les écoles supérieures

ETML

MINF
Programmation des PIC32MX

Chapitre 3

Jeu d'instructions

Théorie PIC32MX

Christian HUBER (CHR)
Serge CASTOLDI (SCA)
Version 1.9 novembre 2017

CONTENU

3. JEU D'INSTRUCTIONS DU PIC32

Ce chapitre traite du jeu d'instructions du PIC32.

Le concept du jeu d'instructions du PIC32 s'appuie sur le standard MIPS32. C'est pour cela que l'on ne trouve pas de description du jeu d'instructions dans la documentation spécifique au PIC32.

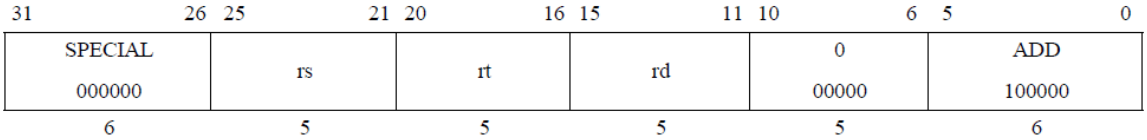
Le document intitulé "MIPS Architecture for Programmers, Volume II-A: The MIPS32 Instruction Set Manual", que l'on trouve sur le réseau sous ...\PROJETS\SLO\1102x_SK32MX775F512L\Data_sheets\PIC32 Family Reference Manual, décrit en détail le jeu d'instructions.

3.1. STRUCTURE DES INSTRUCTIONS

Les instructions du PIC32 sont codées sur 32 bits. L'organisation des 32 bits de l'instruction varie s'il s'agit d'une instruction réalisant une opération, un accès à la mémoire ou un branchement (saut).

3.1.1. INSTRUCTIONS D'OPERATION

Pour comprendre l'organisation des instructions effectuant une opération arithmétique ou logique, voici l'exemple d'une instruction d'addition :



Le code de l'opération tient sur 6 bits. Au niveau des opérandes, 3 registres sont impliqués.

Le format de l'instruction est `ADD rd, rs, rt`
rd, rs et rt indiquent le numéro du GPR (General Purpose Register) impliqué dans l'opération, avec rd registre destination, rs registre source et rt registre temporaire.

L'action d'addition correspond à : `rd ← rs + rt`

3.1.2. INSTRUCTIONS D'OPERATION

3.1.2.1. OPERATIONS ARITHMETIQUES

ARITHMETIC OPERATIONS			
ADD	RD, RS, RT	RD = RS + RT	(OVERFLOW TRAP)
ADDI	RD, RS, CONST16	RD = RS + CONST16$^{\pm}$	(OVERFLOW TRAP)
ADDIU	RD, RS, CONST16	RD = RS + CONST16$^{\pm}$	
ADDU	RD, RS, RT	RD = RS + RT	
CLO	RD, RS	RD = COUNTLEADINGONES(RS)	
CLZ	RD, RS	RD = COUNTLEADINGZEROS(RS)	
LA	RD, LABEL	RD = ADDRESS(LABEL)	
LI	RD, IMM32	RD = IMM32	
LUI	RD, CONST16	RD = CONST16 << 16	
MOVE	RD, RS	RD = RS	
NEGU	RD, RS	RD = –RS	
SEB^{R2}	RD, RS	RD = $RS_{7:0}^{\pm}$	
SEH^{R2}	RD, RS	RD = $RS_{15:0}^{\pm}$	
SUB	RD, RS, RT	RD = RS – RT	(OVERFLOW TRAP)
SUBU	RD, RS, RT	RD = RS – RT	

3.1.2.2. OPERATIONS DE DECALAGE ET ROTATION

SHIFT AND ROTATE OPERATIONS		
ROTR^{R2}	RD, RS, BITS5	RD = $RS_{BITS5-1:0}$ :: $RS_{31:BITS5}$
ROTRV^{R2}	RD, RS, RT	RD = $RS_{RT4:0-1:0}$ :: $RS_{31:RT4:0}$
SLL	RD, RS, SHIFT5	RD = RS << SHIFT5
SLLV	RD, RS, RT	RD = RS << $RT_{4:0}$
SRA	RD, RS, SHIFT5	RD = $RS^{\pm}$ >> SHIFT5
SRAV	RD, RS, RT	RD = $RS^{\pm}$ >> $RT_{4:0}$
SRL	RD, RS, SHIFT5	RD = $RS^{\varnothing}$ >> SHIFT5
SRLV	RD, RS, RT	RD = $RS^{\varnothing}$ >> $RT_{4:0}$

3.1.2.3. OPERATIONS LOGIQUES

Logical and Bit-Field Operations		
AND	RD, RS, RT	RD = RS & RT
ANDI	RD, RS, CONST16	RD = RS & CONST16$^{\varnothing}$
EXT^{R2}	RD, RS, P, S	RS = RS$_{P+S-1:P}^{\varnothing}$
INS^{R2}	RD, RS, P, S	RD$_{P+S-1:P}$ = RS$_{S-1:0}$
NOP		NO-OP
NOR	RD, RS, RT	RD = ~(RS | RT)
NOT	RD, RS	RD = ~RS
OR	RD, RS, RT	RD = RS | RT
ORI	RD, RS, CONST16	RD = RS | CONST16$^{\varnothing}$
WSBH^{R2}	RD, RS	RD = RS$_{23:16}$ :: RS$_{31:24}$ :: RS$_{7:0}$ :: RS$_{15:8}$
XOR	RD, RS, RT	RD = RS ⊕ RT
XORI	RD, RS, CONST16	RD = RS ⊕ CONST16$^{\varnothing}$

3.1.2.4. OPERATIONS D'ACTION CONDITIONNELLE

Condition Testing and Conditional Move Operations		
MOVN	RD, RS, RT	IF RT ≠ 0, RD = RS
MOVZ	RD, RS, RT	IF RT = 0, RD = RS
SLT	RD, RS, RT	RD = (RS$^{\pm}$ < RT$^{\pm}$) ? 1 : 0
SLTI	RD, RS, CONST16	RD = (RS$^{\pm}$ < CONST16$^{\pm}$) ? 1 : 0
SLTIU	RD, RS, CONST16	RD = (RS$^{\varnothing}$ < CONST16$^{\varnothing}$) ? 1 : 0
SLTU	RD, RS, RT	RD = (RS$^{\varnothing}$ < RT$^{\varnothing}$) ? 1 : 0

3.1.2.5. OPERATIONS DE MULTIPLICATION ET DIVISION

Multiply and Divide Operations		
DIV	RS, RT	LO = RS$^{\pm}$ / RT$^{\pm}$; HI = RS$^{\pm}$ MOD RT$^{\pm}$
DIVU	RS, RT	LO = RS$^{\varnothing}$ / RT$^{\varnothing}$; HI = RS$^{\varnothing}$ MOD RT$^{\varnothing}$
MADD	RS, RT	ACC += RS$^{\pm}$ × RT$^{\pm}$
MADDU	RS, RT	ACC += RS$^{\varnothing}$ × RT$^{\varnothing}$
MSUB	RS, RT	ACC −= RS$^{\pm}$ × RT$^{\pm}$
MSUBU	RS, RT	ACC −= RS$^{\varnothing}$ × RT$^{\varnothing}$
MUL	RD, RS, RT	RD = RS$^{\pm}$ × RT$^{\pm}$
MULT	RS, RT	ACC = RS$^{\pm}$ × RT$^{\pm}$
MULTU	RS, RT	ACC = RS$^{\varnothing}$ × RT$^{\varnothing}$

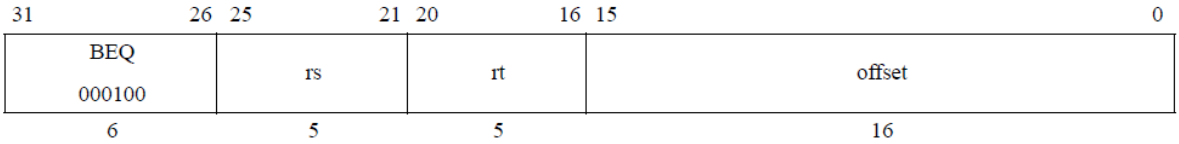
3.1.2.6. OPERATIONS D'ACCES A L'ACCUMULATEUR

ACCUMULATOR ACCESS OPERATIONS		
MFHI	RD	RD = HI
MFLO	RD	RD = LO
MTHI	RS	HI = RS
MTLO	RS	LO = RS

3.1.3. INSTRUCTIONS DE SAUT ET BRANCHEMENTS

3.1.3.1. INSTRUCTIONS DE SAUT CONDITIONNEL

L'instruction BEQ (Branch on EQual) illustre bien ce type d'instruction :



Son mnémonique est : `BEQ rs, rt, offset` et son action est :

```
if rs = rt then branch
```

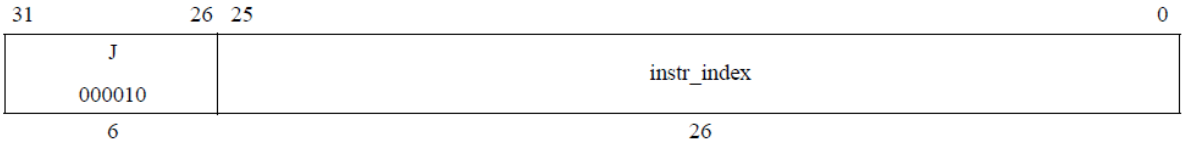
rs et *rt* spécifient un no de registre.

Les détails de l'exécution sont les suivants :

```
I:      target_offset ← sign_extend(offset || 0^2)
           condition ← (GPR[rs] = GPR[rt])
I+1:    if condition then
           PC ← PC + target_offset
        endif
```

3.1.3.2. INSTRUCTION J (JUMP)

Voici le format de l'instruction J (jump)



Mnémonique : `J target`

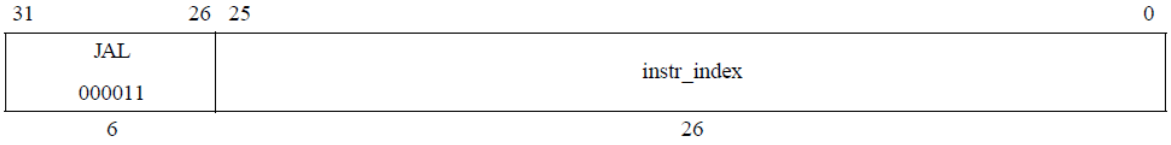
La valeur de instr_index est décalée à gauche de 2 pour former une valeur 28 bits.

Détails exécution :

```
I:
I+1:PC ← PC_GPRLEN..28 || instr_index || 0^2
```

3.1.3.3. INSTRUCTION JAL (JUMP AND LINK)

Voici le format de l'instruction JAL (jump and link), cette instruction correspond à un CALL (appel de routine).



Mnémonique : `JAL target`

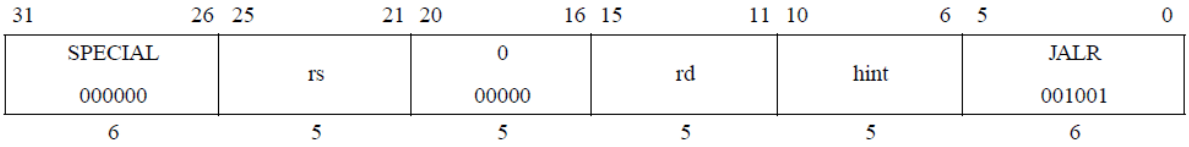
La valeur de instr_index est décalée à gauche de 2 pour former une valeur 28 bits.

Détails exécution :

```
I:   GPR[31]← PC + 8
I+1:PC      ← PC_GPRLEN..28 || instr_index || 0^2
```

3.1.3.4. INSTRUCTION JALR (JUMP AND LINK REGISTER)

Voici le format de l'instruction JALR (Jump And Link Register) :



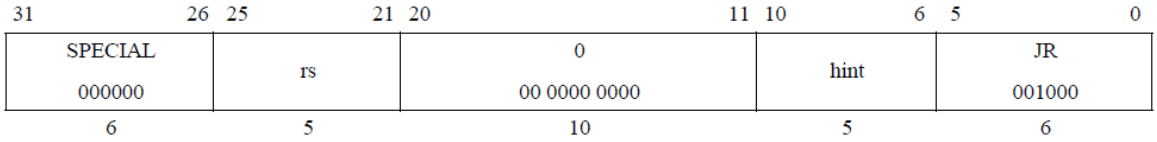
Mnémoniques :
```
JALR rs (rd = 31 implied)
JALR rd, rs
```

Principe exécution : `rd ← return_addr, PC ← rs`

Cette instruction effectue un CALL, l'adresse de destination est fournie par rs, tandis que l'adresse de retour est mémorisée dans rd.

3.1.3.5. INSTRUCTION JR (JUMP REGISTER)

Voici le format de l'instruction JR (Jump Register) :



Mnémoniques : `JR rs`

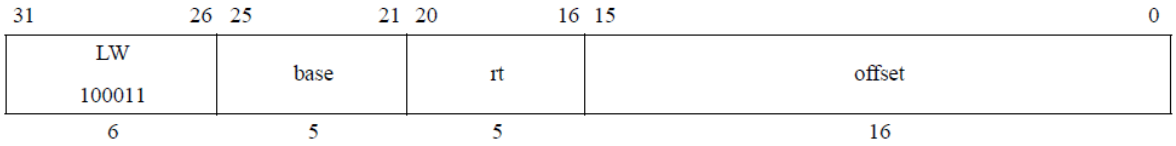
Principe exécution : `PC ← rs`

3.1.3.6. LISTE DES INSTRUCTIONS DE SAUT ET BRANCHEMENT

JUMPS AND BRANCHES (NOTE: ONE DELAY SLOT)		
B	OFF18	PC += OFF18$^{\pm}$
BAL	OFF18	RA = PC + 8, PC += OFF18$^{\pm}$
BEQ	RS, RT, OFF18	IF RS = RT, PC += OFF18$^{\pm}$
BEQZ	RS, OFF18	IF RS = 0, PC += OFF18$^{\pm}$
BGEZ	RS, OFF18	IF RS ≥ 0, PC += OFF18$^{\pm}$
BGEZAL	RS, OFF18	RA = PC + 8; IF RS ≥ 0, PC += OFF18$^{\pm}$
BGTZ	RS, OFF18	IF RS > 0, PC += OFF18$^{\pm}$
BLEZ	RS, OFF18	IF RS ≤ 0, PC += OFF18$^{\pm}$
BLTZ	RS, OFF18	IF RS < 0, PC += OFF18$^{\pm}$
BLTZAL	RS, OFF18	RA = PC + 8; IF RS < 0, PC += OFF18$^{\pm}$
BNE	RS, RT, OFF18	IF RS ≠ RT, PC += OFF18$^{\pm}$
BNEZ	RS, OFF18	IF RS ≠ 0, PC += OFF18$^{\pm}$
J	ADDR28	PC = PC$_{31:28}$ :: ADDR28$^{\varnothing}$
JAL	ADDR28	RA = PC + 8; PC = PC$_{31:28}$ :: ADDR28$^{\varnothing}$
JALR	RD, RS	RD = PC + 8; PC = RS
JR	RS	PC = RS

3.1.4. INSTRUCTIONS LOAD/STORE

Voici le format de l'instruction LW (Load Word) pour illustrer l'organisation de ce type d'instructions :

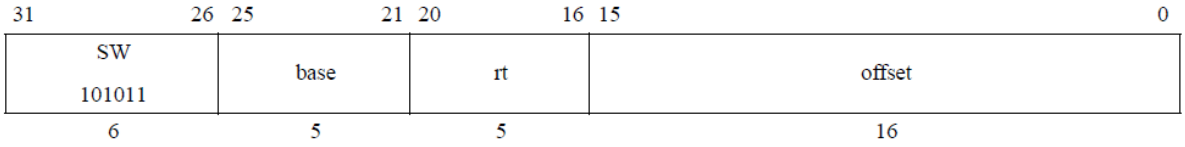


L'instruction s'écrit `LW rt, offset(base)`, et son action est la suivante :

```
rt ← memory[base+offset]
```

base correspond au no du registre contenant l'adresse de base. Pour atteindre la mémoire, il y a combinaison de l'adresse de base et de l'offset. La valeur lue est stockée dans le registre spécifié par *rt*.

Pour comparaison, voici l'instruction SW (Store Word) :



L'instruction s'écrit `SW rt, offset(base)` et son action est la suivante :

```
memory[base+offset] ← rt
```

La valeur du registre spécifié par *rt* est transférée dans la mémoire à l'adresse obtenue par la combinaison de la valeur du registre spécifié par *base* et de l'*offset*.

3.1.4.1. INSTRUCTIONS DE LOAD & STORE

LOAD AND STORE OPERATIONS		
LB	RD, OFF16(RS)	RD = MEM8(RS + OFF16$^{\pm}$)$^{\pm}$
LBU	RD, OFF16(RS)	RD = MEM8(RS + OFF16$^{\pm}$)$^{\varnothing}$
LH	RD, OFF16(RS)	RD = MEM16(RS + OFF16$^{\pm}$)$^{\pm}$
LHU	RD, OFF16(RS)	RD = MEM16(RS + OFF16$^{\pm}$)$^{\varnothing}$
LW	RD, OFF16(RS)	RD = MEM32(RS + OFF16$^{\pm}$)
LWL	RD, OFF16(RS)	RD = LOADWORDLEFT(RS + OFF16$^{\pm}$)
LWR	RD, OFF16(RS)	RD = LOADWORDRIGHT(RS + OFF16$^{\pm}$)
SB	RS, OFF16(RT)	MEM8(RT + OFF16$^{\pm}$) = RS$_{7:0}$
SH	RS, OFF16(RT)	MEM16(RT + OFF16$^{\pm}$) = RS$_{15:0}$
SW	RS, OFF16(RT)	MEM32(RT + OFF16$^{\pm}$) = RS
SWL	RS, OFF16(RT)	STOREWORDLEFT(RT + OFF16$^{\pm}$, RS)
SWR	RS, OFF16(RT)	STOREWORDRIGHT(RT + OFF16$^{\pm}$, RS)
ULW	RD, OFF16(RS)	RD = UNALIGNED_MEM32(RS + OFF16$^{\pm}$)
USW	RS, OFF16(RT)	UNALIGNED_MEM32(RT + OFF16$^{\pm}$) = RS

3.1.4.2. INSTRUCTIONS DE RMW ATOMIQUE

Ces 2 instructions sont appairées. Utilisées judicieusement ensemble, elles permettent des opérations, de lecture-modification-écriture (RMW : Read-Modifiy-Write) atomiques :

ATOMIC READ-MODIFY-WRITE OPERATIONS		
LL	RD, OFF16(RS)	RD = MEM32(RS + OFF16$^{\pm}$); LINK
SC	RD, OFF16(RS)	IF ATOMIC, MEM32(RS + OFF16$^{\pm}$) = RD; RD = ATOMIC ? 1 : 0

3.2. LES REGISTRES DU PIC32

Pour comprendre les choix possibles, voici le principe d'utilisation des 32 GPR (General Purpose Registers) :

REGISTERS		
0	zero	Always equal to zero
1	at	Assembler temporary; used by the assembler
2-3	v0-v1	Return value from a function call
4-7	a0-a3	First four parameters for a function call
8-15	t0-t7	Temporary variables; need not be preserved
16-23	s0-s7	Function variables; must be preserved
24-25	t8-t9	Two more temporary variables
26-27	k0-k1	Kernel use registers; may change unexpectedly
28	gp	Global pointer
29	sp	Stack pointer
30	fp/s8	Stack frame pointer or subroutine variable
31	ra	Return address of the last subroutine call

MIPS32® Instruction Set Quick Reference

RD	– Destination register
RS, RT	– Source operand registers
RA	– Return address register (R31)
PC	– Program counter
ACC	– 64-bit accumulator
LO, HI	– Accumulator low ($ACC_{31:0}$) and high ($ACC_{63:32}$) parts
±	– Signed operand or sign extension
∅	– Unsigned operand or zero extension
::	– Concatenation of bit fields
R2	– MIPS32 Release 2 instruction
DOTTED	– Assembler pseudo-instruction

Please refer to *"MIPS32 Architecture For Programmers Volume II: The MIPS32 Instruction Set"* for complete instruction set information.

Arithmetic Operations			
ADD	RD, RS, RT	RD = RS + RT	(overflow trap)
ADDI	RD, RS, const16	RD = RS + const16±	(overflow trap)
ADDIU	RD, RS, const16	RD = RS + const16±	
ADDU	RD, RS, RT	RD = RS + RT	
CLO	RD, RS	RD = CountLeadingOnes(RS)	
CLZ	RD, RS	RD = CountLeadingZeros(RS)	
LA	RD, label	RD = Address(label)	
LI	RD, imm32	RD = imm32	
LUI	RD, const16	RD = const16 << 16	
MOVE	RD, RS	RD = RS	
NEGU	RD, RS	RD = –RS	
SEB[R2]	RD, RS	$RD = RS_{7:0}^{\pm}$	
SEH[R2]	RD, RS	$RD = RS_{15:0}^{\pm}$	
SUB	RD, RS, RT	RD = RS – RT	(overflow trap)
SUBU	RD, RS, RT	RD = RS – RT	

Shift and Rotate Operations		
ROTR[R2]	RD, RS, bits5	$RD = RS_{bits5-1:0} :: RS_{31:bits5}$
ROTRV[R2]	RD, RS, RT	$RD = RS_{RT4:0-1:0} :: RS_{31:RT4:0}$
SLL	RD, RS, shift5	RD = RS << shift5
SLLV	RD, RS, RT	$RD = RS << RT_{4:0}$
SRA	RD, RS, shift5	$RD = RS^{\pm} >> shift5$
SRAV	RD, RS, RT	$RD = RS^{\pm} >> RT_{4:0}$
SRL	RD, RS, shift5	$RD = RS^{\varnothing} >> shift5$
SRLV	RD, RS, RT	$RD = RS^{\varnothing} >> RT_{4:0}$

Logical and Bit-Field Operations		
AND	RD, RS, RT	RD = RS & RT
ANDI	RD, RS, const16	$RD = RS \,\&\, const16^{\varnothing}$
EXT[R2]	RD, RS, P, S	$RS = RS_{P+S-1:P}^{\varnothing}$
INS[R2]	RD, RS, P, S	$RD_{P+S-1:P} = RS_{S-1:0}$
NOP		No-op
NOR	RD, RS, RT	RD = ~(RS | RT)
NOT	RD, RS	RD = ~RS
OR	RD, RS, RT	RD = RS | RT
ORI	RD, RS, const16	$RD = RS \mid const16^{\varnothing}$
WSBH[R2]	RD, RS	$RD = RS_{23:16} :: RS_{31:24} :: RS_{7:0} :: RS_{15:8}$
XOR	RD, RS, RT	RD = RS ⊕ RT
XORI	RD, RS, const16	$RD = RS \oplus const16^{\varnothing}$

Condition Testing and Conditional Move Operations		
MOVN	RD, RS, RT	if RT ≠ 0, RD = RS
MOVZ	RD, RS, RT	if RT = 0, RD = RS
SLT	RD, RS, RT	$RD = (RS^{\pm} < RT^{\pm})\ ?\ 1 : 0$
SLTI	RD, RS, const16	$RD = (RS^{\pm} < const16^{\pm})\ ?\ 1 : 0$
SLTIU	RD, RS, const16	$RD = (RS^{\varnothing} < const16^{\varnothing})\ ?\ 1 : 0$
SLTU	RD, RS, RT	$RD = (RS^{\varnothing} < RT^{\varnothing})\ ?\ 1 : 0$

Multiply and Divide Operations		
DIV	RS, RT	$LO = RS^{\pm} / RT^{\pm}$; $HI = RS^{\pm} \bmod RT^{\pm}$
DIVU	RS, RT	$LO = RS^{\varnothing} / RT^{\varnothing}$; $HI = RS^{\varnothing} \bmod RT^{\varnothing}$
MADD	RS, RT	$ACC \mathrel{+}= RS^{\pm} \times RT^{\pm}$
MADDU	RS, RT	$ACC \mathrel{+}= RS^{\varnothing} \times RT^{\varnothing}$
MSUB	RS, RT	$ACC \mathrel{-}= RS^{\pm} \times RT^{\pm}$
MSUBU	RS, RT	$ACC \mathrel{-}= RS^{\varnothing} \times RT^{\varnothing}$
MUL	RD, RS, RT	$RD = RS^{\pm} \times RT^{\pm}$
MULT	RS, RT	$ACC = RS^{\pm} \times RT^{\pm}$
MULTU	RS, RT	$ACC = RS^{\varnothing} \times RT^{\varnothing}$

Accumulator Access Operations		
MFHI	RD	RD = HI
MFLO	RD	RD = LO
MTHI	RS	HI = RS
MTLO	RS	LO = RS

Jumps And Branches (Note: One Delay Slot)		
B	off18	$PC \mathrel{+}= off18^{\pm}$
BAL	off18	$RA = PC + 8$; $PC \mathrel{+}= off18^{\pm}$
BEQ	RS, RT, off18	if RS = RT, $PC \mathrel{+}= off18^{\pm}$
BEQZ	RS, off18	if RS = 0, $PC \mathrel{+}= off18^{\pm}$
BGEZ	RS, off18	if RS ≥ 0, $PC \mathrel{+}= off18^{\pm}$
BGEZAL	RS, off18	$RA = PC + 8$; if RS ≥ 0, $PC \mathrel{+}= off18^{\pm}$
BGTZ	RS, off18	if RS > 0, $PC \mathrel{+}= off18^{\pm}$
BLEZ	RS, off18	if RS ≤ 0, $PC \mathrel{+}= off18^{\pm}$
BLTZ	RS, off18	if RS < 0, $PC \mathrel{+}= off18^{\pm}$
BLTZAL	RS, off18	$RA = PC + 8$; if RS < 0, $PC \mathrel{+}= off18^{\pm}$
BNE	RS, RT, off18	if RS ≠ RT, $PC \mathrel{+}= off18^{\pm}$
BNEZ	RS, off18	if RS ≠ 0, $PC \mathrel{+}= off18^{\pm}$
J	addr28	$PC = PC_{31:28} :: addr28^{\varnothing}$
JAL	addr28	$RA = PC + 8$; $PC = PC_{31:28} :: addr28^{\varnothing}$
JALR	RD, RS	RD = PC + 8; PC = RS
JR	RS	PC = RS

Load and Store Operations		
LB	RD, off16(RS)	$RD = mem8(RS + off16^{\pm})^{\pm}$
LBU	RD, off16(RS)	$RD = mem8(RS + off16^{\pm})^{\varnothing}$
LH	RD, off16(RS)	$RD = mem16(RS + off16^{\pm})^{\pm}$
LHU	RD, off16(RS)	$RD = mem16(RS + off16^{\pm})^{\varnothing}$
LW	RD, off16(RS)	$RD = mem32(RS + off16^{\pm})$
LWL	RD, off16(RS)	RD = LoadWordLeft(RS + off16±)
LWR	RD, off16(RS)	RD = LoadWordRight(RS + off16±)
SB	RS, off16(RT)	$mem8(RT + off16^{\pm}) = RS_{7:0}$
SH	RS, off16(RT)	$mem16(RT + off16^{\pm}) = RS_{15:0}$
SW	RS, off16(RT)	$mem32(RT + off16^{\pm}) = RS$
SWL	RS, off16(RT)	StoreWordLeft(RT + off16±, RS)
SWR	RS, off16(RT)	StoreWordRight(RT + off16±, RS)
ULW	RD, off16(RS)	RD = unaligned_mem32(RS + off16±)
USW	RS, off16(RT)	unaligned_mem32(RT + off16±) = RS

Atomic Read-Modify-Write Operations		
LL	RD, off16(RS)	$RD = mem32(RS + off16^{\pm})$; link
SC	RD, off16(RS)	if Atomic, $mem32(RS + off16^{\pm}) = RD$; RD = Atomic ? 1 : 0

MD00565 Revision 01.01

Registers

0	zero	Always equal to zero
1	at	Assembler temporary; used by the assembler
2-3	v0-v1	Return value from a function call
4-7	a0-a3	First four parameters for a function call
8-15	t0-t7	Temporary variables; need not be preserved
16-23	s0-s7	Function variables; must be preserved
24-25	t8-t9	Two more temporary variables
26-27	k0-k1	Kernel use registers; may change unexpectedly
28	gp	Global pointer
29	sp	Stack pointer
30	fp/s8	Stack frame pointer or subroutine variable
31	ra	Return address of the last subroutine call

Default C Calling Convention (O32)

Stack Management

- The stack grows down.
  - Subtract from $sp to allocate local storage space.
  - Restore $sp by adding the same amount at function exit.
- The stack must be 8-byte aligned.
  - Modify $sp only in multiples of eight.

Function Parameters

- Every parameter smaller than 32 bits is promoted to 32 bits.
- First four parameters are passed in registers $a0–$a3.
  - 64-bit parameters are passed in register pairs:
    - Little-endian mode: $a1:$a0 or $a3:$a2.
    - Big-endian mode: $a0:$a1 or $a2:$a3.
- Every subsequent parameter is passed through the stack.
  - First 16 bytes on the stack are not used.
  - Assuming $sp was not modified at function entry:
    - The 1st stack parameter is located at 16($sp).
    - The 2nd stack parameter is located at 20($sp), etc.
  - 64-bit parameters are 8-byte aligned.

Return Values

- 32-bit and smaller values are returned in register $v0.
- 64-bit values are returned in registers $v0 and $v1:
  - Little-endian mode: $v1:$v0.
  - Big-endian mode: $v0:$v1.

MIPS32 Virtual Address Space

kseg3	0xE000.0000	0xFFFF.FFFF	Mapped	Cached
ksseg	0xC000.0000	0xDFFF.FFFF	Mapped	Cached
kseg1	0xA000.0000	0xBFFF.FFFF	Unmapped	Uncached
kseg0	0x8000.0000	0x9FFF.FFFF	Unmapped	Cached
useg	0x0000.0000	0x7FFF.FFFF	Mapped	Cached

Reading The Cycle Count Register From C

```
unsigned mips_cycle_counter_read()
{
     unsigned cc;
     asm volatile("mfc0 %0, $9" : "=r" (cc));
     return (cc << 1);
}
```

Assembly-Language Function Example

```
# int asm_max(int a, int b)
# {
#   int r = (a < b) ? b : a;
#   return r;
# }

     .text
     .set     nomacro
     .set     noreorder

     .global  asm_max
     .ent     asm_max

asm_max:
     move     $v0, $a0          # r = a
     slt      $t0, $a0, $a1     # a < b ?
     jr       $ra               # return
     movn     $v0, $a1, $t0     # if yes, r = b

     .end     asm_max
```

C / Assembly-Language Function Interface

```
#include <stdio.h>

int asm_max(int a, int b);

int main()
{
     int x = asm_max(10, 100);
     int y = asm_max(200, 20);
     printf("%d %d\n", x, y);
}
```

Invoking MULT And MADD Instructions From C

```
int dp(int a[], int b[], int n)
{
     int i;
     long long acc = (long long) a[0] * b[0];
     for (i = 1; i < n; i++)
          acc += (long long) a[i] * b[i];
     return (acc >> 31);
}
```

Atomic Read-Modify-Write Example

```
atomic_inc:
     ll       $t0, 0($a0)      # load linked
     addiu    $t1, $t0, 1      # increment
     sc       $t1, 0($a0)      # store cond'l
     beqz     $t1, atomic_inc  # loop if failed
     nop
```

Accessing Unaligned Data

note: ULW and USW automatically generate appropriate code

Little-Endian Mode		Big-Endian Mode	
LWR	RD, OFF16(RS)	LWL	RD, OFF16(RS)
LWL	RD, OFF16+3(RS)	LWR	RD, OFF16+3(RS)
SWR	RD, OFF16(RS)	SWL	RD, OFF16(RS)
SWL	RD, OFF16+3(RS)	SWR	RD, OFF16+3(RS)

Accessing Unaligned Data From C

```
typedef struct
{
     int u;
} __attribute__((packed)) unaligned;


int unaligned_load(void *ptr)
{
     unaligned *uptr = (unaligned *)ptr;
     return uptr->u;
}
```

MIPS SDE-GCC Compiler Defines

__mips	MIPS ISA (= 32 for MIPS32)
__mips_isa_rev	MIPS ISA Revision (= 2 for MIPS32 R2)
__mips_dsp	DSP ASE extensions enabled
_MIPSEB	Big-endian target CPU
_MIPSEL	Little-endian target CPU
_MIPS_ARCH_*CPU*	Target CPU specified by `-march=`*CPU*
_MIPS_TUNE_*CPU*	Pipeline tuning selected by `-mtune=`*CPU*

Notes

- Many assembler pseudo-instructions and some rarely used machine instructions are omitted.
- The C calling convention is simplified. Additional rules apply when passing complex data structures as function parameters.
- The examples illustrate syntax used by GCC compilers.
- Most MIPS processors increment the cycle counter every other cycle. Please check your processor documentation.

3.4. CONCLUSION

Ce chapitre offre un bref aperçu de l'organisation du jeu d'instructions du PIC32.
Il doit permettre à l'étudiant de pouvoir observer le code assembleur produit par le compilateur et parvenir à le comprendre dans les grandes lignes.

3.5. HISTORIQUE DES VERSIONS

3.5.1. VERSION 1.0 JANVIER 2014

Création du document et découverte du jeu d'instructions MIPS32.

3.5.2. VERSION 1.5 NOVEMBRE 2014

Passage à la version 1.5 pour cohérence avec l'ensemble des chapitres. Pas de modifications liées à Harmony. Quelques retouches.

3.5.3. VERSION 1.7 NOVEMBRE 2015

Saut à la version 1.7 pour cohérence avec l'ensemble des chapitres. Pas de modifications liées à Harmony. Correction de la numérotation des titres.

3.5.4. VERSION 1.8 NOVEMBRE 2016

Saut à la version 1.8 pour cohérence avec l'ensemble des chapitres. Pas de modifications liées à Harmony. Modification du chemin de la documentation liée au Kit PIC32.

3.5.5. VERSION 1.9 NOVEMBRE 2017

Reprise et relecture par SCA.