Numéro d'anonymat (donné sur votre étiquette)

Examen 2020 – 2021

Architecture des ordinateurs 1 – LU3IN029

Durée : 1h30

Documents autorisés : Aucun document ni machine électronique n'est autorisé à l'exception du mémento MIPS.

Le sujet comporte 18 pages. Ne pas désagrafer les feuilles. Répondre directement sur le sujet. Le barème indiqué pour chaque question n'est donné qu'à titre indicatif tout comme le barème total qui sera ramené à une note sur 20 points. Le poids relatif des exercices et des questions par contre ne changera pas.

L'examen est composé de 3 exercices indépendants.

- Exercice 1 - 12 points : Détection de débordement en logiciel – (p. 1)
- Exercice 2 - 22 points : Fonctions récursives – (p. 4)
- Exercice 3 - 12 points : Architecture et programmation système – (p. 4)

Exercice 1 : Détection de débordement en logiciel – 12 points

Le dépassement de capacité sur entiers relatifs lors d'une addition peut se détecter uniquement à partir des opérandes et du résultat : il y a un dépassement de capacité si et seulement si les opérandes sont de même signe et de signe différent du résultat.

Question 1.1 : 2 points

Réalisez les opérations ci-dessous (directement en hexadécimal), indiquez s'il y dépassement de capacité en justifiant uniquement par analyse du signe des opérandes et du résultat.

	0x	A	B	C	D	E	F	7	8
+	0x	A	B	C	D	E	F	7	8
	0x								

Depassement : Oui - Non
Justification :

	0x	7	F	F	F	F	F	F	E
+	0x	7	F	F	F	F	F	F	E
	0x								

Depassement : Oui - Non
Justification :

	0x	7	F	F	F	F	F	F	E
+	0x	8	F	F	F	F	F	F	E
	0x								

Depassement : Oui - Non
Justification :

Solution:

0xABCDEF78 + 0xABCDEF78 = 0x579BDEF0 + dépassement de capacité (2 entiers négatifs et résultat positif)

0x7FFFFFFE + 0x7FFFFFFE = 0xFFFFFFFC + dépassement de capacité (2 entiers positifs et un résultat négatif)

0x7FFFFFFE + 0x8FFFFFFE = 0x0FFFFFFC + pas de dépassement (2 entiers de signe différent)

Question 1.2 : 10 points

Un programme assembleur en MIPS n'a pas accès aux valeurs des drapeaux calculés par l'ALU lors d'une addition. Il est toutefois possible de détecter un dépassement de capacité en testant la condition de dépassement énoncée ci-dessus.

L'objectif de cet exercice est d'écrire un programme assembleur détectant les dépassements de capacité lors d'une addition entre deux entiers rangés en mémoire.

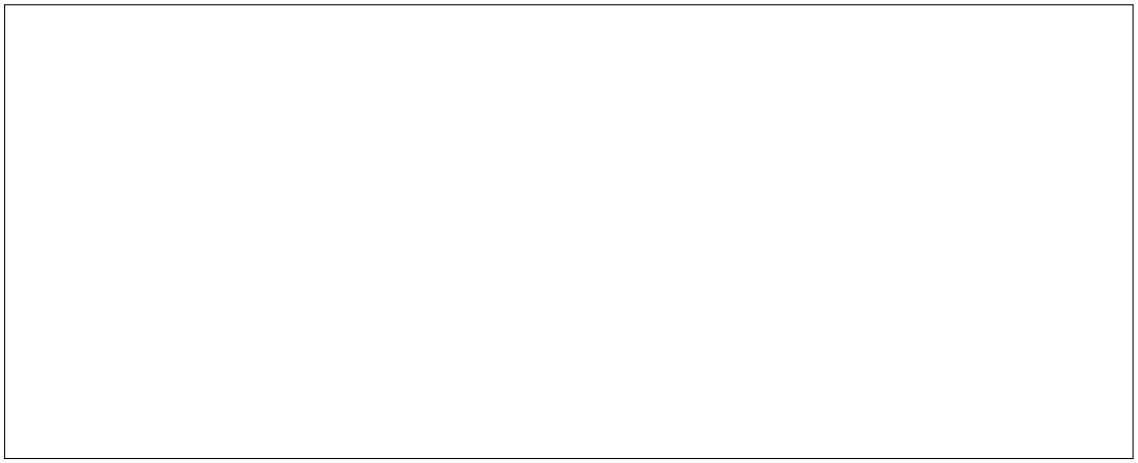
Donner un code assembleur complet qui :

- possède trois variables globales `n1`, `n2`, `n3` de type entier. `n1` et `n2` sont initialisées en donnant des valeurs exprimées en hexadécimal et **provoquant un débordement**. `n3` est initialisée à 0.
- comporte un programme principal qui :
  1. effectue l'addition entre les 2 entiers `n1` et `n2`,
  2. teste si le résultat est valide (c'est-à-dire si la condition de débordement ci-dessus n'est pas remplie),
  3. s'il est valide, range le résultat dans `n3` et l'affiche en hexadécimal (appel système 34 qui a les même arguments que celui d'affichage d'un entier en décimal),
  4. sinon affiche la chaine de caractères `"débordement"`,
  5. termine.

Solution:

```
 .data
n1: .word 0xABCDEFFF # 0x7FFFFFFE
n2: .word 0xABCDEFFF #1 # 0x7FFFFFFF
n3: .word 0

erreur: .ascii "débordement"

.text
  lui $8,   0x1001
  lw  $16,  0($8)
  lw  $17,  4($8)

  addu $18, $16, $17

  bgtz $17, op2_pos     # test et saut si n2 positif
  bgez $16, no_overflow # n2 négatif ou nul, test et saut si n1 positif ou nul
  blez $18, no_overflow # n1 et n2 négatifs, test et saut si resultat negatif ou nul
  j overflow            # debordement car résultat positif

op2_pos:                 # cas n2 positif
  blez $16, no_overflow  # test et saut si n1 négatif ou nul
  bgez $18, no_overflow  # n1 et n2 positifs, test et saut si resultat positif ou nul
overflow: # affichage chaine
  lui $4, 0x1001
  ori $4, $4, 0x000c
```

```
  ori $2, $0, 4
  syscall
  j fin
no_overflow: # affichage valeur hexa
  lui $8, 0x1001
  sw  $18, 8($8)
  ori $4, $18, 0
  ori $2, $0, 34
  syscall
fin:
  ori $2, $0, 10
  syscall
```



Exercice 2 : Fonction récursive – 22 points

Cet exercice concerne l'implémentation d'une fonction récursive calculant l'élément de rang `n` de la suite de Fibonnacci. Dans l'implémentation de la fonction récursive vue en TD, il y a de nombreux appels récursifs redondants effectués. L'implémentation considérée dans cet exercice, donnée ci-dessous, évite les suites identiques d'appels récursifs en mémorisant dans un tableau les valeurs déjà calculées. Ce tableau est passé en argument de la fonction.

```
int fib(int n, int tab[]){
  if (tab[n] != -1)
     return tab[n];

  if (n == 0 || n == 1) {
      tab[n] = 1;
      return tab[n];
  }

  tab[n] =  fib(n-1, tab) + fib(n-2, tab);
  return tab[n];
}
```

Le code C complet contenant la fonction `fib` a 2 variables globales :

```
int tab[256]; /* tableau de mémorisation des valeurs calculées */
int N = 256;  /* taille du tableau */
```

Question 2.1 : 1 points

Donner le contenu de la section `.data` du programme assembleur correspondant au programme C donné ci-dessus. Vous indiquerez les adresses des variables en commentaire.

Solution:

```
.data
tab:    .align 2
   .space 1024
N: .word 256
```

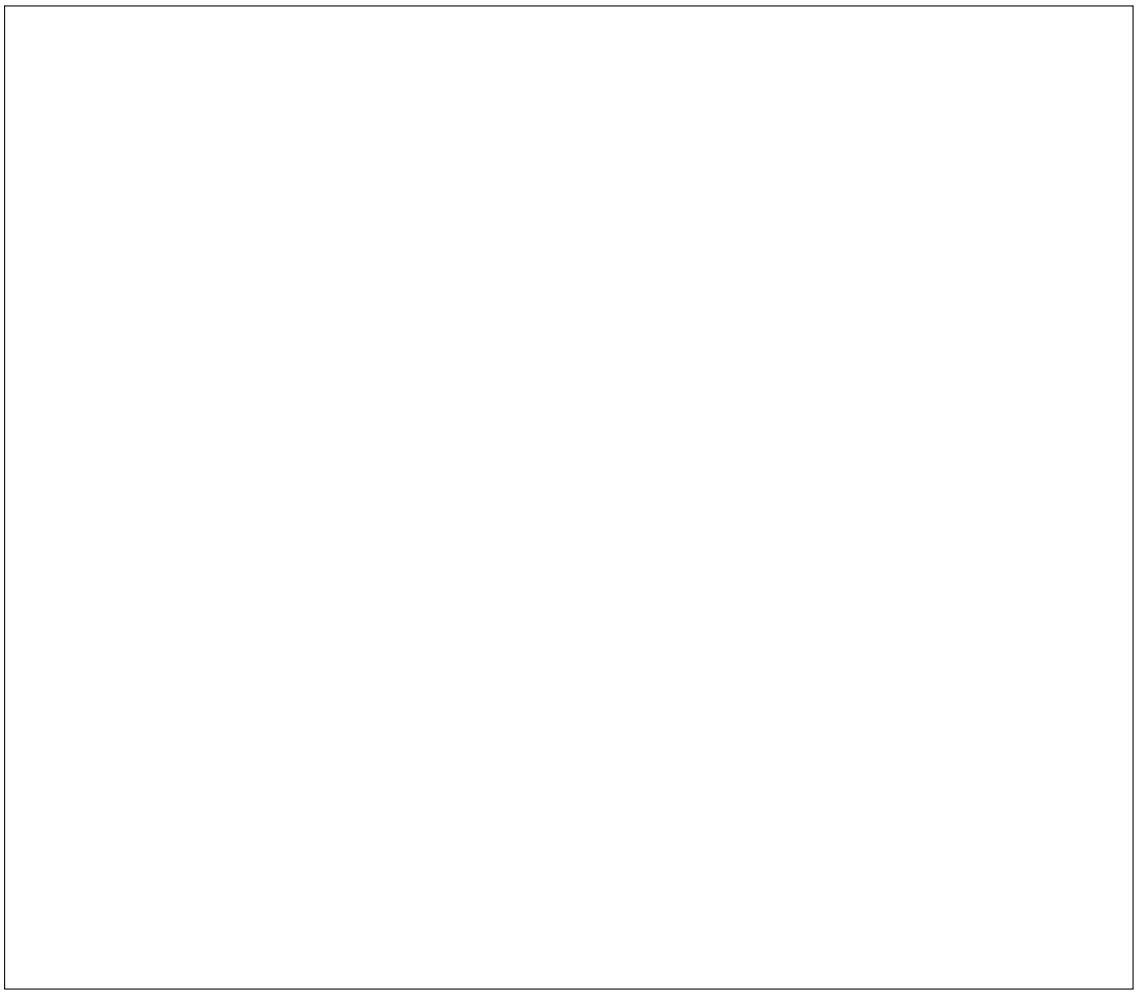
Le programme principal est le suivant :

```
int fib(int n, int tab[]);  /* signature de la fonction fib */
void main(){
  int i = 0;
  int n;

  while (i < N){  /* initialisation du tableau */
    tab[i] = -1;
    i++;
  }

  scanf("%d", &n);  /* lecture d'un entier mis dans n */
  n = n & 0xFF;    /* n doit etre entre 0 et 255 */

  printf("%d", fib(n, tab));
  exit();
}
```

Question 2.2 : 6 points

Donner le code assembleur correspondant au programme principal. Vous agrémenterez votre code de commentaires faisant le lien avec le code source. Vous justifierez notamment le nombre d'octets alloués sur la pile pour le contexte du `main`.

Votre code peut être optimisé (variables locales en registre, etc.).



Solution:

```
.text
   addiu $29, $29, -16    # nv = 2 + na = 2
   xor   $8,  $8,  $8     # i = 0
   lui   $9, 0x1001
   ori   $9, $9, 1024  # @N
   lw    $9, 0($9)
   lui   $10, 0x1001    # @tab[0]
   addiu $12, $0, -1

while_main:
   slt   $11,  $8, $9
   beq   $11,  $0, suite_main
   sll   $11,  $8, 2        # i * 4
   addu  $11,  $10, $11     # @tab[i]
   sw    $12, 0($11)        # tab[i] = -1
   addiu $8, $8, 1          # i++
   j while_main
suite_main:
   ori   $2, $0, 5
   syscall
   andi  $2, $2, 0xFF       # valeur de n optilisée dans $2  ici

   ori   $4, $2, 0          # n
   lui   $5, 0x1001         # tab (rechargement car syscall avant)
   jal   fib
```

```
   ori   $4, $2, 0
   ori   $2, $0, 1
   syscall

   addiu $29, $29, 16
   ori   $2, $0, 10
   syscall
```

Question 2.3 : 10 points

Donner le code assembleur correspondant à la fonction `fib` dont le code est donné ci-dessous. Vous agrémenterez votre code de commentaires faisant le lien avec le code source. Vous justifierez notamment dans vos commentaires la taille du contexte alloué pour la fonction.

Votre code peut être optimisé.

```
int fib(int n, int tab[]){
  if (tab[n] != -1)
     return tab[n];

  if (n == 0 || n == 1) {
      tab[n] = 1;
      return tab[n];
  }

  tab[n] =  fib(n-1, tab) + fib(n-2, tab);
  return tab[n];
}
```

Solution:

```
fib:
  addiu $29, $29, -20        # na = 2 + nv = 0 + nr = 2 + 31
  sw    $31,  16($29)
  sw    $16,  12($29)
  sw    $17,   8($29)
  sw    $4,   20($29)
  sw    $5,   24($29)

  sll   $16,  $4,  2       # n * 4
  addu  $16,  $16,  $5     # @tab[n] dans reg persistant
  lw    $8,   0($16)       # $8 <- tab[n]
  addiu $9,   $0,  -1
  beq   $9,   $8,   cas_non_deja_calcule
  ori   $2,   $8,  0
  j     epilogue_fib

cas_non_deja_calcule:
  bne   $4,   $0, cas_n_non_0
  ori   $2,   $0,  1
```

```
  sw    $2,   0($16)
  j     epilogue_fib
cas_n_non_0:
  ori   $2,   $0,  1
  bne   $4,   $2,  cas_n_sup_1
  sw    $2,   0($16)
  j     epilogue_fib
cas_n_sup_1:
       # fib(n-1, tab)
  addiu $4,   $4,  -1    # n - 1
  jal   fib
  ori   $17,  $2,   0    # sauvegarde du resultat de fib(n-1,tab)

  # fib(n-2, tab)
  lw    $4,   20($29)    # lecture n dans la pile
  addiu $4,   $4,  -2    # n - 2
  lw    $5,   24($29)    # lecture tab dans la pile
  jal   fib
  # tab[n] = fib(n-1, tab) + fib(n-2, tab)
  addu  $2,   $17,  $2
  sw    $2,   0($16)

epilogue_fib:
  lw    $31,  16($29)
  lw    $16,  12($29)
  lw    $17,   8($29)
  addiu $29, $29, 20
  jr    $31
```

Question 2.4 : 2 points

On suppose que la valeur lue est 5. Dessiner l'arbre des appels récursifs pour cette valeur (en tenant compte de votre code assembleur, non du code C).

Solution:

```
main -> fib(5) -> fib(4) -> fib(3) -> fib(2) -> fib(1)
                                               -> fib(0)
                                    -> fib(1) (calculé)
                           -> fib(2) (calculé)
                -> fib(3) (calculé)
```

Question 2.5 : 3 points

On suppose que la valeur lue est 5. Dessiner la pile avec son contenu lorsque l'on est à l'entrée de la fonction `fib` lors du premier appel à `fib(3)`.

Vous entourerez sur l'arbre dessiné à la question précédente l'appel correspondant.

Vous indiquerez les adresses de retour par `addr_ret1`, `addr_ret2`, `addr_ret3`, ... que vous ajouterez dans la marge devant les instructions correspondantes des codes donnés en réponse aux questions précédentes. Vous utiliserez ces noms dans votre schéma de pile.

Toute valeur d'argument sur la pile, le cas échéant, devra être numérique. Sinon, tout contenu d'un emplacement sur la pile contenant une valeur d'un registre du programme sera indiqué par le nom du registre.

Solution:

Exercice 3 : Architecture et programmation système – 12 points

Cette partie du module d'architecture est évaluée par un QCM et un petit exercice de codage en C.

Pour chaque question du QCM, nous faisons 4 affirmations et vous devez dire, pour chacune, si elle est vraie ou fausse en cochant l'une ou l'autre des cases correspondantes.

Pour une affirmation, si vous ne cochez aucune case ou si vous cochez les deux cases, alors votre réponse est considérée comme une erreur. Les questions ne sont pas difficiles, mais vous devez prendre le temps de réfléchir avant de répondre. Toutes les affirmations peuvent être vraies, ou toutes peuvent être fausses, ou il peut y en avoir un mélange de vraies et de fausses affirmations.

Pour chaque question du QCM, le barème est le suivant :
- 1 point si vous n'avez commis aucune erreur.
- 0,5 point si vous avez commis une erreur.
- 0 point si vous avez commis 2 erreurs ou plus.

Question 3.1 : 7 points

1. **Propositions sur l'espace d'adressage du MIPS**
   (a) vrai [ ] ou faux [ ]
   L'expace d'adressage, c'est l'ensemble des adresses que le MIPS peut produire.
   (b) vrai [ ] ou faux [ ]
   Les registres de périphériques sont dans l'espace d'adressage du MIPS.
   (c) vrai [ ] ou faux [ ]
   Les registres du processeur sont accessibles dans l'espace d'adressage du MIPS.
   (d) vrai [ ] ou faux [ ]
   L'espace d'adressage n'est accessible qu'en mode kernel.

   Solution:
   (a) vrai ; (b) vrai ; (c) faux ; (d) faux

2. **Propositions sur l'architecture vue dans le module**
   (a) vrai [ ] ou faux [ ]
   Les périphériques sont accédés comme de la mémoire.
   (b) vrai [ ] ou faux [ ]
   Le code de démarrage du MIPS est dans le segment de code du noyau.
   (c) vrai [ ] ou faux [ ]
   Les registres de contrôle des terminaux sont dans la section des variables globales du noyau uniquement.
   (d) vrai [ ] ou faux [ ]
   Le noyau choisit au démarrage les plages d'adresses (régions) utilisées par les mémoires.

   Solution:
   (a) vrai ; (b) faux ; (c) faux ; (d) faux

3. **Propositions sur les modes d'exécution du MIPS**
   (a) vrai [ ] ou faux [ ]
   Le MIPS dispose de deux modes d'exécution (user et kernel) et il peut passer de l'un à l'autre grâce à l'instruction privilégiée `stmd`.
   (b) vrai [ ] ou faux [ ]
   L'instruction `syscall` est une instruction privilégiée.

(c) vrai [ ] ou faux [ ]
L'existence d'un mode user pour les applications est indispensable, sans quoi l'utilisation du processeur serait impossible pour exécuter des programmes.

(d) vrai [ ] ou faux [ ]
l'instruction `eret` permet de passer du mode kernel au mode user.

Solution:

(a) faux ; (b) faux ; (c) faux ; (d) vrai

4. **Propositions sur la chaine de compilation**

   (a) vrai [ ] ou faux [ ]
   L'édition de liens produit le binaire exécutable à partir des différents fichiers objets issus de la phase de compilation

   (b) vrai [ ] ou faux [ ]
   L'outil `gcc` prend exclusivement en argument du code en langage C.

   (c) vrai [ ] ou faux [ ]
   Un Makefile permet de construire un exécutable grâce à des règles contenant des commandes shell.

   (d) vrai [ ] ou faux [ ]
   Le préprocesseur produit du code binaire (non éditable par un éditeur de texte normal).

   Solution:
   (a) vrai ; (b) faux ; (c) vrai ; (d) faux

5. **Propositions sur le fichier ldscript**

   (a) vrai [ ] ou faux [ ]
   Le fichier ldscript est utilisé pendant la compilation du code C (.c) pour produire le code objet (.o)

   (b) vrai [ ] ou faux [ ]
   Le fichier ldscript contient la description des régions de l'espace d'adressage occupé par la mémoire et la manière de les remplir avec les sections présentes dans les fichiers objet (.o).

   (c) vrai [ ] ou faux [ ]
   Les variables définies dans le fichier ldscript sont accessibles depuis le programme C.

   (d) vrai [ ] ou faux [ ]
   Si un programme est composé d'un seul fichier C et que ce fichier n'a qu'une seule fonction, le ldscript n'est pas obligatoire.

   Solution:
   (a) faux ; (b) vrai ; (c) vrai ; (d) faux

6. **Propositions sur le système d'exploitation**

   (a) vrai [ ] ou faux [ ]
   Le code du système d'exploitation s'exécute toujours en mode kernel.

   (b) vrai [ ] ou faux [ ]
   Le noyau du système d'exploitation est écrit en C et en ldscript.

   (c) vrai [ ] ou faux [ ]
   Les pilotes de périphériques doivent être nécessairement écrits en assembleur.

   (d) vrai [ ] ou faux [ ]
   L'emplacement en mémoire du noyau du système d'exploitation est un choix du programmeur.

   Solution:
   (a) faux ; (b) faux ; (c) faux ; (d) faux

7. **Propositions sur le passage de mode**

   (a) vrai [ ] ou faux [ ]
   Il y a seulement deux entrées dans le noyau `kinit` et `kentry`.

   (b) vrai [ ] ou faux [ ]
   Lorsqu'une application de l'utilisateur s'exécute, il est possible de masquer les exceptions grâce au registre de status (`c0_sr`).

(c) vrai [ ] ou faux [ ]
La première chose que le noyau doit faire après une des trois causes (syscall, interruption et exception) est d'analyser la cause d'appel.

(d) vrai [ ] ou faux [ ]
Les arguments des appels système sont donnés dans les registres $4 à $7.

Solution:

(a) vrai ; (b) faux ; (c) vrai ; (d) vrai

Question 3.2 : 5 points

L'adresse du contrôleur de TTY est `__tty_regs_maps=0xD0200000`, elle est définie dans le fichier '`kernel.ld`' et l'ordre des registres de contrôle est (par adresse croissante) : `write`, `status`, `read`, `unused`.

- `write` est le registre de sortie vers l'écran.
  Chaque écriture est faite à l'emplacement du curseur, lequel avance automatiquement.
- `status` est le registre qui contient 0 lorsqu'aucune touche n'a été tapée au clavier,
  et autre chose que 0, lorsqu'un caractère est en attente de lecture dans le registre `read`.
- `read` est le registre qui contient le code ASCII de la touche tapée au clavier.
  Il ne faut lire le registre `read` que si le registre `status` est différent de 0.

Pour déclarer les registres du contrôleur de TTY, vous utiliserez :

```
extern volatile int __tty_regs_maps[4];
```

Écrivez la fonction en C `puts()`. La fonction prend en argument un pointeur sur une chaine de caractères, elle envoie tous les caractères sur le terminal, puis elle envoie un '`\n`' (retour chariot) et elle rend le nombre de caractères affichés. Vous devez commenter votre code.
Le prototype de la fonction est : `int puts(char * str);`

Solution:

```
extern volatile int __tty_regs_maps[4];
int puts(char * str)
{
    int res = 0;
    while ( *str )
    {
        __tty_regs_maps[0] = *str ;
        res++;
        str++;
    }
    __tty_regs_maps[0] = '\n' ;
    return res+1;
}
```

Écrivez une fonction C qui lit un unique caractère depuis le clavier (vous devez utiliser les deux registres `read` et `status`). Vous devez commenter votre code.
Le prototype de la fonction est : `int getchar(void);`

Solution:

```
extern volatile int __tty_regs_maps[4];
int getchar(void)
{
    while ( __tty_regs_maps[1] == 0) ;
    return __tty_regs_maps[2];
}
```