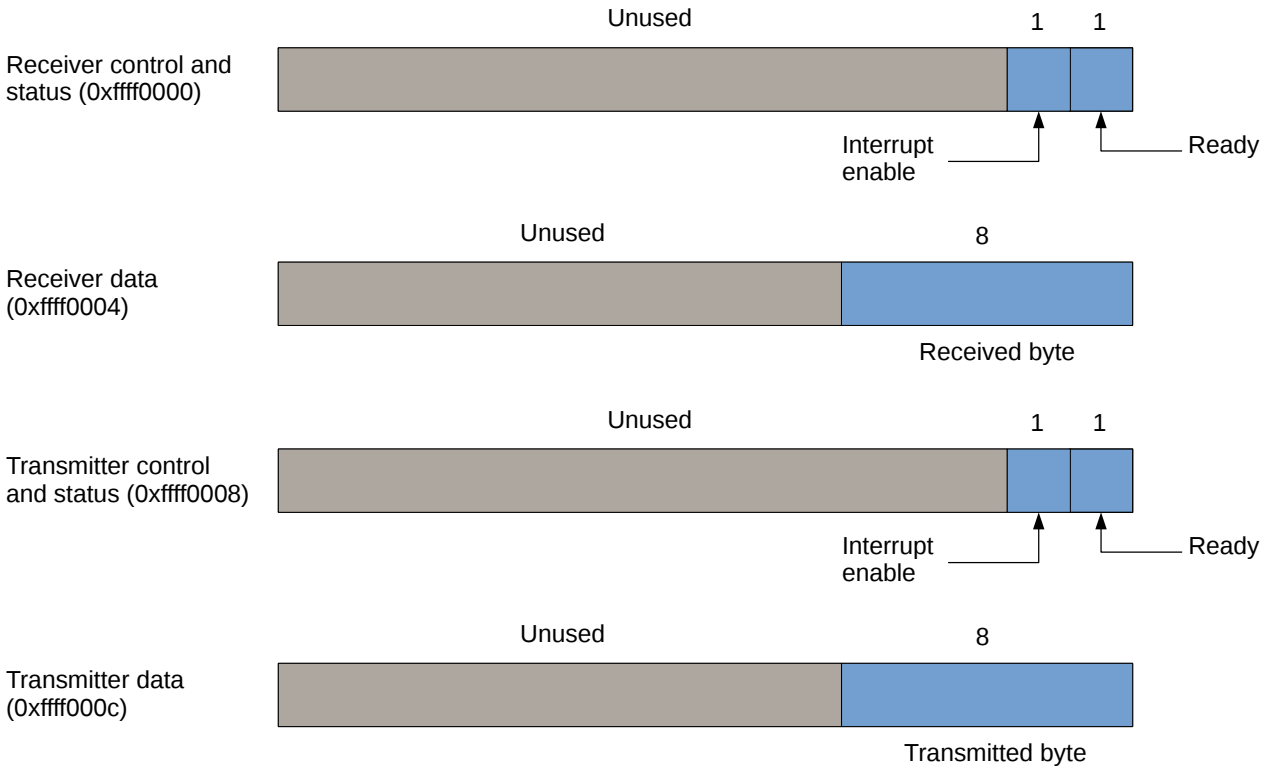
Architecture *MIPS* mono-cycle avec gestion des exceptions et des interruptions

Nous allons ajouter à notre architecture sur un cycle la gestion des exceptions et des interruptions et de nouvelles instructions.

1 Modèle de programmation

1.1 Espace d'adressage

Nous allons ajouter dans l'espace d'adressage de l'architecture mono-cycle un terminal qui sera géré par quatre registres :



Chacun de ces quatre registres, chacun sur 32 bits, apparaissent comme quatre adresses mémoires particulières.

Le registre de contrôle et d'état en réception (*Receiver control and status*) se trouve à l'adresse `0xffff0000`. Si le bit `0` (*Ready*) est à `1`, cela signifie qu'un caractère est arrivé

depuis le clavier, mais qu'on a pas encore lu la donnée (*Received byte*) du registre de donnée en réception (*Receiver data*). Si le bit `1` du registre de contrôle et d'état en réception (*Interrupt enable*) est à 1, une demande d'interruption sera envoyée au processeur tant que le bit *Ready* est à `1`. Lorsqu'on lit le registre de réception (*Receiver data*), le bit *Ready* passe à zéro. Initialement, les bits `Interrupt enable` et `Ready` sont à 0.

Le registre de contrôle et d'état en émission (*Transmitter control and status*) se trouve à l'adresse `0xffff0008`. Si le bit `0` (*Ready*) est à 1, l'émetteur est prêt à recevoir un nouveau caractère. Si ce bit est à 0, cela veut dire que l'émetteur est toujours occupé à envoyer le précédent caractère. Si le bit `1` de ce registre est à 1, une demande d'interruption sera envoyée au processeur quand le bit `0` est à 1, ce qui signifie que l'emetteur est prêt à envoyer un nouveau caractère. Le caractère que l'on veut envoyer est à écrire dans la partie (*Transmitted byte*) du registre d'émission (*Transmitter data*). Initialement, le bit `Interrupt enable` est à 0 et le bit `Ready` est à 1.

1.2 Gestion des exceptions et des interruptions

Nous allons implémenter de manière simplifiée la gestion des exceptions et des interruptions. Dans la terminologie *MIPS*, on parle d'interruption pour un évènement extérieur au processeur (lié à un périphérique) et d'exception pour un évènement interne au processeur (division par zéro par exemple).

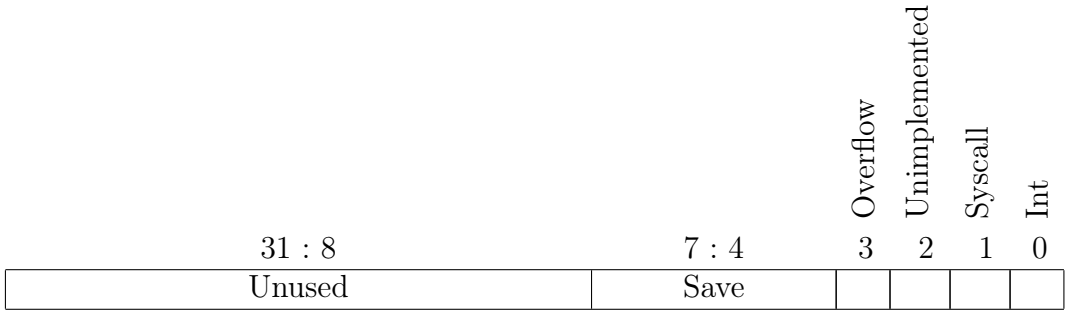
Dans notre implémentation,

- On ne gère que trois exceptions : débordement arithmétique, instruction non implémentée et appel système.
- Les seules interruptions viennent du terminal.
- On suppose que les instructions et exceptions sont masquées au lancement de la machine (registre `Status` à 0).

1.2.1 Registres

L'architecture *MIPS* possède normalement un coprocesseur 0 pour gérer, entre autres, les exceptions et interruptions. Nous n'implémenterons que trois des registres associés à ce coprocesseur : `Status`, `Cause` et `EPC`.

- Le registre numéro 12, appelé *Status register* (pour registre d'état), permet de savoir si les exceptions ou les interruptions sont masquées ou non. Ce registre est décrit ci-dessous :



La signification des différents champs est la suivante :

- ⋆ Le bit `0` est à 0 si les demandes d'interruptions (arrivant sur la broche `Int` du processeur) sont masquées. Elle est à 1 sinon.
- ⋆ Le bit `1` est à 0 si les appels systèmes sont masqués. Elle est à 1 sinon.
- ⋆ Le bit `2` est à 0 si l'exception correspondant à une instruction non implémentée est masquée. Elle est à 1 sinon.
- ⋆ Le bit `3` est à 0 si l'exception correspondant à un débordement est masquée. Elle est à 1 sinon.
- ⋆ Les bits `4` à `7` permettent de sauvegarder les quatre premiers bits lors du traitement d'une interruption ou d'une exception.

- Le registre numéro 13, appelé *Cause register* (pour registre de cause), permet de connaître la cause d'une exception ou d'une interruption. Ce registre est décrit ci-dessous :

Les bits `0` et `1` contiennent un nombre indiquant le type d'exception ou d'interruption qui vient de se produire. Les différentes valeurs qui nous intéressent sont décrites dans la table 1.

Valeur du champ `Exception Code`	Nom	Description
0	`Int`	Interruption provenant d'un contrôleur de périphérique
1	`Syscall`	On vient d'exécuter l'instruction `syscall`
2	`Unimplemented`	L'instruction n'est pas valide (instruction non reconnue)
3	`Overflow`	Un débordement s'est produit pour une instruction arithmétique

TABLE 1 – Les différentes valeurs du champ *Exception Code*.

- Le registre numéro 14, appelé `EPC`, permet de sauvegarder la valeur du compteur ordinal (`PC`) lors de la prise en compte d'une exception ou d'une interruption.

1.2.2 Déroulement de la gestion d'une exception ou d'une interruption

Notre gestion simplifiée des interruptions et des exceptions est la suivante :

- Pour les exceptions, l'adresse de l'instruction qui a déclenchée celle-ci est sauvegardée dans le registre `EPC`. Pour les interruptions, l'adresse de l'instruction suivante est sauvegardée dans `EPC`.
- Le contenu du registre `Status` est décalé à gauche de 4 bits afin de sauvegarder les valeurs actuelles de ce registre et de masquer les interruptions et les exceptions.
- Le champ `ExcCode` du registre `Cause` est mis à jour.

- On met dans le compeur ordinal l'adresse `0x80000180` qui est celle du traitant d'interruption.
- Lorsque l'on sort du traitant d'interruption ou d'exception (instruction `eret`), le contenu du registre `Status` est restauré en le décalant de 4 bits vers la droite.
- On remet dans le compteur ordinal la valeur de `EPC`.

1.2.3 Instructions

Nous allons ajouter quatre nouvelles instructions qui vont permettre de réaliser des appels système dans le cadre des systèmes d'exploitation (`syscall`), de sortir d'un traitant d'interruption (`eret`), de déplacer des valeurs du banc de registres du processeur vers celui du coprocesseur 0 (`mtc0`) et de déplacer des valeurs du banc de registres du coprocesseur 0 vers celui du processeur (`mfc0`).

- `syscall`
  Cette instruction permet de générer une exception afin de passer la main au système d'exploitation. L'addresse de l'instruction est sauvegardée dans le registre `EPC`, la cause de l'exception `Syscall` est placée dans le registre de cause (*Cause register*) (voir la table 1). Le compteur ordinal est alors chargé avec l'adresse du traitant d'exceptions ou d'interruptions (adresse `0x80000180`). L'instruction `syscall` est codée en langage machine :

0	001100
26 bits	6 bits

- `eret`
  Cette instruction permet de sortir d'un traitant d'exceptions ou d'interruptions. Elle a pour effet de remettre simultanément le compteur ordinal à la valeur sauvegardée dans le registre `EPC` et de décaler de 4 vers la droite le registre `Status`. L'instruction `eret` est codée en langage machine :

100000	1	0	011000
6 bits	1 bits	19 bits	6 bits

- `mtc0 rt, rd`
  Cette instruction permet de copier la valeur du registre numéro `rt` du banc de registres du processeur vers le registre numéro `rd` du banc de registres du coprocesseur 0. L'instruction `mtc0 rt, rd` est codée en langage machine :

010000	00100	rt	rd	0
6 bits	5 bits	5 bits	5 bits	11 bits

- `mfc0 rd, rt`
  Cette instruction permet de copier la valeur du registre numéro `rt` du banc de registres du coprocesseur 0 vers le registre numéro `rd` du banc de registres du processeur. L'instruction `mfc0 rd, rt` est codée en langage machine :

010000	0	rt	rd	0
6 bits	5 bits	5 bits	5 bits	11 bits

1.3 Jeu complet d'instructions

Le jeu complet d'instructions de notre architecture *MIPS* est décrit dans la table 2.

Inst.	[31 : 26]	[25 : 21]	[20 : 16]	[15 : 11]	[10 : 6]	[5 : 0]	Signification
add	000000	rs	rt	rd	00000	100000	R[rd] ← R[rs] + R[rt]
sub	000000	rs	rt	rd	00000	100010	R[rd] ← R[rs] − R[rt]
and	000000	rs	rt	rd	00000	100100	R[rd] ← R[rs] `&` R[rt]
or	000000	rs	rt	rd	00000	100101	R[rd] ← R[rs] `|` R[rt]
xor	000000	rs	rt	rd	00000	100110	R[rd] ← R[rs] `^` R[rt]
slt	000000	rs	rt	rd	00000	101010	R[rd] ← 1 si R[rs] < R[rt] R[rd] ← 0 sinon
sll	000000	00000	rt	rd	sa	000000	rd ← rs << sa
srl	000000	00000	rt	rd	sa	000010	rd ← rs >>> sa
sra	000000	00000	rt	rd	sa	000011	rd ← rs >> sa
addi	001000	rs	rt	v(aleur immédiate)			R[rt] ← R[rs] + (signe)v
andi	001100	rs	rt	v(aleur immédiate)			R[rt] ← R[rs] `&` (zéro)v
ori	001101	rs	rt	v(aleur immédiate)			R[rt] ← R[rs] `|` (zéro)v
xori	001110	rs	rt	v(aleur immédiate)			R[rt] ← R[rs] `^` (zéro)v
lui	001111	00000	rt	v(aleur immédiate)			R[rt] ← v << 16
lw	100011	rs	rt	d(éplacement)			R[rt] ← M[R[rs] + (signe)d]
sw	101011	rs	rt	d(éplacement)			M[R[rs] + (signe)d] ← R[rt]
beq	000100	rs	rt	d(éplacement)			si R[rs] = R[rt] alors pc ← pc + 4 + d * 4
bne	000101	rs	rt	d(éplacement)			si R[rs] ≠ R[rt] alors pc ← pc + 4 + d * 4
j	000010	a(dresse)					pc ← $(pc + 4)_{[31:28]}(a * 4)_{[27:0]}$
jal	000011	a(dresse)					R[$31] ← pc + 4 pc ← $(pc + 4)_{[31:28]}(a * 4)_{[27:0]}$
jr	000000	rs	00000	00000	00000	001000	pc ← R[rs]
mfc0	010000	00000	rt	rd	00000	000000	R[rd] ← C0_R[rt]
mtc0	010000	00100	rt	rd	00000	000000	C0_R[rd] ← R[rt]
syscall	000000	00000	00000	00000	00000	011000	appel système
eret	000000	00000	00000	00000	00000	011000	retour du traitant d'exception ou d'interruption

TABLE 2 – Notre jeu complet d'instructions pour notre implémentation de *MIPS*.

On pourra aussi utiliser la pseudo-instruction `la`, qui permet de charger l'adresse d'un label dans un registre. Par exemple, le programme suivant,

```
.bss
input_index:  .word
.text
la $t0, input_index
```
mettra l'adresse du label `input_index` dans le registre `$t0`.

On pourra aussi utiliser la pseudo-instruction `li` qui permet de charger une valeur sur 32 bits dans un registre, par exemple, `li $t0, 0xbadcaffe`.

1.4 Programmation de la gestion du terminal

Nous supposons que nous disposons de toutes les instructions de la table 2. Notons que les registres `$k0` et `$k1` sont réservés pour le système d'exploitation et pourront être utilisés sans avoir à être sauvegardés dans le traitant d'exceptions et d'interruptions.

1.4.1 Réception par attente active

Écrire la fonction `read_busy_waiting` permettant de stocker, en utilisant de l'attente active, les éléments reçus du clavier dans le buffer `input_buffer`. On suppose, pour simplier, que le buffer ne sera jamais plein.

```
  .kdata
input_index:  .word 0
input_buffer: .space BIG

  .ktext
read_busy_waiting:
  # to do
```

1.4.2 Réception par interruption

Écrire le traitant d'interruption, `it_handler`, qui permet de stocker les éléments reçus du clavier dans le buffer `input_buffer`.

```
  .kdata
input_index:  .word 0
input_buffer: .space BIG

  .ktext 0x80000180
it_handler:
  # to do
```

1.4.3 Émission par attente active

Écrire la fonction `write_busy_waiting` permettant d'écrire, en utilisant de l'attente active, les éléments du buffer `output_buffer` vers le terminal. Pour simplifier, on suppose que le buffer contient une quantité énorme de données, et on ne se soucie pas de savoir quand on a atteint la fin de celui-ci.

```
  .kdata
output_index:  .word 0
output_buffer: .space BIG

  .ktext
write_busy_waiting:
  # to do
```

1.4.4 Émission par interruption

Écrire le traitant d'interruption, `it_handler`, qui permet d'envoyer les éléments du buffer `output_buffer` vers le terminal.

```
  .kdata
output_index:  .word 0
output_buffer: .space BIG

  .ktext 0x80000180
it_handler:
  # to do
```

1.5 Modification de l'architecture mono-cycle

Nous rappelons ici l'architecture réalisée lors du dernier TD :

- La figure 1 décrit l'unité de traitement.
- La table 3 décrit l'unité de contrôle.

Ligne de contrôle	Condition
`RegDst`	op = 0
`RegWrite`	op = 0 ou op = lw
`ALUSrc`	op = lw ou op = sw
`Branch`	op = beq
`Jump`	op = j
`MemRead`	op = lw
`MemWrite`	op = sw
`MemToReg`	op = lw
`ALUOp0`	op = beq
`ALUOp1`	op = 0

TABLE 3 – Contrôle de l'architecture mono-cycle.

Ajouter la gestion des exceptions et des interruptions à l'architecture mono-cycle.

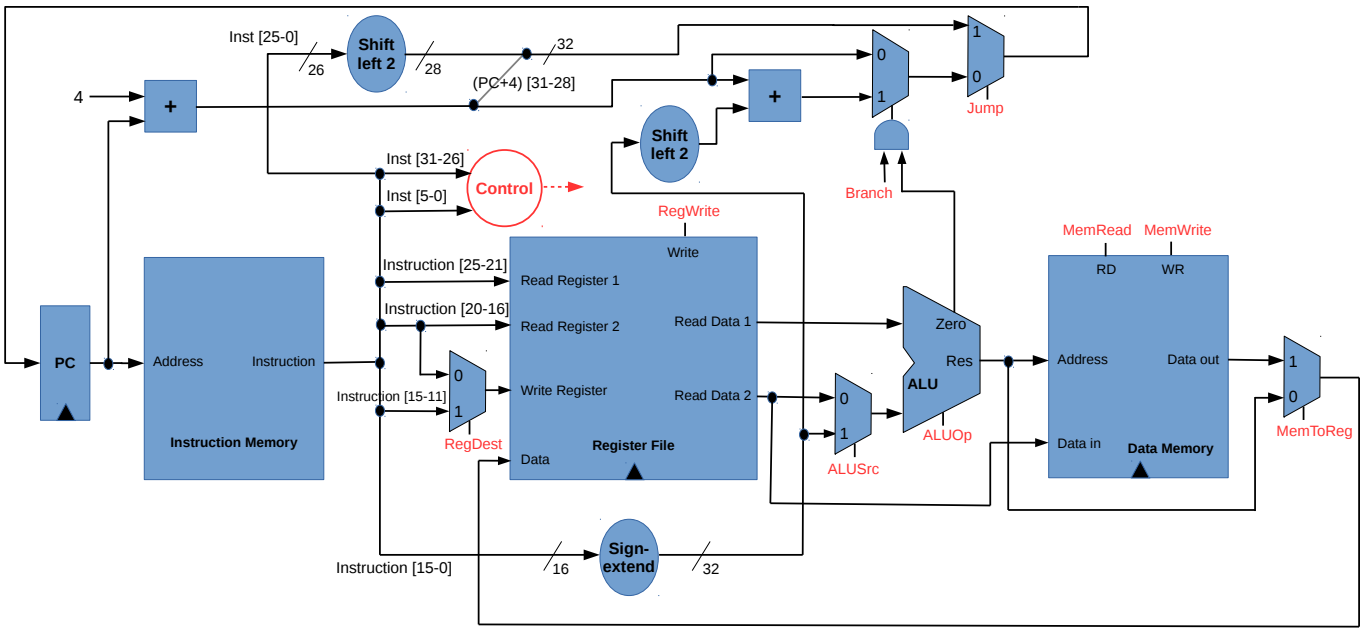


FIGURE 1 – Unité de traitement de l'architecture mono-cycle.