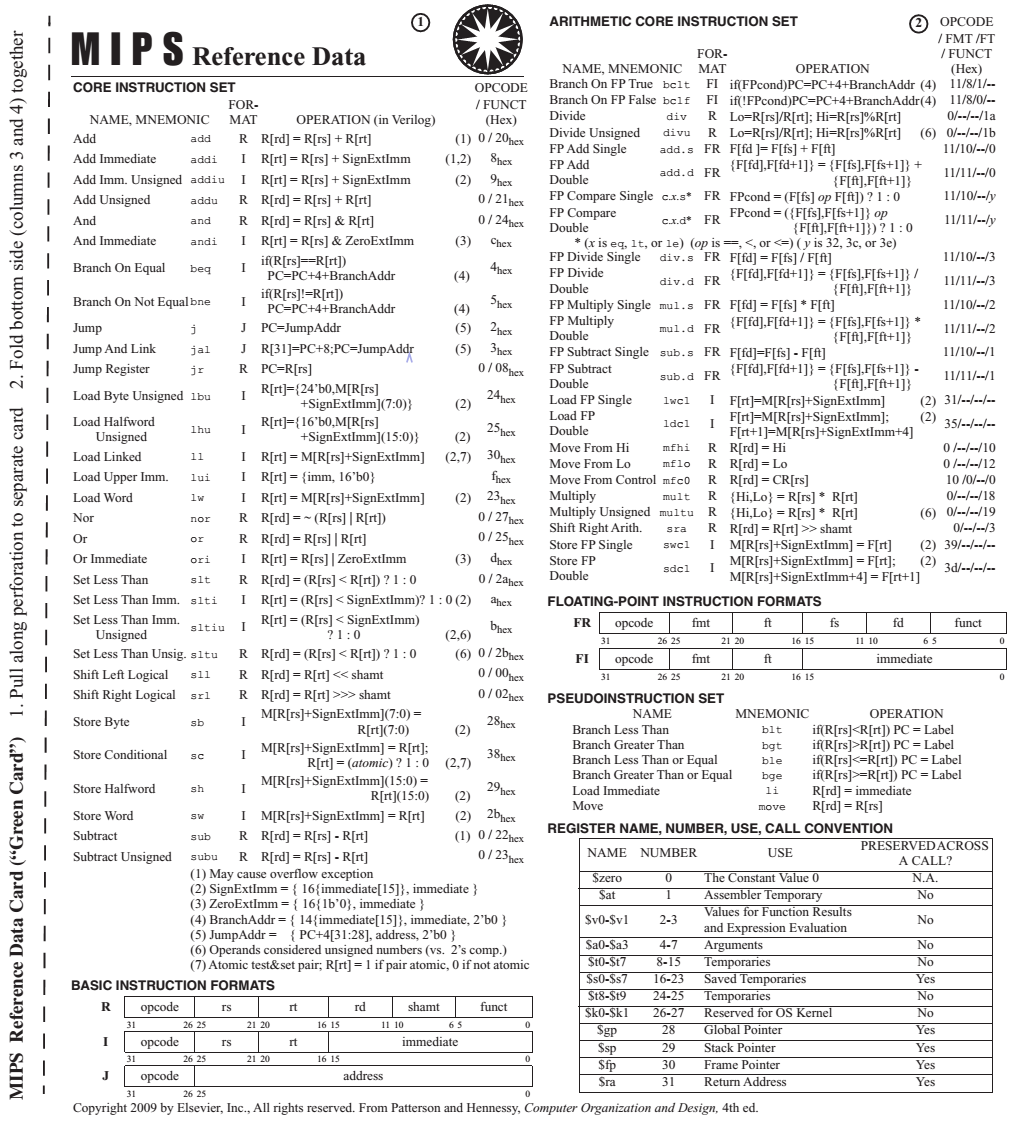
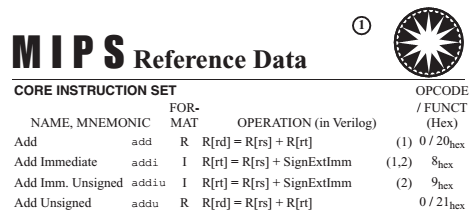
**Một số lệnh assembly MIPS cơ bản**



**Bảng 1. Tóm tắt các lệnh MIPS cơ bản (tham khảo [1])**

**Các lệnh assembly MIPS trong tài liệu này sẽ được diễn tả theo từng hàng trong bảng 1**

****

Một số ghi chú cho lệnh tương ứng, được làm rõ ở cuối bảng

opcode và funct cho từng lệnh tương ứng.

Ví dụ: lệnh add có số ở cột này là 0/20hex, tức opcode của add là 0; trường funct trong R-format của add là 20hex

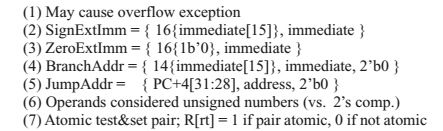
Lệnh thực hiện chức năng gì

Chức năng của từng lệnh được diễn tả theo kiểu viết của Verilog. Verilog là ngôn ngữ lập trình dùng để mô tả thiết kế phần cứng (sinh viên năm 1, 2, 3 chưa học).

Lệnh thuộc nhóm gì (R, I hay J)

Tên lệnh

Tên lệnh (đầy đủ)



1. **May cause overflow exception**

Những lệnh có phần ghi chú (1) sẽ một thông báo lỗi, hay còn gọi là gây ra một ngoại lệ (exception) khi phép toán bị tràn (overflow)

1. **SignExtImm = {16 {immediate[15]}, immediate}**

Những lệnh có phần ghi chú (2) luôn chứa một số tức thời 16 bits (có dấu dạng bù 2), và số này được mở rộng thành số 32 bits theo kiểu mở rộng có dấu.

*Viết theo cấu trúc của Verilog*

16{immediate[15]}: là một chuỗi 16 bits; 16 bit này được tao ra giống y như bit thứ 15 của immediate

{16{immediate[15]}, immediate}: là chuỗi 32 bits, 16 bit thuộc nữa cao được tao ra giống như bit thứ 15 của immediate, và 16 bit thuộc nữa thấp chính là số tức thời

Ví dụ:

* SignExtImm của 0011 1110 1101 1100 là 0000 0000 0000 0000 0011 1110 1101 1100
* SignExtImm của 1011 1110 1101 1100 là 1111 1111 1111 1111 1011 1110 1101 1100
* Có thể hiểu đơn giản, nếu số tức thời là dương thì 16 bits của nữa cao thêm vào sẽ là 0, còn nếu số tức thời là âm, thì 16bits của nữa cao thêm vào sẽ là 1

1. **ZeroExtImm = {16{1b’0}, immediate}**

Những lệnh có phần ghi chú (3) luôn chứa một số tức thời 16 bits (có dấu dạng bù 2), và số này được mở rộng thành số 32 bits theo kiểu mở rộng Zero, tức không cần biết đây là âm hay dương, 16 bits của nữa cao thêm vào đều là 0.

*Viết theo cấu trúc của Verilog*

16{1b’0}: là một chuỗi 16 bits mà tất cả các bit đều là 0

{16{1b’0}, immediate}: là chuỗi 32 bits, 16 bit thuộc nữa cao là 0 và 16 bit thuộc nữa thấp chính là số tức thời

Ví dụ:

* SignExtImm của 0011 1110 1101 1100 là 0000 0000 0000 0000 0011 1110 1101 1100
* SignExtImm của 1011 1110 1101 1100 là 0000 0000 0000 0000 1011 1110 1101 1100

1. **BranchAddr = {14{immediate[15]}, immediate, 2’b0}**

sẽ được giải thích trong phần lệnh *beq* và *bne*

1. **JumpAddr = { PC + 4[31:28], address, 2’b0}**

sẽ được giải thích trong phần lệnh *j* và *jal*

1. **Operations considered unsigned numbers (vs. 2’comp.)**

Những lệnh có phần ghi chú (6) luôn làm việc trên số không dấu (unsigned)

1. **Atomic test&set pair; R[rt] = 1 if pair atomic, 0 if not atomic**

Trong bảng 1, chỉ có 2 lệnh *ll* và *sc* là có ghi chú (7)

* 2 lệnh này liên quan đến một số lý thuyết không nằm trong phần giảng dạy lý thuyết, vì vậy bỏ qua 2 lệnh này

**Một số ghi chú:**

**Ký hiệu số:**

* Ký hiệu 0x đầu được dùng để chỉ hệ 16

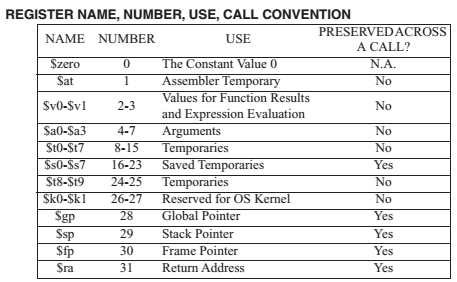
Ví dụ: 0xffff = ffffhex = ffff(16)

* Số ghi bình thường sẽ được hiểu là đang trong hệ 10

**Thanh ghi:**

* Bộ xử lý chứa 32 thanh ghi để hoạt động, mỗi thanh ghi 32 bits.
* Mỗi thanh ghi sẽ có tên gợi nhớ và số thứ tự tương ứng của nó. Bảng 2 mô tả số thứ tự và tên gợi nhớ của từng thanh ghi
* Như vậy, khi làm việc với thanh ghi có 2 vấn đề cần quan tâm: giá trị và địa chỉ
* Giá trị là giá trị đang được chứa trong thanh ghi
* Địa chỉ là chỉ số của thanh ghi trong tập 32 thanh ghi.

Ví dụ: Nếu nói thanh ghi $t3 có giá trị là 5, hoặc thanh ghi $t3 bằng 5, tức giá trị đang chứa trong $t3 là 5 và chỉ số/địa chỉ của $t3 là 11



Bảng 2. Mô tả các thanh ghi (trích từ bảng 1)

Thanh ghi nào “Yes” là những thanh ghi cần được lưu trữ lại khi thực hiện việc gọi một hàm con

Mục đích sử dụng của từng thanh ghi

Tên gợi nhớ của các thanh ghi

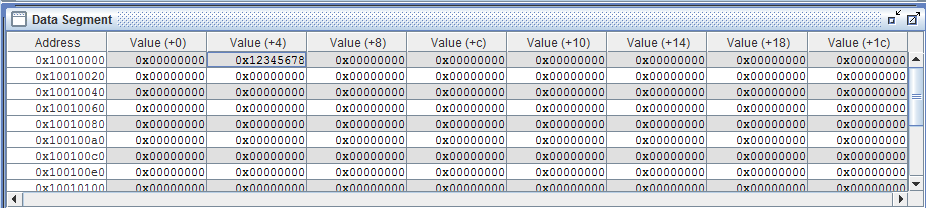
Chỉ số tương ứng của các thanh ghi

**Bộ nhớ:**

Tương tự như thanh ghi, khi làm việc với bộ nhớ có 2 vấn đề cần quan tâm: giá trị và địa chỉ

* Giá trị là giá trị đang được chứa trong một từ nhớ (word), hoặc trong byte
* Địa chỉ địa chỉ được gán cho word hoặc byte đó.

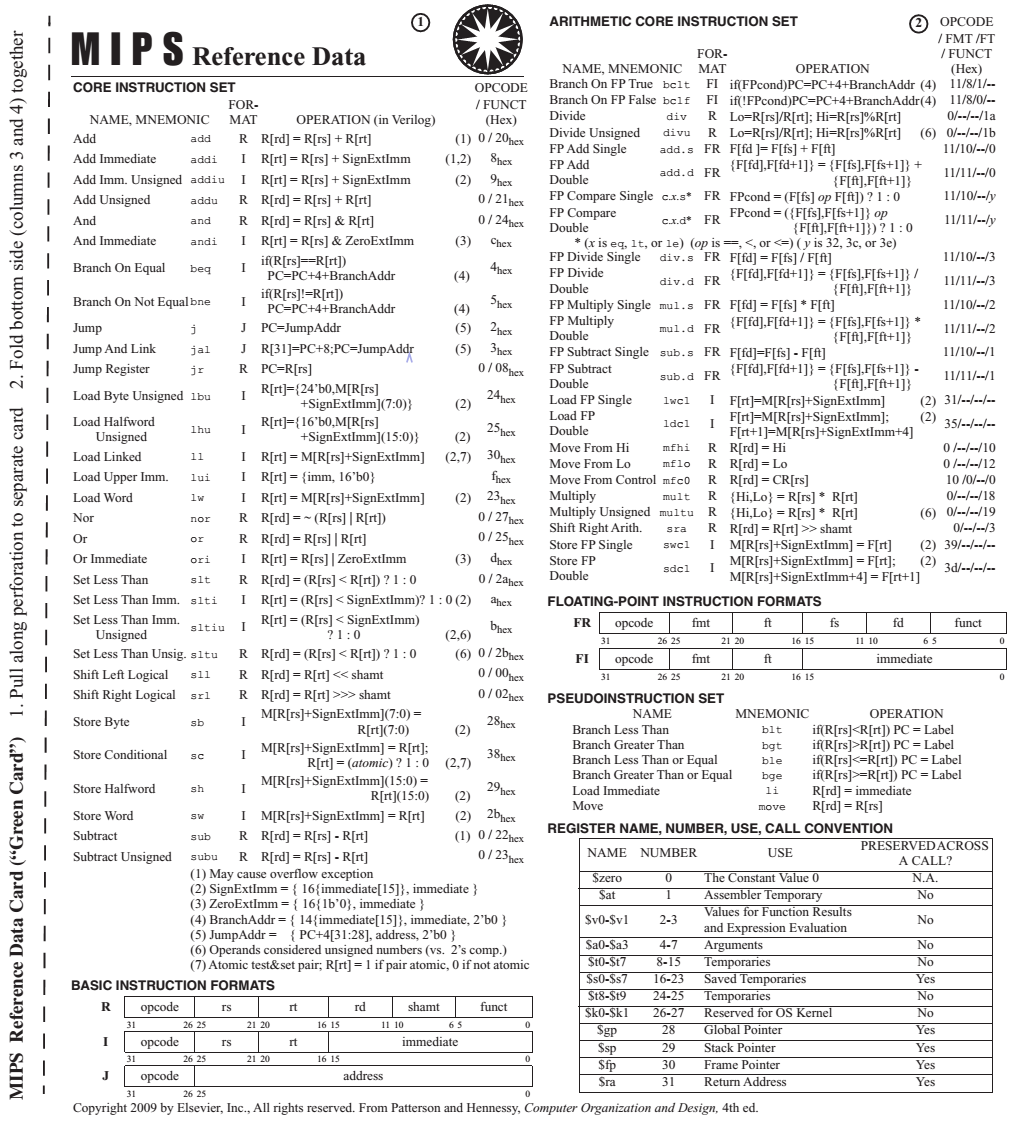
Ví dụ:



Hình 3. Một ví dụ về hình ảnh bộ nhớ từ phần mềm mô phỏng (simulator) MARS 4.4

Đây là word (4 bytes) tại địa chỉ 0x10010004, và có giá trị là 0x12345678

1. **Xét các lệnh số học**

****

**Các lệnh số học:**

* *add, addi, addiu, addu*
* *sub, subu*

R viết tắt của Register

Ví dụ: R[rs] hiểu là giá trị của thanh ghi rs

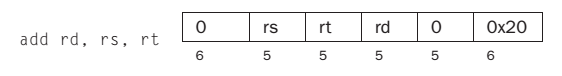
***--------------------------------------------***

1. **Lệnh *add***



* Lệnh này thuộc dạng R-format, có opcode là 0 và trường funct giá trị là 20hex

**Syntax (cú pháp): (tham khảo Appendix B của sách tham khảo [1])**



**Ý nghĩa: R[rd] = R[rs] + R[rt]**

Thực hiện cộng giá trị thanh ghi rs với giá trị thanh ghi rt, tổng đưa vào thanh ghi rd

**Ví dụ:**

*add $t0, $t1, $t2*

Giả sử giá trị đang chứa trong thanh ghi $t1 là 3, giá trị đang chứa trong thanh ghi $t2 là 4

**Kết quả:** Sau khi lệnh add trên thực hiện, giá trị trong thanh ghi $t0 là 7 (4 + 3 = 7).

1. **Lệnh *addi***



* Lệnh này thuộc dạng I-format, có opcode 8hex

**Syntax (cú pháp):**



**Ý nghĩa: R[rt] = R[rs] + SignExtImm**

Thực hiện cộng giá trị thanh ghi rs với số tức thời, kết quả đưa vào thanh ghi rt.

Lưu ý: Phạm vi cho số tức thời trong lệnh này là 16 bits. Số tức thời trước khi cộng với thanh ghi rs phải được **mở rộng có dấu thành** (SignExtImm) thành số 32 bits.

**Ví dụ:**

1. addi $t0, $t1, 3
2. addi $t0, $t1, -3
3. addi $t0, $t1, 32768

Giả sử giá trị đang chứa trong thanh ghi $t1 cho cả 3 câu đều là 4

**Kết quả:**

1. Sau khi addi thực hiện xong, giá trị của $t0 là 7

Quy trình lệnh thực hiện:

số tức thời là 3(10) = 0000 0000 0000 0011(2) (số 16 bit có dấu)

SignExtImm của 3(10) = 0000 0000 0000 0000 0000 0000 0000 0011(2)

­Giá trị thanh ghi $t1 = 0000 0000 0000 0000 0000 0000 0000 0100(2)

Giá trị trong $t1 + SingExtImm của 3(10) = 0000 0000 0000 0000 0000 0000 0000 0111(2)

1. Sau khi addi thực hiện xong, giá trị của $t0 là 1

Quy trình lệnh thực hiện:

số tức thời là -3(10) = 1111 1111 1111 1101(2) (số 16 bit có dấu, biểu diễn theo bù 2)

SignExtImm của 3(10) = 1111 1111 1111 1111 1111 1111 1111 1101 (2)

­Giá trị thanh ghi $t1 = 0000 0000 0000 0000 0000 0000 0000 0100(2)

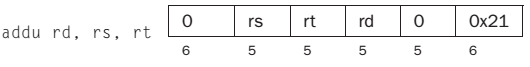
Giá trị trong $t1 + SingExtImm của 3(10) = 0000 0000 0000 0000 0000 0000 0000 0001(2)

1. Lệnh bị báo lỗi, do 32768 ra khỏi giới hạn của số 16 bits có dấu
2. **Lệnh *addiu* và *addu***

* *Addiu* có cú pháp và thực hiện chức năng giống *addi*
* *Addu* có cú pháp và thực hiện chức năng giống *add*

Tuy nhiên, *addiu* và *addu* không xét kết quả có bị overflow hay không, trong khi đó *addi* và add sẽ báo khi overflow xuất hiện

**Syntax (cú pháp):**

****

**Ví dụ:**

1. *addi $t0, $t1, 0x1*

Giả sử thanh ghi $t1 = 0x7fffffff

**Kết quả:**

0x1 + 0x7fffffff = 0x80000000

Cộng môt số dương với một số dương, kết quả ra một số âm => overflow

Khi lệnh addi trên thực hiện, một thông báo overflow sẽ xuất hiện

1. *addiu $t0, $t1, 1*

Giả sử thanh ghi $t1 = 0x7fffffff

**Kết quả:** $t0 = 0x80000000

Khi lệnh addi trên thực hiện, thanh ghi $t0 vẫn nhận giá trị 0x8000000 và không có bất kì thông báo overflow nào xuất hiện.

1. **Lệnh *sub* và *subu***

Lệnh *sub* có cú pháp tương tư như lệnh *add*, nhưng

* ***add*** thực hiện phép toán **cộng** 2 thanh ghi, kết quả lưu vào thanh ghi thứ 3
* trong khi đó, ***sub*** thực hiện phép toán **trừ** 2 thanh ghi, kết quả lưu vào thanh ghi thứ 3

Lệnh *subu* có cú pháp và chức năng giống như *sub*, nhưng

* ***subu*** không xét đến kết quả có bị overflow hay không
* ***sub*** có xét đến kết quả có bị overflow hay không; nếu bị overflow, sẽ có thông báo

**Syntax (cú pháp):**





***Lưu ý: không có lệnh subi (tức trừ một thanh ghi với một số tức thời) vì đã có lệnh addi và số tức thời trong addi có thể âm hoặc dương, nên subi không cần thiết.***

**Tổng kết:**

* *add, addi, addiu, addu*
* *sub, subu*

Nhìn lại côt ghi chú của 6 lệnh trên trong bảng 1:

* Chỉ có lệnh addi và addiu có ghi chú (2) 🡪 tức 2 lệnh này làm việc với số tức thời, và số tức thời 16 bits này được mở rộng có dấu thành thành số 32 bits

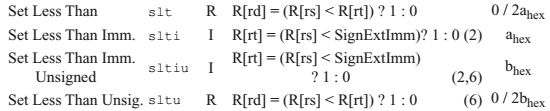
(có ‘i’ 🡪 làm việc với số tức thời)

* Các lệnh không có “u” theo sau: add, addi, sub có thêm ghi chú (1); Các lệnh có “u” theo sau như: addiu, addu và subu không có ghi chú (1) 🡪 tức các lệnh không có “*u*” sẽ báo khi có overflow, còn các lệnh có “u” sẽ không báo khi có overflow
* **Nhóm lệnh so sánh**

*slt / sltu*

*slti / sltiu*

***----------------------------------------------------------***



1. **Lệnh slt/sltu**





* Hai lệnh này thuộc nhóm lệnh R-format, có opcode là 0 và funct trong slt là 2ahex, trong sltu là 2bhex

**Syntax:**

****

****

**Ý nghĩa:**

**slt:** R[rd] = (R[rs] < R[rt])? 1 : 0

Kiểm tra xem giá trị trong thanh ghi rs có nhỏ hơn thanh ghi rt hay không, nếu nhỏ hơn thì thanh ghi rd nhận giá trị là 1; ngược lại thanh ghi rd sẽ nhận giá trị 0

**sltu:** Ý nghĩa thực hiện giống như slt. Nhưng việc kiểm tra giá trị thanh ghi rs có nhỏ hơn thanh ghi rt hay không trong lệnh slt thực hiện trên số có dấu, còn trong sltu thực hiện trên số không dấu

**Ví dụ:**

1. slt $t0, $t1, $t2

Giả sử $t1 = 0xfffffff1, $t2 = 0x00000073

**Kết quả: $t0 = 1**

Lệnh slt so sánh theo kiểu so sánh 2 số có dấu dạng bù 2

$t1 = 0xfffffff1 = 1111 1111 1111 1111 1111 1111 1111 0001(2) = -15(10)

$t2 = 0x00000073 = 01110011(2) = 115(10)

Vậy $t1 < $t2 🡪 giá trị trong thanh ghi $t0 = 1

1. sltu $t0, $t1, $t2

Giả sử $t1 = 0xfffffff1, $t2 = 0x00000073

**Kết quả: $t0 = 0**

Lệnh slt so sánh theo kiểu so sánh 2 số không dấu

$t1 = 0xfffffff1 = 1111 1111 1111 1111 1111 1111 1111 0001(2) = 4294967281(10)

$t2 = 0x00000073 = 01110011(2) = 115(10)

Vậy $t1 > $t2 🡪 giá trị trong thanh ghi $t0 = 0

1. **Lệnh *slti/sltiu***





* Hai lệnh này thuộc nhóm lệnh I-format. Opcode của *slti* là ahex, opcode của *sltiu* là bhex

**Syntax:**

****

****

**Ý nghĩa:**

***slti/sltiu:*** R[rd] = (R[rs] < SignExtImm)? 1 : 0

Ý nghĩa 2 lệnh này giống nhau là so sánh giá trị một thanh ghi với một số tức thời, nếu giá trị trong thanh ghi rs nhỏ hơn số tức thời thì thanh ghi rd nhận giá trị là 1; ngược lại thanh ghi rd sẽ nhận giá trị 0

Số tức thời cho phép trong lệnh này là số 16 bits. Trước khi so sánh với thanh ghi rs, số tức thời được mở rộng có dấu (SignExtImm) thành số 32 bits

*slti* khác *sltiu* là slti so sánh 2 giá trị theo kiểu có dấu dạng bù 2, trong khi đó sltiu so sánh theo kiểu số không dấu

**Ví dụ:**

1. slti $t0, $t1, 0x73

Giả sử $t1 = 0xfffffff1

**Kết quả: $t0 = 1**

Lệnh slt so sánh theo kiểu so sánh 2 số có dấu dạng bù 2

$t1 = 0xfffffff1 = 1111 1111 1111 1111 1111 1111 1111 0001(2) = -15(10)

Số tức thời = 0x73 = 01110011(2)

SignExtImm(0x73) = 0000 0000 0000 0000 0000 0000 0111 0011(2) = 115(10)

Vậy $t1 < $t2 🡪 giá trị trong thanh ghi $t0 = 1

1. sltiu $t0, $t1, 0x83

Giả sử $t1 = 0xfffffff1

**Kết quả: $t0 = 0**

Lệnh slt so sánh theo kiểu so sánh 2 số không dấu

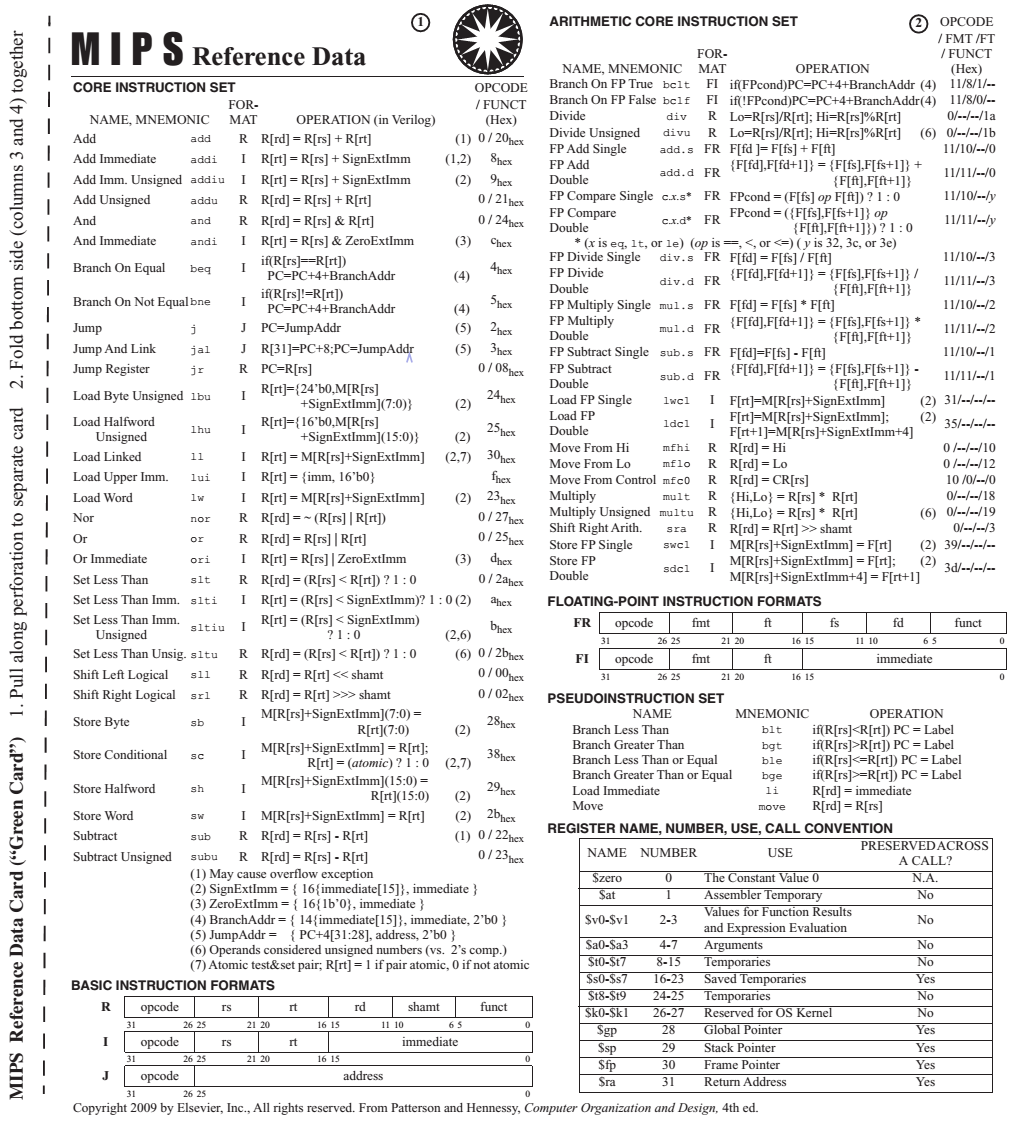
$t1 = 0xfffffff1 = 1111 1111 1111 1111 1111 1111 1111 0001(2) = 4294967281(10)

$t2 = 0x83 = 10000011(2)

SignExtImm(0x83) = 1111 1111 1111 1111 1111 1111 1000 0011(2) = 4294967171(10)

Vậy $t1 > $t2 🡪 giá trị trong thanh ghi $t0 = 0

1. **Các lệnh logic**

****

**Nhóm lệnh logic:**

***and, andi***

***nor***

***or, ori***

***sll, srl***

***----------------------------------------------------------***

1. **Lệnh *and***



* Lệnh này thuộc dạng R-format, có opcode là 0 và trường funct là 24hex

**Syntax (cú pháp):**



**Ý nghĩa: R[rd] = R[rs] & R[rt]**

Thực hiện *and* từng bit giá trị của thanh ghi rs và rt với nhau, kết quả lưu vào thanh ghi rd

**Ví dụ:**

*and $t0, $t1, $t2*

Giả sử giá trị đang chứa trong thanh ghi $t1 là 0x12345678, giá trị đang chứa trong thanh ghi $t2 là 0x0000000f thì

**Kết quả**: sau lệnh add trên, giá trị trong thanh ghi $t0 là 0x00000008.

1. **Lệnh *andi***



* Lệnh này thuộc dạng I-format, có opcode là 0xc

**Syntax (cú pháp):**



**Ý nghĩa: R[rt] = R[rs] & ZeroExtImm**

Lệnh này thực hiện *and* từng bit giá trị thanh ghi *rs* và một số tức thời. Số tức thời đang là số 16 bits, mở rộng thành số 32 bits theo kiểu ZeroExtImm, tức 16 bits nữa cao còn thiếu sẽ điền 0 vào. Sau đó thực hiện *and* từng bit giá trị của thanh ghi *rs* và số tức thời đã được mở rộng thành 32 bits với nhau, kết quả lưu vào thanh ghi *rd*

**Ví dụ:**

1. *andi $t0, $t1, 0xffff*

Giả sử giá trị đang chứa trong thanh ghi $t1 là 0x12345678.

Kết quả: sau lệnh trên, giá trị thanh ghi $t0 = 0x00005678

Quy trình lệnh thực hiện:

Số tức thời: 0xffff = 1111 111111111111 (2)

ZeroExtImm(0xffff) = 0000 0000 0000 0000 1111 111111111111 (2)

$t0 = $t1 & ZeroExtImm(0xffff) = 0x00005678

1. *andi $t0, $t1, -3*

Vấn đề đặt ra là imm ở đây có thể là số âm không?

* Một số simulator chấp nhận imm có thể là âm, ví dụ số -3 trên sẽ chuyển sang thành bù 2 của số 16 bits, sau đó mở rộng theo kiểu ZeroExtImm
* Một số simulator không chấp nhận imm có thể là âm, ví dụ số -3 trên đưa vào sẽ báo lỗi
* Trong phạm vi môn học, chọn trường hợp thứ 2, không chấp nhập imm là âm

1. **Các lệnh or, ori, nor**

*or* và *nor* cách viết tương tự như *and*, nhưng thay vì thực hiện phép toán *and*, 2 lệnh này sẽ thực hiện phép toán *or* hoặc *nor* cho từng bit trong 2 thanh ghi, kết quả lưu vào thanh ghi thứ 3

*ori* tương tự như *andi*, thực hiện *or* một thanh ghi và một số tức thời 16 bits được mở rộng ZeroExtImm thành 32 bits

1. **Lệnh sll/srl**

* **sll**

****

* lệnh dịch trái số học, thuộc nhóm lệnh R, có opcode là 0 và funct 00hex

****

* lệnh dịch phải số học, thuộc nhóm lệnh R, opcode là 0 và funct là 02hex

**Syntax (cú pháp):**





**Ý nghĩa:**

**sll: R[rd] = R[rt] << shamt**

Thanh ghi rt dịch trái shamt bit và kết quả lưu vào thanh ghi rd ( ‘<< ‘ là ký hiệu của phép toán dịch trái)

**srl: R[rd] = R[rt] >>> shamt**

Thanh ghi rt dịch phải shamt bit và kết quả lưu vào thanh ghi rd ( ‘>>> ‘ là ký hiệu của phép toán dịch phải)

**Ví dụ:**

1. *sll $t0, $t1, 5*

Giả sử thanh ghi $t1 đang chứa giá trị 0x12345678

Kết quả: sau lệnh trên, thanh ghi $t0 = 0x468ACF00

Quy trình lệnh thực hiện: lệnh trên dịch trái 5 bit thanh ghi $t1

$t1 = 0x12345678 = 0001 0010 0011 0100 0101 0110 0111 1000(2)

Dịch trái 5 bit $t1 = 0100 0110 1000 1010 1100 1111 0000 0000(2) = 0x468ACF00

Vậy kết quả thanh ghi $t0 = 0x468ACF00

1. *srl $t0, $t1, 5*

Giả sử thanh ghi $t1 đang chứa giá trị 0x12345678

Kết quả: sau lệnh trên, thanh ghi $t0 = 0x91A2B3

Quy trình lệnh thực hiện: lệnh trên dịch phải 5 bit thanh ghi $t1

$t1 = 0x12345678 = 0001 0010 0011 0100 0101 0110 0111 1000(2)

Dịch phải 5 bit $t1 = 0000 0000 1001 0001 1010 0010 1011 0011(2) = 0x91A2B3

Vậy kết quả thanh ghi $t0 = 0x91A2B3

**Tổng kết:**

Các lệnh trong nhóm:

***and, andi***

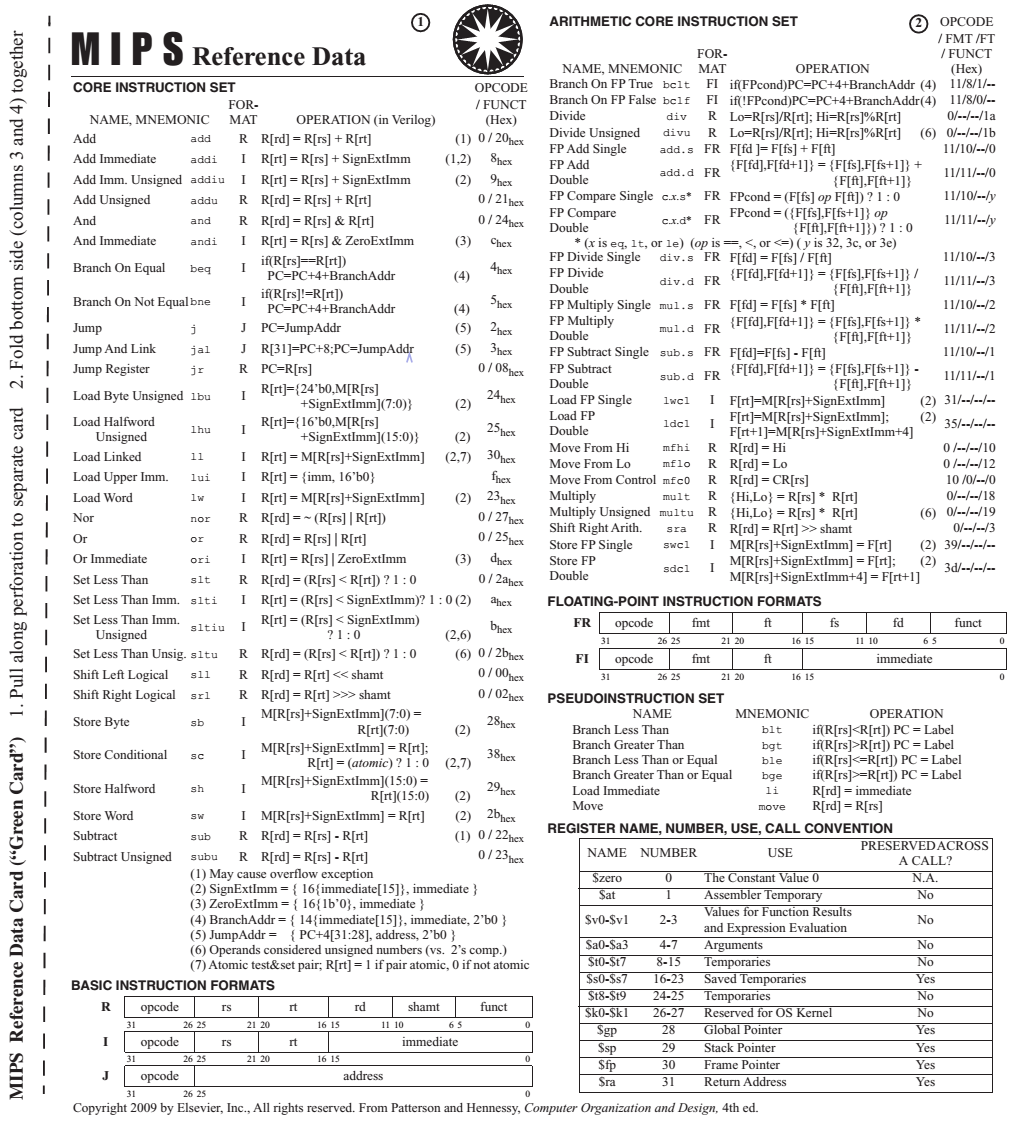
***nor***

***or, ori***

***sll, srl***

Trong cột ghi chú ở bảng 1, chú ý chỉ có 2 lệnh *andi* và *ori* có ghi chú (3) – ghi chú ‘zeroExtImm’, tức các lệnh làm việc với số tức thời trong nhóm này khi mở rộng từ số tức thời 16 bits sang số 32 bits thì dùng zeroExtImm, không phải SignExtImm như nhóm lệnh số học.

1. **Nhóm lệnh Nhánh/Nhảy (Branch/Jump)**

****

**Các lệnh xem xét:**

***beq, bne***

***j, jal, jr***

**Ngoài ra còn các lệnh thuộc bảng “PsedoInstruction Set”**

***blt/blte***

***bgt/bgte***

*Chú ý:*

*Các lệnh assembly có thể chia vào 2 nhóm: nhóm lệnh thật và nhóm lệnh giả*

* *Nhóm lệnh thật: là các lệnh mà thực chất processor sẽ chạy đúng lệnh đó*
* *Nhóm lệnh giả: là các lệnh mà khi thực thi thật sự thì lệnh này được chuyển thành một hoặc một số lệnh thuộc nhóm lệnh thật (nhóm lệnh này được đặt ra để thuận tiện cho người lập trình)*

*Các lệnh thuộc nhóm lệnh “PsedoInstruction Set” là những lệnh giả.*

**--------------------------------------------**

1. **Lệnh *beq***

****

* lệnh này thuộc nhóm lệnh I-format, có opcode 4hex

**Syntax (cú pháp):**



Lệnh beq có 2 cách viết cho vị trí ‘label’, ‘label’ có thể là một nhãn được viết bằng chữ, hoặc có thể là số

|  |  |
| --- | --- |
| Ví dụ 1: ‘label’ viết bằng chữ:  Chạy đoạn chương trình sau:  beq $t1, $t2, **label\_A**  add $s0, $t3, $t4  addi $s1, $t5, 3  **label\_A**: or $t1, $t2, $t3  sub $t3, $t4, $t5   * lệnh beq kiểm tra giá trị của $t1 và $t2, nếu: * 2 thanh ghi này bằng nhau, thì lệnh tiếp theo được thực hiện là lệnh “*or $t1, $t2, $t3*”. Sau khi “or” thực hiện xong thì luồng lệnh theo sau đó sẽ được thực hiện (ví dụ lệnh *sub* tiếp theo sau sẽ được thực hiện) * 2 thanh ghi này không bằng nhau, thì lệnh tiếp theo được thực hiện là lệnh “*add $s0, $t3, $t4*”. Sau khi “*add*” thực hiện xong thì luồng lệnh theo sau đó sẽ được thực hiện (ví dụ chuỗi các lệnh *addi*, *or*, *sub* tiếp theo sau sẽ được thực hiện) | Ví dụ 2: ‘label’ viết bằng số:  Chạy đoạn chương trình sau:  beq $t1, $t2, **2**  add $s0, $t3, $t4  addi $s1, $t5, 3  or $t1, $t2, $t3  sub $t3, $t4, $t5   * Trong ví dụ này, **số 2 thay cho label\_A**   Lệnh beq kiểm tra giá trị của $t1 và $t2, nếu:   * 2 thanh ghi này bằng nhau, thì lệnh tiếp theo được thực hiện **là lệnh cách beq 2 lệnh**, tức là lệnh“*or $t1, $t2, $t3*”. Sau khi “*or*” thực hiện xong thì luồng lệnh theo sau đó sẽ được thực hiện (ví dụ lệnh *sub* tiếp theo sau sẽ được thực hiện) * 2 thanh ghi này không bằng nhau, thì lệnh tiếp theo được thực hiện là lệnh “*add $s0, $t3, $t4*”. Sau khi “*add*” thực hiện xong thì luồng lệnh theo sau đó sẽ được thực hiện (ví dụ chuỗi các lệnh *addi, or, sub* tiếp theo sau sẽ được thực hiện) |

* Khi lập trình, ta có thể sử dụng một trong 2 cách như 2 ví dụ trên. Nhưng thực tế lệnh mà bộ xử lý hiểu là lệnh như ví dụ 2. Khi ta lập trình theo như ví dụ 1 thì lệnh cũng sẽ được chuyển về như ví dụ 2 để gởi cho bộ xử lý.

Như vậy beq chuẩn theo dạng:

**beq rs, rt, label/imm**

****

Số tức thời label/imm này chính là số lệnh mà lệnh beq hiện tại cách lệnh sẽ nhảy tới bao nhiêu, được lưu vào 16 bits của offset

**Ý nghĩa:**

**if(R[rs] = = R[rt]) PC = PC + 4 + BranchAddr**

* Nếu giá trị thanh ghi *rs* bằng *rt* thì chương trình nhảy tới lệnh mà cách lệnh beq đang xét là ***imm*** lệnh, tức địa chỉ con trỏ/thanh ghi PC sẽ chuyển thành **PC + 4 + imm\*4** (đối với trường hợp mỗi lệnh lưu trong một word 4 bytes) = PC + 4 + BranchAddr

BranchAddr = imm \* 4 (đối với trường hợp mỗi lệnh lưu trong một word 4 bytes)

1. **Lệnh bne:**

Cách viết tương tự như beq, nhưng ý nghĩa trái ngược:

* beq: kiểm tra nếu 2 thanh ghi **bằng nhau** thì nhảy đến lệnh mong muốn
* bne: kiểm tra nếu 2 thanh ghi **không bằng nhau** thì nhảy đến lệnh mong muốn

1. **Lệnh bge/bgt/ble/blt**

Ngoài ra, còn một số lênh so sánh và nhảy khác (trong bảng psedoInstruction Set)

*bge $t1, $t2, label* 🡪 Nhảy tới label thực hiện lệnh nếu thanh ghi $t1 >= $t2

*bgt $t1, $t2, label* 🡪 Nhảy tới label thực hiện lệnh nếu thanh ghi $t1 > $t2

*ble $t1, $t2, label* 🡪 Nhảy tới label thực hiện lệnh nếu thanh ghi $t1 <= $t2

*blt $t1, $t2, label* 🡪 Nhảy tới label thực hiện lệnh nếu thanh ghi $t1 < $t2

1. **Lệnh *j* – lệnh nhảy không điều kiện**

****

* Lệnh thuộc nhóm lệnh J-format, có opcode 2*hex*

**Syntax (cú pháp):**



**Ví dụ:**

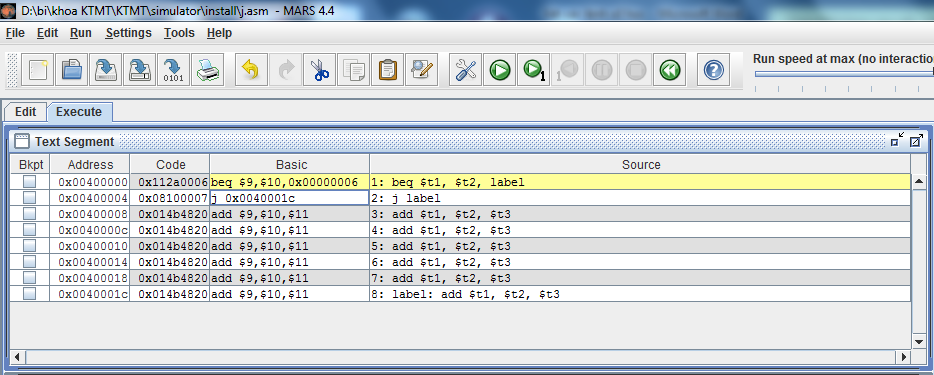
Chạy đoạn lệnh sau trên MARS 4.4:

|  |
| --- |
| *beq $t1, $t2, label*  ***j label***  *add $t1, $t2, $t3*  *add $t1, $t2, $t3*  *add $t1, $t2, $t3*  *add $t1, $t2, $t3*  *add $t1, $t2, $t3*  ***label:*** *add $t1, $t2, $t3* |

**Kết quả:** sau khi tới lệnh “j label”, chương trình sẽ nhảy đến lệnh *add* cuối cùng để thực tiếp

**Quá trình thực hiện lệnh:**

Khi biên dịch đoạn lệnh trên trong MARS 4.4, ta được hình như sau:



Cột Source là cột chứa các lệnh từ chương trình mà ta viết, cột này có thể chứa một số lệnh giả (pseudo-code); Cột basic là cột chứa các lệnh mà thực sự processor sẽ chuẩn bị chạy.

Lệnh “*j label*” khi thật sự chạy sẽ chuyển thành “*j 0x0040001c*”

*Chú ý: Lệnh này chỉ cho nhảy trong phạm vi 256 MB = 228 byte*

1. **Lệnh *jal***

****

* Lệnh này thuộc nhóm J-format, có opcode là 3hex

Sửa lại thành:

R[31] = PC + 4

**Syntax (cú pháp):**



**Ý nghĩa:**

**R[31] = PC + 4; PC = JumpAddr**

* Lệnh này thực hiện việc nhảy giống y như lệnh *j;* nhưng địa chỉ của lệnh ngay sau lệnh *jal* được lưu vào thanh ghi 31 (thanh ghi *ra)* trước khi nhảy

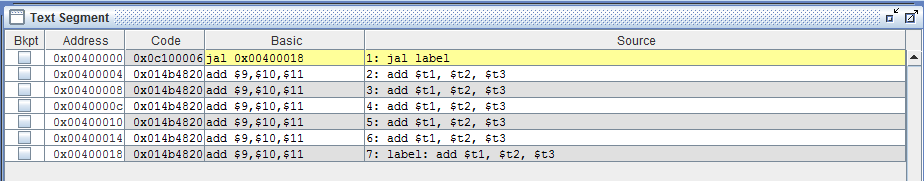
Lệnh này nhằm phục vụ cho việc gọi một hàm con. Theo quy tắc, sau khi hàm con được gọi và thực hiện xong sẽ quay trở về chương trình chính. Do đó việc lưu lại địa chỉ của lệnh ngay sau *jal* vào *ra* nhằm lưu lại địa chỉ quay về này

**Ví dụ:**

**Chạy đoạn lệnh sau trên MARS 4.4:**

|  |
| --- |
| *jal label*  *add $t1, $t2, $t3*  *add $t1, $t2, $t3*  *add $t1, $t2, $t3*  *add $t1, $t2, $t3*  *add $t1, $t2, $t3*  *label: add $t1, $t2, $t3* |

**Khi biên dịch:**



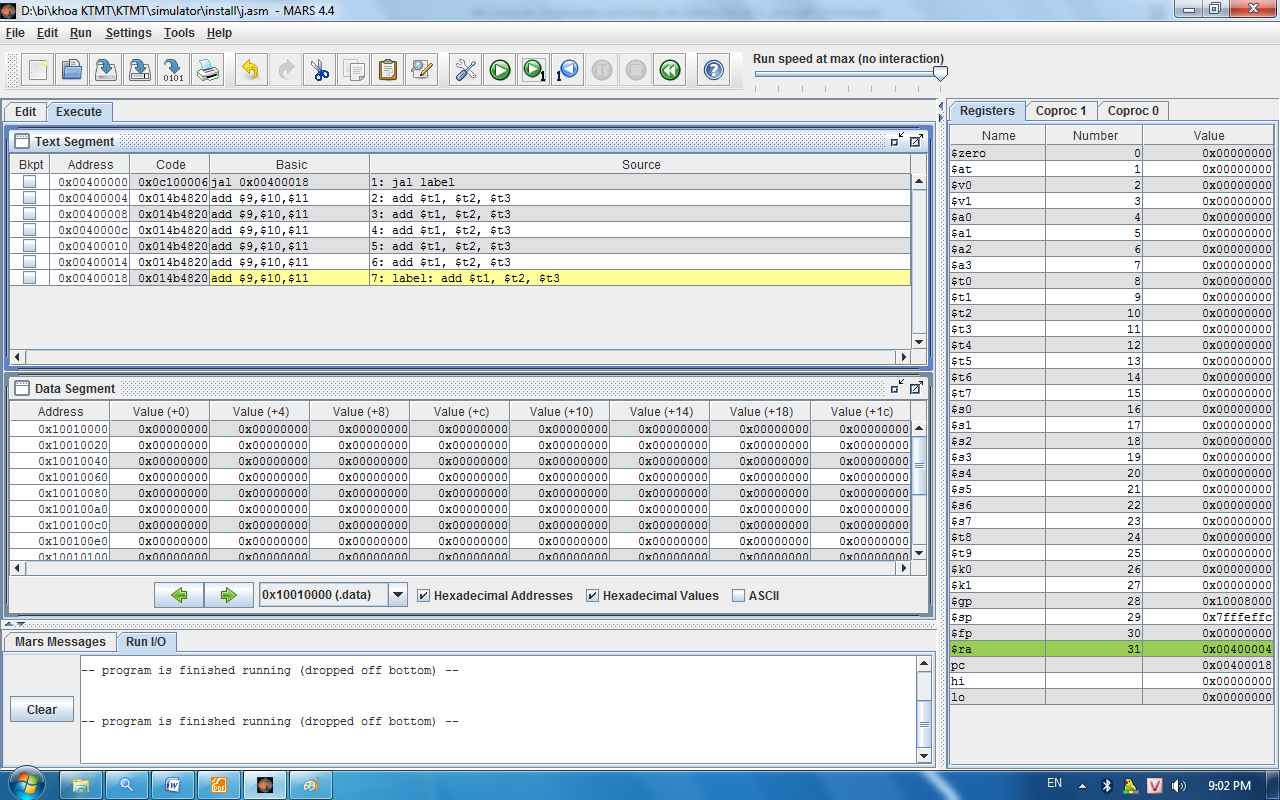
Ta thấy:

“jal label” được chuyển thành “jal 0x0040018” (giá trị ‘target’ trong format lệnh lúc này sẽ bằng 0x0040018) trước khi gởi cho processor

Địa chỉ của lệnh này hiện tại là 0x00400000

**Khi chương trình chạy:**

* PC = 0x00400000
* Đầu tiên, lệnh theo sau *jal* được lưu lại vào thanh ghi *ra 🡪 ra = PC + 4 = 0x00400004*
* Sau đó lệnh sẽ nhảy đến lệnh thứ 7, tức PC đang bằng = 0x00400000 sẽ chuyển thành PC= target/JumpAddr = 0x0040018



$ra = 0x00400004

1. **Lệnh *jr***

****

* Lệnh thuộc nhóm lệnh R (nhưng khá đặc biệt - chỉ quan tâm vùng thanh ghi rs), có opcode 0 và funct 08hex

Syntax (cú pháp):



**Ý nghĩa: PC = R[rs]**

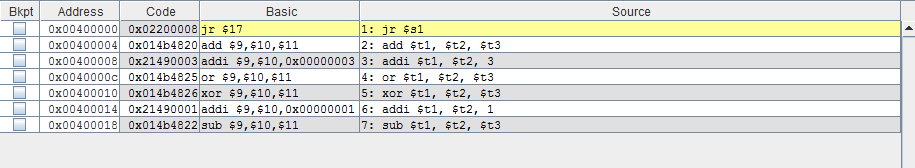
* Nhảy tới lệnh có địa chỉ đang chứa trong thanh ghi rs

**Ví dụ:**

Chạy đoạn chương trình sau trong MARS

|  |
| --- |
| *jr $s1*  *add $t1, $t2, $t3*  *addi $t1, $t2, 3*  *or $t1, $t2, $t3*  *xor $t1, $t2, $t3*  *addi $t1, $t2, 1*  *sub $t1, $t2, $t3* |

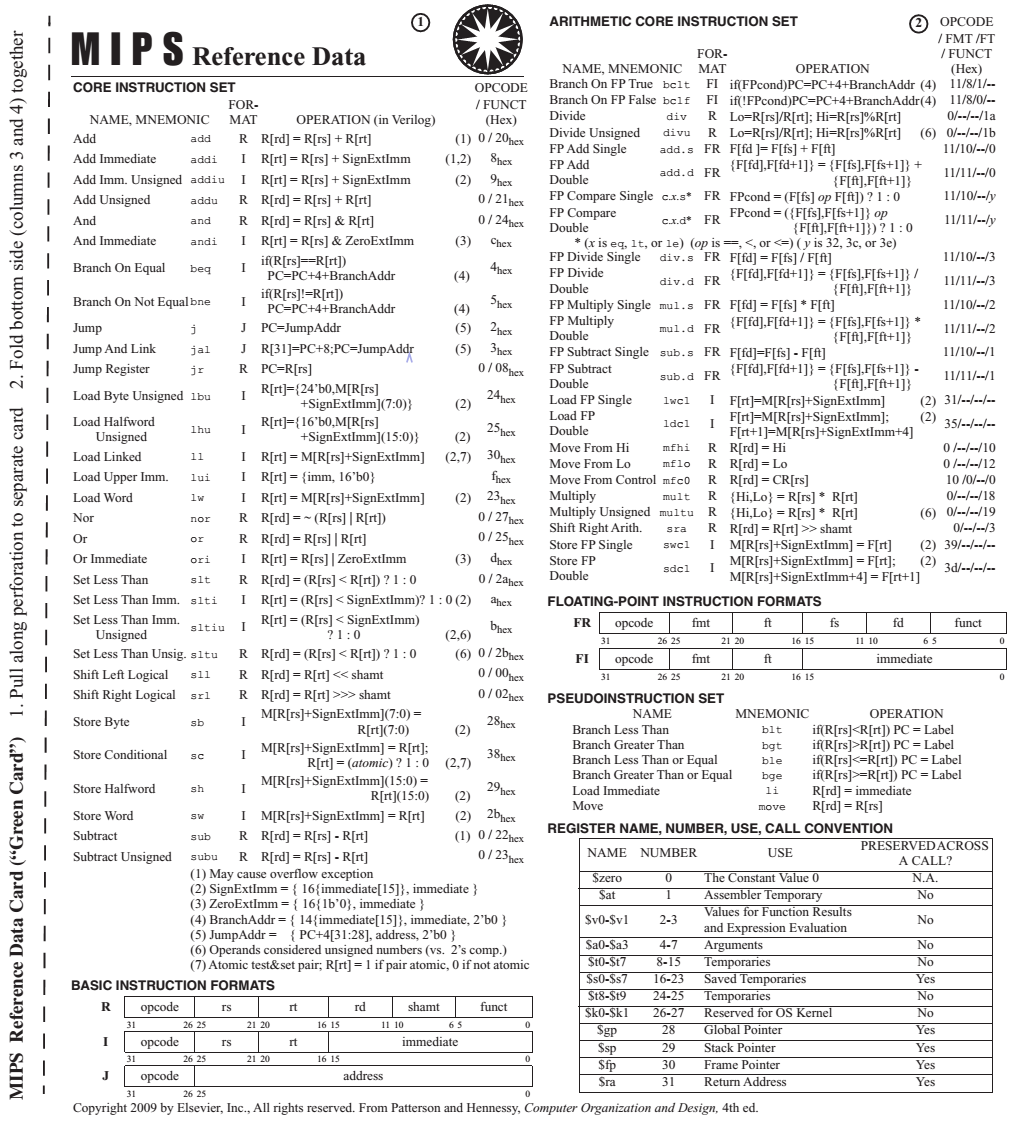
**Khi chương trình được biên dịch:**

****

Giả sử lúc này giá trị trong thanh ghi $s1 = 0x0040000c, lệnh *or* sẽ được thực hiện ngay sau *jr* khi chương trình chạy.

Giả sử lúc này giá trị trong thanh ghi $s1 = 0x00400018, lệnh *sub* sẽ được thực hiện ngay sau *jr* khi chương trình chạy.

1. **Nhóm lệnh memory-instruction**

****

**Các lệnh xét:**

**lbu, lhu, lui, lw**

**sb, sh, sw**

Chú ý: lệnh *ll* và *sc* bỏ qua

**--------------------------------------------**

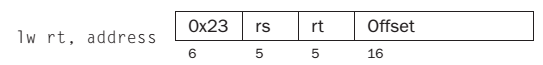
1. **Lệnh lw**

****

* Lệnh thuộc dạng I-format, có opcode = 23hex

Address = Offset(rs)

**Syntax (cú pháp):**



**Ý nghĩa: R[rt] = M[R[rs] + SingExtImm]**

* Lấy giá trị trong thanh ghi rs cộng với số tức thời đang lưu trong offset (số tức thời này này được mở rộng có dấu thanh 32 bits) ta được địa chỉ của từ nhớ cần lấy dữ liệu. Dữ liệu của từ nhớ này sẽ được lấy để lưu vào thanh ghi rt

Lưu ý:

M[X]: là lấy giá trị của từ nhớ tại địa chỉ X

**Ví dụ:**

lw $t1, 4($t0)

giả sử $t0 = 0x10010000

và từ nhớ tại địa chỉ 0x10010004 có giá trị 0x12345678

* Lệnh lw thực hiện việc load một từ nhớ (word) tại địa chỉ $t0 + 4 = 0x10010004 vào thanh ghi $t1

Kết quả: $t1 = 0x12345678

1. **Lệnh lbu/lb**

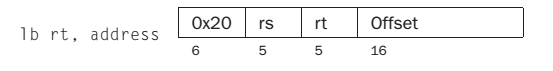
**lbu**

****

* Lệnh thuộc nhóm lệnh I-format, có opcode = 24hex

**Syntax (cú pháp):**

Address = Offset(rs)

****



Address = Offset(rs)

**Ý nghĩa:**

**R[rt] = {24’b0, M[R[rs] + SignExtImm](7:0)}**

* Không giống *lw* là load hết toàn bộ 1 từ nhớ vào thanh ghi rt, lệnh này chỉ load về 1 byte đầu tiên trong từ nhớ vào rt

Vì rt là thanh ghi 32 bits nên 24 bit còn lại của rt có 2 kiểu mở rộng dấu:

* lb: sign-ext
* lbu: zero-ext

**Ví dụ:**

1. *lbu $t1, 8($t0)*

giả sử $t0 = 0x10010000

và từ nhớ tại địa chỉ 0x10010008 có giá trị 0x12345678

Từ nhớ này chứa 4 bytes:

|  |  |  |  |
| --- | --- | --- | --- |
| Byte 4 | Byte 2 | Byte 1 | Byte 0 |
| 12 | 34 | 56 | 78 |

* Lệnh *lbu* thực hiện việc load một byte vào thanh ghi $t1, và byte được load là byte 0. Nhưng do thanh ghi $t1 là 32 bits, nên 24 bits còn lại là 0

Kết quả: $t1 = 0x00000078

1. *lb $t1, 8($t0)*

giả sử $t0 = 0x10010000

và từ nhớ tại địa chỉ 0x10010008 có giá trị 0x123456f8

Từ nhớ này chứa 4 bytes:

|  |  |  |  |
| --- | --- | --- | --- |
| Byte 4 | Byte 2 | Byte 1 | Byte 0 |
| 12 | 34 | 56 | f8 |

* Lệnh lb thực hiện việc load một byte vào thanh ghi $t1, và byte được load là byte 0. Nhưng do thanh ghi $t1 là 32 bits, nên 24 bits còn lại trong lệnh này **được mở rộng có dấu theo bit lớn nhất của byte được load về**

Kết quả: $t1 = 0xfffffff8

1. **Lệnh lhu/lh**

****

* Lệnh thuộc nhóm lệnh I-format, có opcode 25hex

Address = Offset(rs)

**Syntax (cú pháp):**





Address = Offset(rs)

**Ý nghĩa:**

**R[rt] = {16’b0, M[R[rs] + SignExtImm](15:0)}**

* Lệnh này chỉ load về 2 byte đầu tiên (nữa word thấp) trong từ nhớ vào rt

Vì rt là thanh ghi 32 bits nên 16 bit còn lại của rt có 2 kiểu mở rộng dấu:

* lh: sign-ext
* lhu: zero-ext

**Ví dụ:**

1. lhu $t1, 8($t0)

giả sử $t0 = 0x10010000

và từ nhớ tại địa chỉ 0x10010008 có giá trị 0x12345678

Từ nhớ này chứa 4 bytes:

|  |  |  |  |
| --- | --- | --- | --- |
| Byte 4 | Byte 2 | Byte 1 | Byte 0 |
| 12 | 34 | 56 | 78 |

* Lệnh lhu thực hiện việc load 2 byte vào thanh ghi $t1, và byte được load là byte 0 và byte 1. Nhưng do thanh ghi $t1 là 32 bits, nên 16 bits còn lại là 0

Kết quả: $t1 = 0x00005678

1. lh $t1, 8($t0)

giả sử $t0 = 0x10010000

và từ nhớ tại địa chỉ 0x10010008 có giá trị 0x123456f8

Từ nhớ này chứa 4 bytes:

|  |  |  |  |
| --- | --- | --- | --- |
| Byte 4 | Byte 2 | Byte 1 | Byte 0 |
| 12 | 34 | 56 | f8 |

* Lệnh lh thực hiện việc load 2 byte vào thanh ghi $t1, và byte được load là byte 0 và byte 1. Nhưng do thanh ghi $t1 là 32 bits, nên 16 bits còn lại trong lệnh này **được mở rộng có dấu theo bit lớn nhất của byte được load về**

Kết quả: $t1 = 0x000056f8

1. lh $t1, 8($t0)

giả sử $t0 = 0x10010000

và từ nhớ tại địa chỉ 0x10010008 có giá trị 0x12348cde

Từ nhớ này chứa 4 bytes:

|  |  |  |  |
| --- | --- | --- | --- |
| Byte 4 | Byte 2 | Byte 1 | Byte 0 |
| 12 | 34 | 8c | de |

* Lệnh lh thực hiện việc load 2 byte vào thanh ghi $t1, và byte được load là byte 0 và byte 1. Nhưng do thanh ghi $t1 là 32 bits, nên 16 bits còn lại trong lệnh này **được mở rộng có dấu theo bit lớn nhất của byte được load về**

Kết quả: $t1 = 0xffff8cde

1. **Lệnh sw**

****

* Lệnh thuộc nhóm lệnh I-format, có opcode = 2bhex

Address = Offset(rs)

**Syntax (cú pháp):**



**Ý nghĩa:** M[R[rs] + SignExtImm] = R[rt]

* Lưu giá trị thanh ghi rt vào từ nhớ có địa chỉ được tính bằng giá trị thanh ghi rs cộng với offset (offset được mở rộng có dấu thành số 32 bits trước khi cộng)

**Ví dụ:**

sw $t1, 8($t0)

giả sử $t0 = 0x10010000

$t1 = 0x87654321

và từ nhớ tại địa chỉ 0x10010008 có giá trị 0x12345678

Từ nhớ này chứa 4 bytes:

|  |  |  |  |
| --- | --- | --- | --- |
| Byte 4 | Byte 2 | Byte 1 | Byte 0 |
| 12 | 34 | 56 | 78 |

* Lệnh sw thực hiện việc lưu giá trị của thanh ghi $t1 vào từ nhớ có địa chỉ = $t0 + 8 = 0x10010008

Giá trị của từ nhớ tại địa chỉ 0x10010008 sau khi lệnh trên thực hiện là:

|  |  |  |  |
| --- | --- | --- | --- |
| Byte 4 | Byte 2 | Byte 1 | Byte 0 |
| 87 | 65 | 43 | 21 |

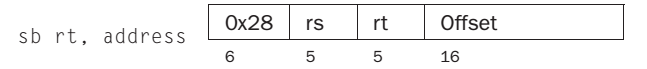
1. **Lệnh sb**

****

* lệnh thuộc nhóm I-format, có opcode = 28hex

Address = Offset(rs)

**Syntax (cú pháp):**



**Ý nghĩa: M[R[rs] + SignExtImm](7:0) = R[rt](7:0)**

* Lấy byte thấp nhất của trong thanh ghi rt lưu vào byte thấp nhất của từ nhớ có địa chỉ được tính bằng giá trị thanh ghi rs cộng với offset (offset được mở rộng có dấu thành số 32 bits trước khi cộng)

**Ví dụ:**

sb $t1, 8($t0)

giả sử $t0 = 0x10010000

$t1 = 0x87654321

và từ nhớ tại địa chỉ 0x10010008 có giá trị 0x12345678

Từ nhớ này chứa 4 bytes:

|  |  |  |  |
| --- | --- | --- | --- |
| Byte 4 | Byte 2 | Byte 1 | Byte 0 |
| 12 | 34 | 56 | 78 |

* Lệnh sb thực hiện việc lưu byte 0 của thanh ghi t1 (0x21) vào byte 0 của từ nhớ tại địa chỉ 0x10010008. Nên sau lệnh trên, hình ảnh từ nhớ:

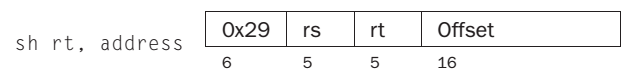
|  |  |  |  |
| --- | --- | --- | --- |
| Byte 4 | Byte 2 | Byte 1 | Byte 0 |
| 12 | 34 | 56 | 21 |

1. **Lệnh sh**

****

* lệnh thuộc nhóm I-format, có opcode = 29hex

**Syntax (cú pháp):**



**Ý nghĩa: M[R[rs] + SignExtImm](15:0) = R[rt](15:0)**

* Lấy 2 byte thấp nhất trong thanh ghi rt (nữa thấp) lưu vào 2 byte thấp nhất của từ nhớ có địa chỉ được tính bằng giá trị thanh ghi rs cộng với offset (offset được mở rộng có dấu thành số 32 bits trước khi cộng)

**Ví dụ:**

sh $t1, 8($t0)

giả sử $t0 = 0x10010000

$t1 = 0x87654321

và từ nhớ tại địa chỉ 0x10010008 có giá trị 0x12345678

Từ nhớ này chứa 4 bytes:

|  |  |  |  |
| --- | --- | --- | --- |
| Byte 4 | Byte 2 | Byte 1 | Byte 0 |
| 12 | 34 | 56 | 78 |

* Lệnh sh thực hiện việc lưu byte 0 và byte 1 của thanh ghi t1 (0x4321) vào byte 0 và byte 1 của từ nhớ tại địa chỉ 0x10010008. Nên sau lệnh trên, hình ảnh từ nhớ:

|  |  |  |  |
| --- | --- | --- | --- |
| Byte 4 | Byte 2 | Byte 1 | Byte 0 |
| 12 | 34 | 43 | 21 |

1. **Lệnh lui**

****

* **Lệnh thuộc I-format, có opcode là fhex**

**Syntax (cú pháp):**



**Ý nghĩa: R[rt] = {imm, 16’b0}**

* Gán số tức thời 16 bits vào nữa cao của thanh ghi rt, nữa thấp đưa 0 vào

**Ví dụ:**

1. lui $t1, 0x1234

Kết quả: $t1 = 0x12340000

1. lui $t1, 0x12345

* báo lỗi do số tức thời tràn quá số 16 bits
* **Ngoài ra còn 2 lệnh thuộc nhóm PseudoInstruction set: *li* và *move***

****

1. **Lệnh li**

**Ý nghĩa:** đưa một số tức thời (32 bits) vào một thanh ghi

**Ví dụ:**

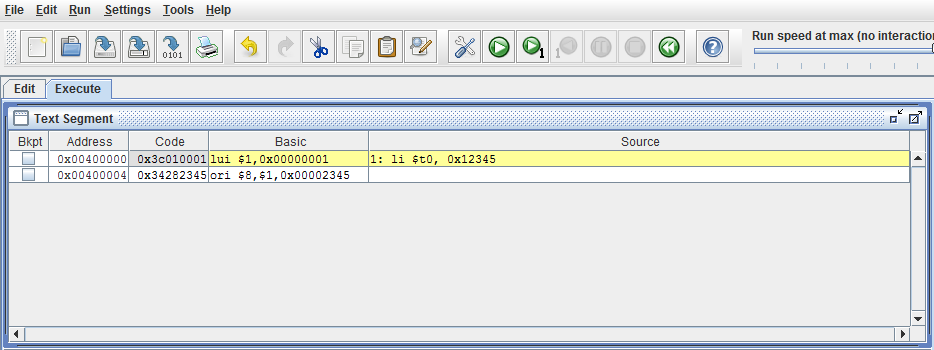
*li $t0, 0x12345*

**Kết quả:** $t0 = 0x00012345

**Lưu ý:**

Số đưa vào lui chỉ được phép là số 16 bits, còn số đưa và li có thể lên đến 32 bits

Lệnh *li* này thực chất là lệnh giả, lệnh này được chuyển thành 2 lệnh (lui và or) khi processor chạy thật sự:



1. **Lệnh *move***

**Ý nghĩa**: sao chép/di chuyển giá trị từ thanh ghi này sang thanh ghi kia

**Ví dụ:**

*move $t1, $t2*

Giả sử $t1 = 0x12345678

$t2 = 0x87654321

Khi lênh trên thực thi, giá trị thanh ghi $t2 được đưa vào thanh ghi $t1

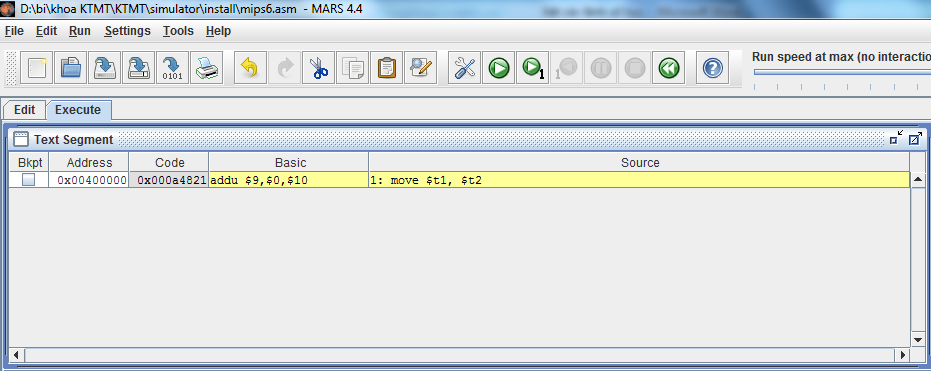
Kết quả sau khi chay:

$t1 = 0x87654321

$t2 = 0x87654321

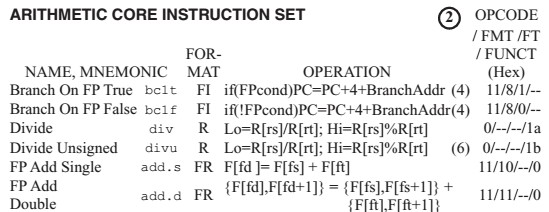
**Chú ý:**

Lệnh *li* này thực chất là lệnh giả, lệnh này được chuyển thành lệnh ‘*addu*’ khi processor chạy thật sự (thực hiện việc cộng thanh ghi $t2 với thanh ghi zero, kết quả nạp vào thanh ghi $t1:



**CÁC LỆNH ASSEMBLY TRONG KHỐI “ARITHMETIC CORE INSTRUCTION SET”**

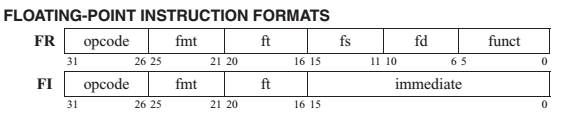
Các lệnh assembly nằm ở bảng con bên phải của bảng số 1 chứa các lệnh số học phức tạp hơn so với các lệnh bên bảng con trái.



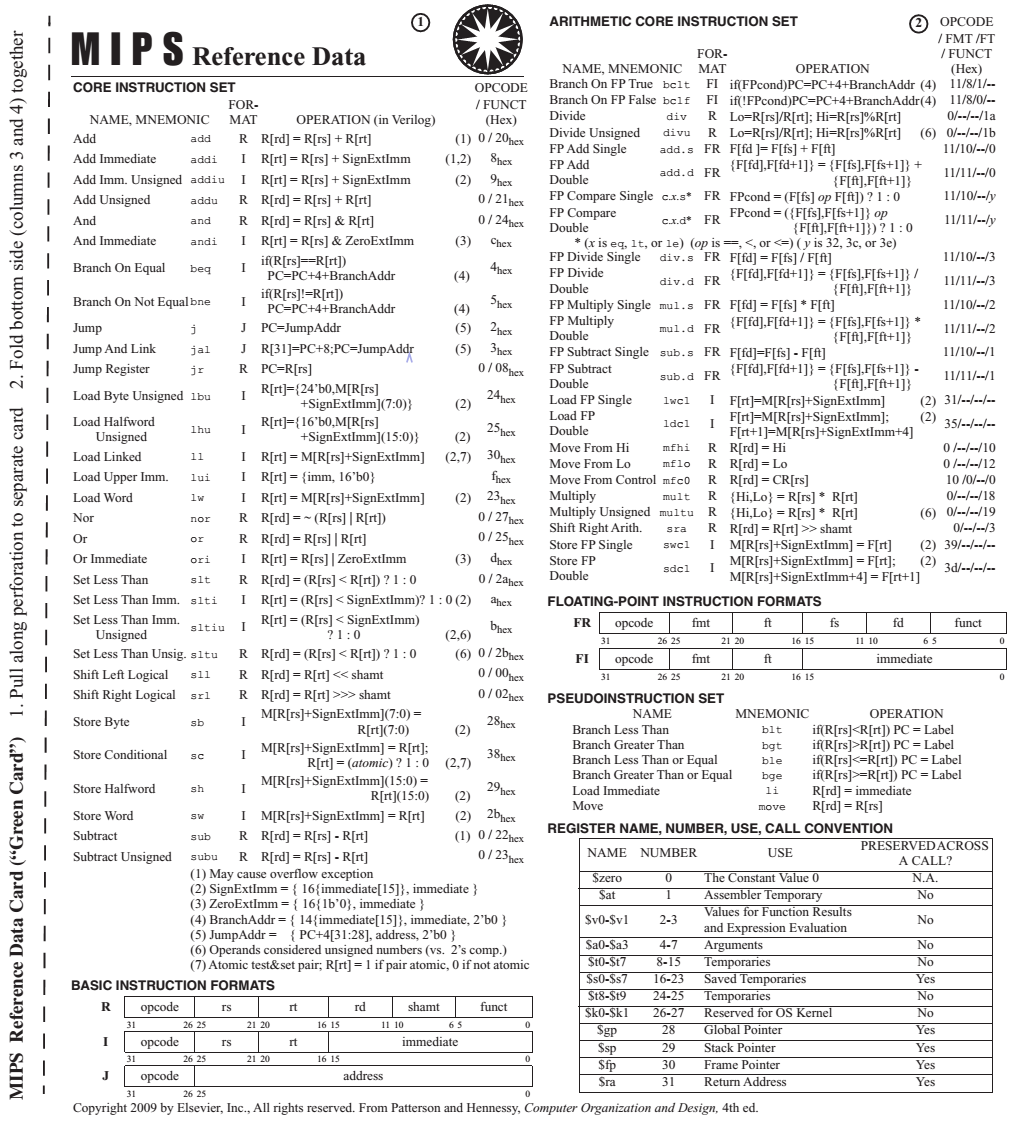
Do các lệnh làm việc với số floating-point có format lệnh hơi khác so với R-format hoặc I-format chuẩn (Có thêm trường ‘fmt’: để phân biệt làm việc với số floating-point độ chính xác đơn hay độ chính kép)

Ví dụ: lệnh bclt có giá trị cột này là “11/8/1/…” tức opcode của lệnh = 11hex, fmt = 8 hex, ft = 1 hex và funct không quan tâm

Do trong bảng này chứa một số lệnh làm việc với số dấu chấm động (floating-point), nên lệnh nào có format có “F” phía trước tức là lệnh làm việc với số dấu chấm động



1. **Các lệnh nhân và chia**

****

Các lệnh assembly cần khi thực hiện phép nhân và chia:

* Nhân: *mult/multu*
* Chia: *div/divu*
* Hai lệnh hỗ trợ: *mfhi/mflo*

*Lệnh mult/multu*



* Hai lệnh này thuộc nhóm lệnh R-format, có opcode là 0. Lệnh *mult* có funct là 18, lệnh *multu* có funct là 19

**Syntax:**

****

****

**Ý nghĩa:**

**{Hi, Lo} = R[rs] \* R[rt]**

* Giá trị trong thanh ghi rs (số 32 bits) nhân với giá trị trong thanh ghi rt (số 32 bits), kết quả là số 64 bits. 32 bits thuộc nữa thấp của kết quả được lưu trong thanh ghi ***Lo***, và 32 bits thuộc nữa cao của kết quả được lưu trong thanh ghi ***Hi***

Chú ý: *Hi* và *Lo* là 2 thanh ghi phụ thêm cho processor khi thực hiện phép toán nhân chia. Vì các thanh ghi đều là 32 bits, mà kết quả phép toán nhân là 64 bits, nên phải dùng 2 thanh ghi tạm này ghép lại

* *mult*: nhân 2 số có dấu
* *multu*: nhân 2 số không dấu

**Ví dụ:**

1. *mult $t1, $t2*

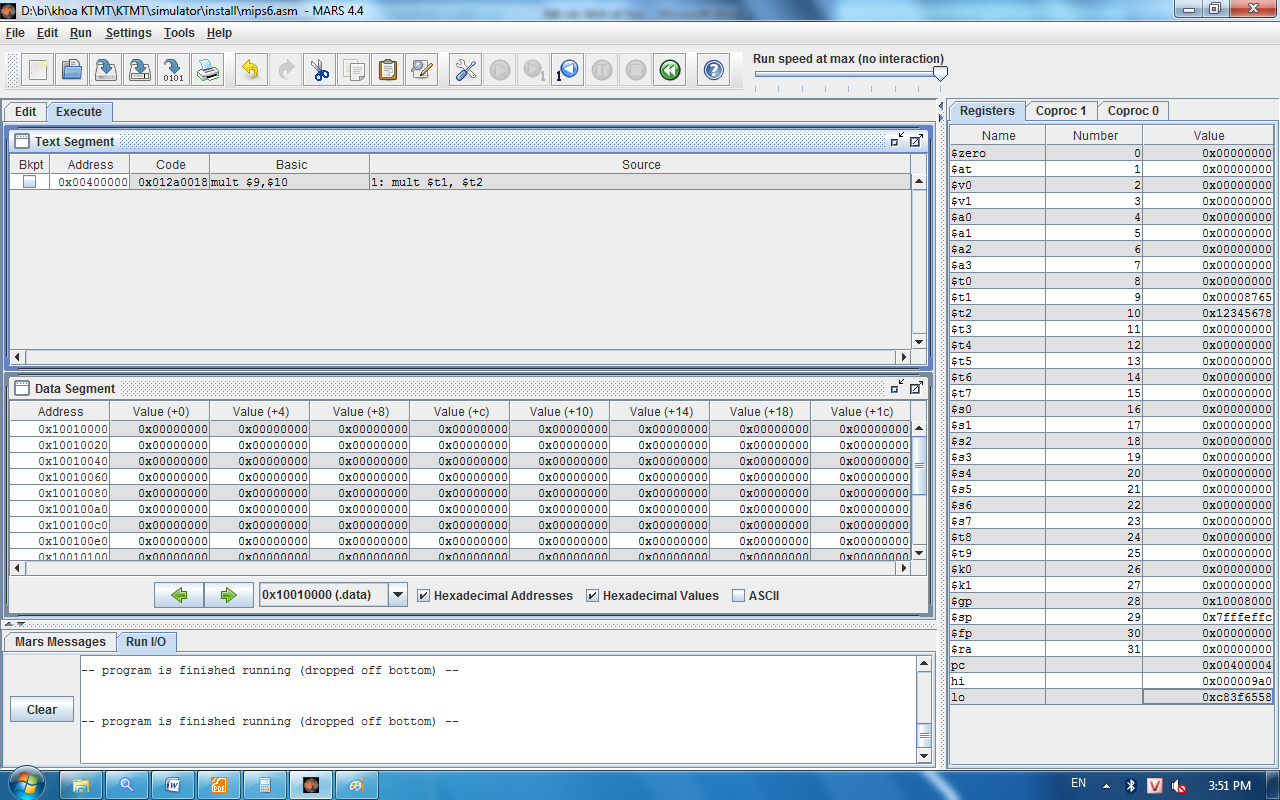
Giả sử $t1 = 0x00008765

$t2 = 0x12345678

Kết quả:

$hi = 0x000009a0

$lo = 0xc83f6558



1. *mult $t1, $t2*

Giả sử $t1 = 0x80008765

$t2 = 0x12345678

Kết quả:

$hi = 0xf6e5de64

$lo = 0xc83f6558

Tức $t1\*$t2 = -0x91A219B37C09AA8 (xét trên số có dấu dạng bù 2)

Kiểm tra lại:

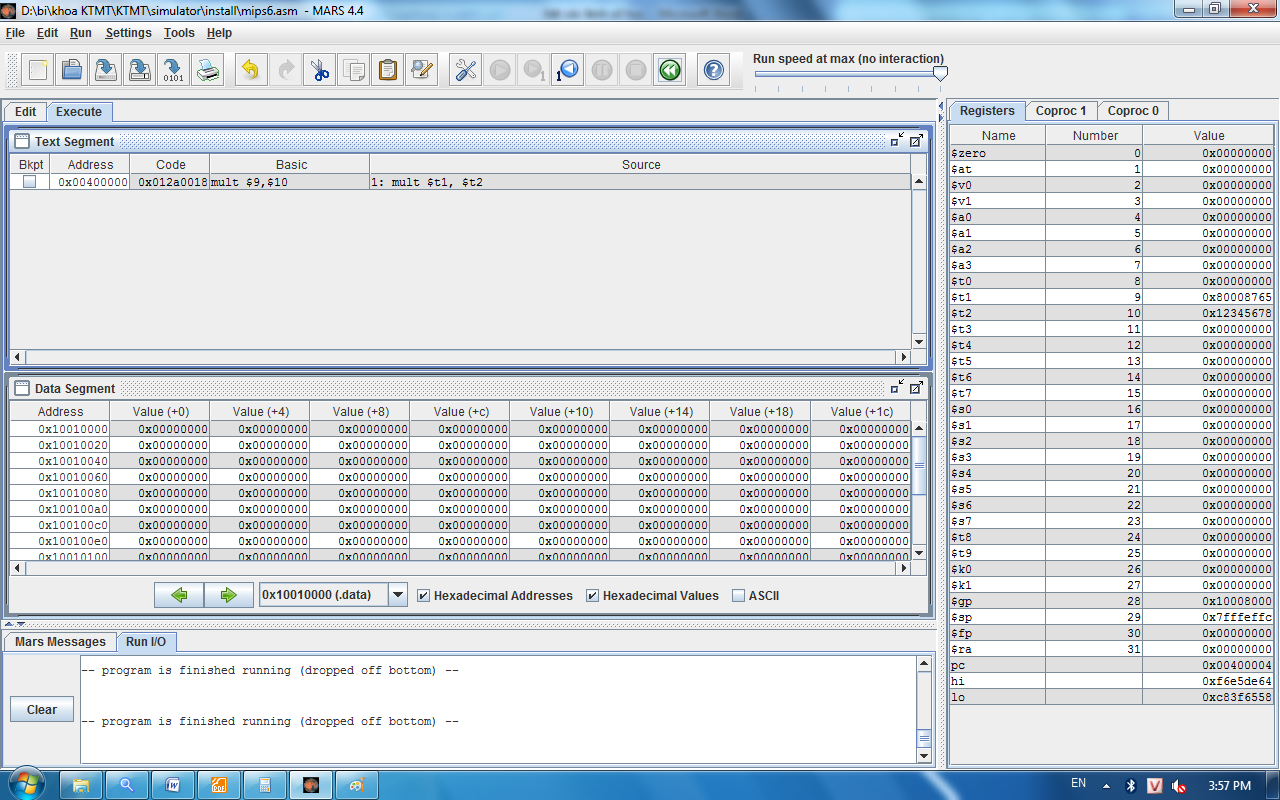
$t1 = 0x80008765 = 1000 0000 0000 0000 1000 0111 0110 0101(2)

= -0x7FFF789B

$t2 = 0x12345678

* $t1 \* $t2 = -(0x7FFF789B x 0x12345678) = -0x91A219B37C09AA8

= 0xF6E5DE64C83F6558 (bù 2)



1. *multu $t1, $t2*

Giả sử $t1 = 0x80008765

$t2 = 0x12345678

Kết quả:

$hi = 0x091a34dc

$lo = 0xc83f6558

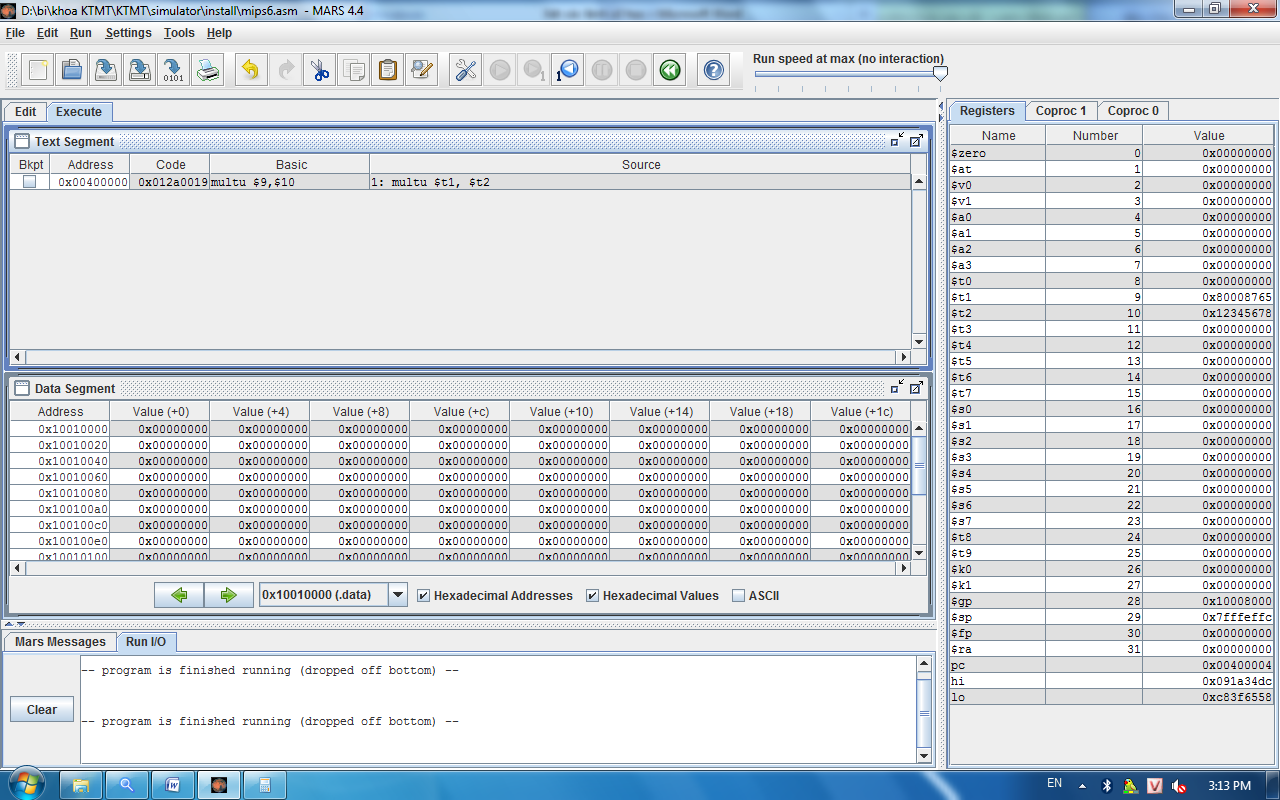
Kiểm chứng kết quả:

multu thực hiện phép nhân 2 số dạng không dấu

$t1 = 0x80008765 = 20000103545(10)

$t2 = 0x12345678 = 2215053170(10)

$t1 x $t2 = 20000103545(10) x 2215053170(10) = 44321515631017662530(10) = 0x91A34DCC83F6558

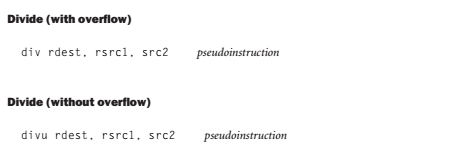


**Chia: *div/divu***



* Hai lệnh này thuộc nhóm lệnh R-format, có opcode là 0. Lệnh *div* có funct là 1ahex, lệnh *divu* có funct là 1bhex

**Syntax:**

****

**Ý nghĩa:**

rdest, rsrcl và src2 là 3 thanh ghi.

Hai lệnh trên lấy giá trị trong thanh ghi rsrcl chia cho src2, thương số đặt vào thanh ghi rdest. Đồng thời thanh ghi Lo cũng chứa thương số giống than ghi rdest và thanh ghi Hi chứa phần dư.

* *div*: chia 2 số có xét tràn
* *divu*: chia 2 số không xét tràn

**Ví dụ:**

a. div $t1, $t2, $t3

giả sử $t2 = 0x6, $t3 = 0x5

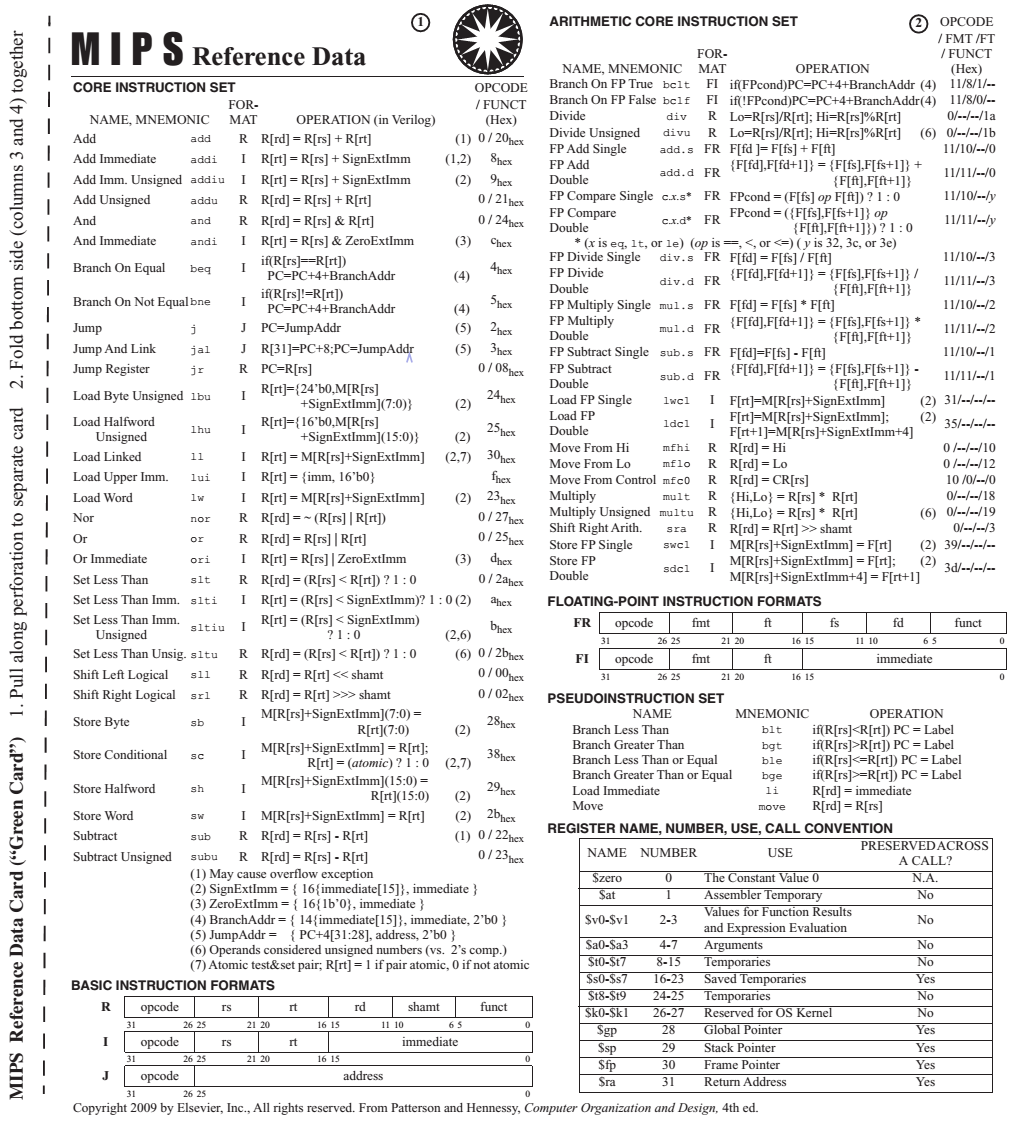
**Kết quả**: $t1 = 0x1

$Lo= 0x1

$Hi = 0x1

* Đưa thêm ví dụ kiểm chứng trường hơp tràn và không tràn (div/divu)

**CÁC LỆNH ASSEMBLY CHO SỐ DẤU CHẤM ĐỘNG (floating-point number)**

****

**Nhóm lệnh số học:**

**Cộng**

*add.s*

*add.d*

**Trừ**

*sub.s*

*sub.d*

**Nhân**

*mul.s*

*mul.d*

**Chia**

*div.s*

*div.d*

Chú ý: Các lênh làm việc với số floating-point làm việc trên 32 thanh ghi f

**----------------------------------------**

****

****

****

****

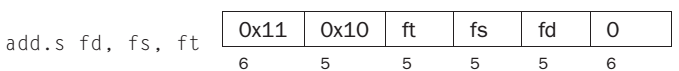
* Các lệnh này thuộc nhóm lệnh FR, có opcode đều bằng 11hex

Lệnh add có funct bằng 0, lệnh sub có funct bằng 1 hex, lệnh mul có funct bằng 2 hex, lệnh div có funct bằng 3 hex

Chú ý: các lệnh .s là làm việc với độ chính xác đơn, có trường fmt luôn bằng 10hex; các lệnh .d là làm việc với độ chính xác kep, có trường fmt luôn bằng 11hex

**add.s**

**Syntax**

****

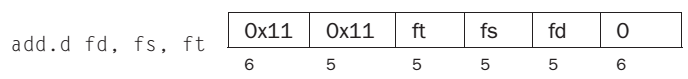
**Ý nghĩa:**

**F[fd] = F[fs] + F[ft]**

* Giá trị trong thanh ghi fs cộng với giá trị trong thanh ghi ft, tổng lưu vào thanh ghi fd

**add.d**

**Syntax**

****

**Ý nghĩa:**

**{F[fd], F[fd+1]} = {F[fs], F[fs+1]} + {F[ft], F[ft+1]}**

* Số tức thời lúc này do biểu diễn trong độ chính xác kép nên cần 64 bits, tức phải cần 2 thanh ghi f liên tục nhau.

Lệnh add.d sẽ thực hiện việc cộng giá trị của số floating point độ chính xác kép đang lưu trong

* Giá trị trong thanh ghi fs cộng với giá trị trong thanh ghi ft, tổng lưu vào thanh ghi fd