste 0
  - -128 et -1 sont représentables
  - or, 128 ne l'est pas

Division en RISC-V

Instructions `div` et `rem` (type R)

Quotient arrondi vers 0

- `div rd, rs1, rs2` (*divide*)
- `divu rd, rs1, rs2` (*divide unsigned*)
- `rem rd, rs1, rs2` (*remainder*)
- `remu rd, rs1, rs2` (*remainder unsigned*)

Pas de version immédiate ni de pseudoinstruction

Division par zéro et débordement

Valeurs de retour particulières

- `div` par 0: tous les bits de `rd` à 1
- `rem` par 0: `rd` $=$ `rs1`
- `div` débordement: `rd` $=$ `rs1` (-2^{n-1})
- `rem` débordement: 0

Le programmeur a la responsabilité de tester

Exercices

Écrire un programme `facteur.s`

- Lire un nombre et afficher ses facteurs premiers
- Pseudocode:

```
facteurs(n) {
  f = 2;
  while (n > 1) {
    if (n % f == 0) {
      print(f);
      n = n / f;
    } else {
      f++;
  }
```

Exercices (à la maison)

Écrire un programme `fizzbuzz.s`

- Lire un nombre m
- Pour chaque nombre n de 1 à m (inclus)
- Affiche une ligne `Fizz` si n est divisible par 3
- Affiche une ligne `Buzz` si n est divisible par 5
- Affiche à la place un ligne `FizzBuzz` si n est divisible par 3 et par 5
- Autrement, affiche une ligne avec seulement le nombre n
- Exercice classique d'entretien d'embauche

```
1 2 Fizz 4 Buzz Fizz 7 8 Fizz Buzz...
```

- Solution 100% Java

Exercice: Crible d'Ératosthène (276-194 av. J-C.)

Écrier un programme `eratosthene.s`

- Calculer et affiche les nombre premiers inférieurs à 4096
- Afficher le nombre de nombres premiers affichés (564)
- Pseudocode:

```
limite = 4096;
tab = tableau de 4096 octets initialisés à 0;
n = 0; // nombre de nombres premiers
for (i=2; i<limite; i++) // Pour chaque nombre
  if (tab[i] == 0) { // Si pas éliminé
    print(i);
    n++;
    for (j=i*i; j<limite; j=j+i)
      tab[j] = 1; // Éliminer les multiples
  }
print(n);
```

Exercice: Crible d'Ératosthène

Refaire le programme `eratosthene2.s`

- Utiliser un tableau huit fois plus petit (512o)
- Marquer un entier par bit au lieu d'un entier par octet.
- Utilisez les instructions logiques et de décalage pour accéder individuellement aux bits

Développement et logiciels

Niveaux d'abstraction des langages

Programmation en langage machine

- On se met exactement au niveau de la machine
- Pas beaucoup d'intérêt en soit
  - Sauf cas très particulier
  - Rétro-ingénierie binaire (voir INF600C)

Programmation en assembleur

- On facilite la mise au niveau de la machine
- On exhibe et manipule **directement** des concepts architecturaux
- C'est l'**intérêt premier** du cours INF2171

Niveaux d'abstraction des langages

Programmation en langage de plus haut niveau

- On s'éloigne du niveau de la machine
- On s'approche d'un niveau plus humain
- Objectifs d'ingénierie logicielle
  - Développer mieux de meilleurs logiciels

Corollaire de la programmation assembleur

- Développer moins bien des logiciel pires
- Travailler plus fort pour des résultats moins impressionnants
- C'est la **difficulté première** du cours INF2171

Qualité logicielle: ISO/IEC 25010 (2011)

- Aptitude fonctionnelle: complet, correct, approprié
- Performance et efficience: processeur, mémoire, énergie, etc.
- Compatibilité: interopérabilité et coexistence
- Utilisabilité: accessibilité, ergonomie, etc.
- Fiabilité: maturité, disponibilité, robustesse, récupérabilité
- Sécurité: confidentialité, intégrité, etc.
- Maintenabilité: modularité, réutilisabilité, compréhensibilité, modifiabilité, testabilité
- Portabilité: adaptabilité, instabilité, remplaçabilité

Les détails dans vos cours de génie logiciel...

Assembleur en général et INF2171 en particulier

- Est-ce que tout s'applique?
- Quoi privilégier?
- Exercice: faire un top 4

Qualité logicielle assembleur

Fonctionnalité + Fiabilité

- Le programme fait ce qu'il dit
- Complet, sans bogues, robuste

Compréhensibilité (et testabilité)

- Le programme dit ce qu'il fait
- Est lisible et compréhensible par un informaticien humain
- Même à 3 heures du matin

Performance

- S'il vous reste du temps de développement

Le reste des critères (et sous critères)

- Ne sont pas pertinent pour nous
- Ou concernent la spécification logicielle (cahier des charges)
  Et non l'implémentation

Stratégie générale de développement assembleur

- Itératif et incrémental
- Une étape à la fois
  - Valider et tester à chaque étape
  - Limiter la dette technique
  - Ne pas briser l'existant (monotonie)
- Penser autant à la machine qu'à l'humain
  - Votre binôme de TP
  - La version future de vous-même
  - Le correcteur (qui vous note!)

Remarque

- C'est exactement ce que l'on fait quand on programme en classe

Bogues

- *If debugging is the process of removing software bugs, then programming must be the process of putting them in.*
  — Edsger W. Dijkstra (attribué)
- *Everyone knows that debugging is twice as hard as writing a program in the first place. So if you're as clever as you can be when you write it, how will you ever debug it?*
  — Brian W. Kernighan 1974 (loi de Kernighan)
- *There are two ways of constructing a software design. One way is to make it so simple that there are obviously no deficiencies. And the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult.*
  — Tony Hoare 1981
- *The only difference between a bug and a feature is the documentation.* — Anonyme (classique)
- *Always code as if the guy who ends up maintaining your code will be a violent psychopath who knows where you live.*

Signaler un bug à un programmeur

Règles

- Dire précisément ce qui ne fonctionne pas
- Dire précisément ce qui était attendu
- Dire précisément comment arriver à ce résultat incorrect (utilisation pas-à-pas)
- Décrire les symptômes (et non proposer un diagnostic)
- http://www.chiark.greenend.org.uk/~sgtatham/bugs-fr.html

Lisibilité au fur et à mesure

Commenter

- Expliquer ce qui est fait (et non comment c'est fait)
- Expliquer qui s'en sert (et/ou comment s'en servir)
- $\neq$ Traduire ou paraphraser l'assembleur

Utiliser des identifiants littéraux sémantiques

- Symboles et étiquettes: le nom d'une étiquette correspond aux instructions et donnés qui suivent
- Utiliser les noms d'ABI des registres (et les respecter)

Lisibilité au fur et à mesure

Structurer

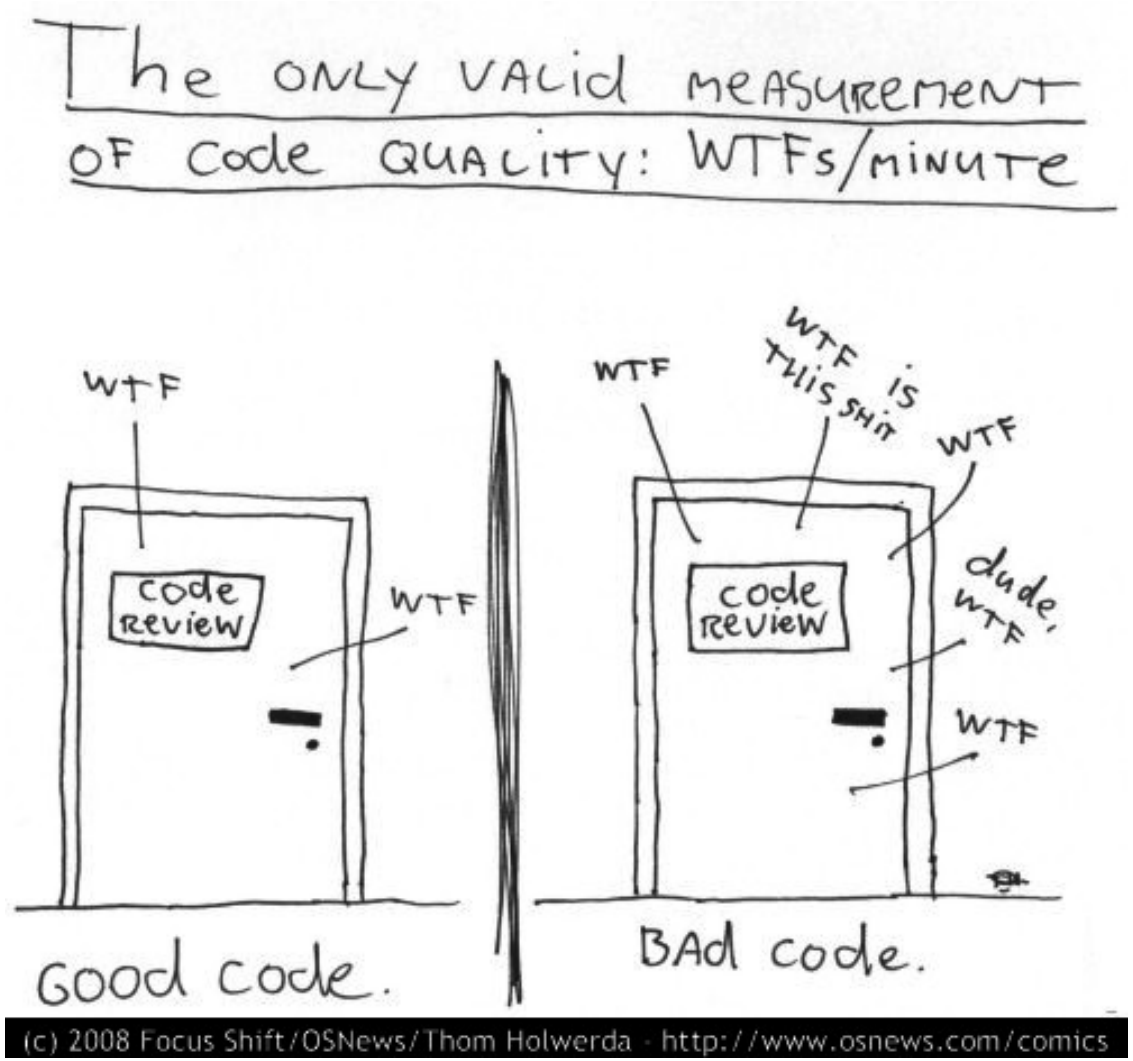
- Les morceaux identifiés et aux bons endroits
- Les liens entre morceaux: explicites et nécessaires

Factoriser et simplifier

- Un code long et complexe est illisible
- Limiter les redondances (c'est pas toujours facile)
- Documenter les différences entre parties similaires

Risque: perdre le contrôle

- Tout à coup, on ne comprend plus rien
- Ni au code, ni au comportement observé



Optimisation

- *Premature optimization is the root of all evil*

  — Donald E. Knuth 1974

- *We follow two rules in the matter of optimization:*
  Rule 1: Don't do it.
  Rule 2 (for experts only). Don't do it yet - that is, not until you have a perfectly clear and unoptimized solution.

  — Michael A. Jackson 1975

Efficacité des programmes

Principe 1

- On n'optimise que les parties les plus exécutées (sur le chemin critique)

Corollaire

- Toute partie non critique d'un programme peut être désoptimisée

Principe 2

- L'optimisation ne doit pas nuire à la correction et à la clarté

Efficacité des programmes (suite)

Moyens

- Réduire le nombre d'instructions
  Remplacer par des instructions plus efficaces
- Changer les structures de données et les algorithmes

Risques

- Optimisation prématurée
  - Optimiser un programme incomplet ou bogué
- Pertes des autre qualités
  - Optimisation qui introduit des bogues
  - Optimisation qui rend le programme peu lisible

Conclusion

Résumé

Plus de structures de contrôles

- Boucles et conditions

Plus d'instructions arithmétiques

- Instructions: `and`, `andi`, `or`, `ori`, `xor`, `xori`, `sll`, `slli`, `srl`, `srli`, `sra`, `srai`, `mul`, `mulh`, `mulhu`, `mulhsu`, `div`, `divu`, `rem`, `remu`
- Pseudoinstructions: `not`

Exercices

- `compare.s`, `spam.s`, `capital.s`, `decuple.s`, `nbbbits.s`, `fact.s`, `facteurs.s`, `eratosthene.s`

La prochaine fois

Adressage mémoire

- Tableaux, pointeurs et tas
- Bonus: routines