ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC BÁCH KHOA
KHOA KHOA HỌC - KỸ THUẬT MÁY TÍNH

Computer Architecture (Extended)

Multiplying Two 64-bit Integers in MIPS Assembly

GVHD: Phạm Quốc Cường
SV: Ngô Đức Anh - 2210077

TP. HỒ CHÍ MINH, THÁNG 5/2025

Mục lục

1 Giới thiệu tổng quan

Báo cáo này được thực hiện trong khuôn khổ học phần Kiến trúc Máy tính, do Thầy Phạm Quốc Cường giảng dạy tại Trường Đại học Bách Khoa – Đại học Quốc gia TP. Hồ Chí Minh. Môn học nhằm cung cấp cho sinh viên nền tảng vững chắc về cấu trúc và nguyên lý hoạt động của hệ thống máy tính, đặc biệt nhấn mạnh đến việc lập trình ở cấp độ thấp và hiểu rõ mối quan hệ giữa phần mềm và phần cứng.

Mục tiêu của bài tập là xây dựng một chương trình bằng hợp ngữ MIPS để thực hiện phép nhân hai số nguyên 64-bit có dấu, đồng thời đảm bảo xử lý đúng các trường hợp đặc biệt như:

- Nhân với các số đặc biệt: 0, 1, -1
- Tràn số (overflow) khi kết quả vượt quá khả năng biểu diễn
- Biểu diễn đúng dấu của kết quả

Chương trình được thực thi trên trình mô phỏng MARS (MIPS Assembler and Runtime Simulator), hỗ trợ việc biên dịch và mô phỏng hành vi của bộ xử lý MIPS. Đây là công cụ phổ biến trong giảng dạy kiến trúc máy tính vì cho phép sinh viên quan sát quá trình xử lý ở mức thanh ghi và bộ nhớ một cách trực quan.

Vai trò của chương trình không chỉ là minh họa kỹ thuật nhân số nguyên có dấu trên kiến trúc 32-bit mà còn giúp sinh viên hiểu sâu hơn về cách các ngôn ngữ lập trình bậc cao chuyển hóa các phép toán số học phức tạp thành tập lệnh cấp thấp điều khiển phần cứng. Từ đó, sinh viên nhận thức được vai trò của từng lệnh Assembly trong chuỗi xử lý dữ liệu – một kiến thức thiết yếu trong việc tối ưu hóa chương trình và thiết kế hệ thống nhúng.

Phạm vi và giới hạn của chương trình bao gồm:

- Xử lý phép nhân hai số nguyên 64-bit có dấu, cho ra kết quả 128-bit
- Xử lý các trường hợp đặc biệt như: nhân với 0, 1, -1, tràn số (overflow)
- In kết quả dưới dạng hexadecimal

Phần tiếp theo của báo cáo sẽ trình bày chi tiết về giải thuật xử lý, phân tích mã lệnh MIPS, hướng dẫn sử dụng chương trình, và các minh họa kết quả đầu ra thông qua các test case tiêu biểu.

2 Phân tích và Thiết kế giải thuật

2.1 Tiền xử lý đầu vào

Mô tả:

Đoạn mã này là một phần của quá trình tiền xử lý đầu vào trong phép nhân hai số 64-bit có dấu. Mục tiêu chính là kiểm tra dấu của hai số (dương hay âm), và nếu số âm, chuyển đổi chúng sang dạng bù 2 để phép nhân có thể được thực hiện như với các số không dấu. Dấu của kết quả cuối cùng sẽ được điều chỉnh dựa trên dấu của hai số ban đầu.

2.1.1 Kiểm tra dấu của hai số

```
# Kiem tra dau cua hai so
lw $t0, dulieu1_high
lw $t1, dulieu2_high
```

Một số 64-bit được biểu diễn bằng hai từ 32-bit: phần thấp (low word) và phần cao (high word). Trong hệ số có dấu, bit cao nhất của phần cao (bit 31 của high word) là bit dấu: 0 nếu số dương, 1 nếu số âm. **Mục đích:** Kiểm tra dấu của hai số bằng cách xem xét bit dấu trong phần cao.

2.1.2 Khởi tạo các biến đánh dấu dấu

```
# Kiem tra so am va xu ly
li $s6, 0      # Bien danh dau so am cho so thu nhat (1 neu am)
li $s7, 0      # Bien danh dau so am cho so thu hai (1 neu am)
```

$s6 là biến đánh dấu dấu của số thứ nhất: 0 nếu dương, 1 nếu âm. $s7 là biến đánh dấu dấu của số thứ hai: 0 nếu dương, 1 nếu âm. Các thanh ghi này được sử dụng để lưu trạng thái dấu của hai số ban đầu, giúp xác định dấu của kết quả phép nhân sau này mà không cần kiểm tra lại. **Mục đích:** Theo dõi dấu để điều chỉnh kết quả cuối cùng (ví dụ: nếu một số âm và một số dương, kết quả sẽ âm).

2.1.3 Kiểm tra dấu của số thứ nhất

```
# Kiem tra so thu nhat co am khong
srl $t2, $t0, 31
beqz $t2, check_second_number
```

srl $t2,$t0, 31: Dịch phải logic (shift right logical) giá trị trong $t0 (phần cao của số thứ nhất) 31 bit, kết quả lưu vào $t2. Điều này đưa bit dấu (bit 31) xuống vị trí thấp nhất (bit 0) của $t2. beqz $t2, check_second_number: Nếu $t2 = 0 (tức là bit dấu là 0, số dương), nhảy đến nhãn check_second_number để kiểm tra số thứ hai. Nếu $t2 = 1 (bit dấu là 1), số thứ nhất là âm, và chương trình sẽ tiếp tục thực hiện các lệnh tiếp theo để xử lý số âm. **Mục đích:** Xác định dấu của số thứ nhất để quyết định xem có cần chuyển đổi sang bù 2 hay không.

2.1.4 Chuyển đổi số thứ nhất sang bù 2 nếu âm

```
# So thu nhat am, lay bu 2
li $s6, 1
lw $t3, dulieu1_low
lw $t4, dulieu1_high
not $t3, $t3    # Lay bu 1
not $t4, $t4
addiu $t3, $t3, 1 # Cong 1 de lay bu 2
sltiu $t5, $t3, 1 #  Kim  tra carry
beqz $t5, store_neg_1
addiu $t4, $t4, 1 #  Cng  carry  vo  high word
store_neg_1:
sw $t3, dulieu1_low
sw $t4, dulieu1_high
```

Quá trình lấy bù 2 bao gồm hai bước: đảo bit (bù 1) và cộng 1. Vì số 64-bit được chia thành hai phần, cần xử lý carry giữa phần thấp và phần cao. Giá trị sau khi lấy bù 2 là giá trị dương tương ứng của số âm ban đầu, cho phép thực hiện phép nhân như với số không dấu. **Mục đích:** Chuyển số âm thành dạng bù 2 để xử lý tiếp theo.

2.1.5 Kiểm tra dấu của số thứ hai

```
check_second_number:
# Kiem tra so thu hai co am khong
srl $t2, $t1, 31
beqz $t2, start_multiplication
```

Nếu $t2 = 1$ (số âm), chương trình tiếp tục thực hiện các lệnh tiếp theo để chuyển đổi số thứ hai sang bù 2. **Mục đích:** Xác định dấu của số thứ hai, tương tự như với số thứ nhất.

2.1.6 Chuyển đổi số thứ hai sang bù 2 nếu âm

```
# So thu hai am, lay bu 2
li $s7, 1
lw $t3, dulieu2_low
lw $t4, dulieu2_high
nor $t3, $t3, $zero # Ly  b  1  ti  u  hn  dng  not
nor $t4, $t4, $zero
addiu $t3, $t3, 1  # Cong 1 de lay bu 2
sltiu $t5, $t3, 1   #  Kim  tra carry
addu $t4, $t4, $t5 #  Cng  carry  vo  high word
store_neg_2:
sw $t3, dulieu2_low
sw $t4, dulieu2_high
```

Lệnh nor với $zero là một cách tối ưu để lấy bù 1, vì nó tương đương với not nhưng có thể

nhanh hơn trên một số kiến trúc MIPS. Quá trình xử lý carry giữa phần thấp và phần cao đảm bảo tính chính xác của dạng bù 2 cho số 64-bit. **Mục đích:** Chuyển số thứ hai âm thành dạng bù 2 để sẵn sàng cho phép nhân.

2.2 Phép nhân 64-bit

Hàm Mult64 thực hiện phép nhân hai số 64-bit có dấu, tạo ra kết quả 128-bit được lưu vào bốn từ 32-bit (result_0 đến result_3). Để đảm bảo tính chính xác, đoạn mã xử lý các trường hợp đặc biệt (như nhân với 0 hoặc trường hợp -2^{63} * -1) trước khi thực hiện phép nhân thông thường. Phép nhân được chia thành các phép nhân 32-bit không dấu, với việc cộng dồn kết quả và xử lý carry để tạo ra kết quả 128-bit chính xác.

2.2.1 Kiểm tra trường hợp một trong hai số bằng 0

```
# Kiem tra neu 1 trong 2 la so 0
lw $t0, dulieu1_low
lw $t1, dulieu1_high
or $t2, $t0, $t1
beqz $t2, mult_complete

lw $t0, dulieu2_low
lw $t1, dulieu2_high
or $t2, $t0, $t1
beqz $t2, mult_complete
```

Phép OR kiểm tra xem toàn bộ số 64-bit có bằng 0 hay không bằng cách xác minh cả hai phần 32-bit đều là 0. Nếu một trong hai số là 0, hàm lập tức kết thúc với kết quả bằng 0, được lưu trong result_0 đến result_3 (đã được khởi tạo bằng 0 trước đó). **Mục đích:** Tối ưu hóa bằng cách bỏ qua phép nhân nếu một trong hai số là 0, tránh các phép tính không cần thiết.

2.2.2 Kiểm tra trường hợp đặc biệt: $-2^{63} * -1$

```
# Kiem tra truong hop dac biet -2^63 * -1
lw $t0, dulieu1_low
lw $t1, dulieu1_high
li $t2, 0
li $t3, 0x80000000
bne $t0, $t2, check_second_min_int
bne $t1, $t3, check_second_min_int

lw $t0, dulieu2_low
lw $t1, dulieu2_high
li $t2, 0xFFFFFFFF
bne $t0, $t2, check_second_min_int
bne $t1, $t2, check_second_min_int

# Xu ly -2^63 * -1
li $t0, 0
```

```
li $t1, 0x80000000
sw $t0, result_0
sw $t1, result_1
sw $zero, result_2
sw $zero, result_3
j mult_complete
```

Trường hợp-2^{63} * -1 là đặc biệt vì -2^{63} (0x80000000_00000000) là số nhỏ nhất có thể biểu diễn trong 64-bit có dấu, và kết quả của phép nhân này (2^{63}) có thể gây nhầm lẫn về overflow nếu không xử lý riêng. Kết quả được gán trực tiếp để tránh thực hiện phép nhân và kiểm tra overflow không cần thiết. **Mục đích:** Xử lý trường hợp đặc biệt để đảm bảo kết quả đúng và tránh báo sai overflow.

2.2.3 Thực hiện phép nhân thông thường

```
# Thuc hien phep nhan thong thuong
continue_mult_setup:
lw $s0, dulieu1_low  # A_low
lw $s1, dulieu1_high # A_high
lw $s2, dulieu2_low  # B_low
lw $s3, dulieu2_high # B_high
```

Các thanh ghi $s0 đến $s3 được sử dụng để lưu trữ các phần 32-bit của hai số, chuẩn bị cho các phép nhân 32-bit. **Mục đích:** Chia mỗi số 64-bit thành hai phần 32-bit để thực hiện bốn phép nhân: A_low * B_low, A_high * B_low, A_low * B_high, và A_high * B_high.

2.2.4 Phép nhân A_low * B_low

```
multu $s0, $s2
mflo $t0
mfhi $t1
sw $t0, result_0
sw $t1, result_1
```

Kết quả của A_low * B_low đóng góp trực tiếp vào các phần thấp của kết quả 128-bit (result_0 và result_1). Vì đây là phép nhân không dấu, các số đầu vào đã được chuyển thành dạng dương (bù 2 nếu âm) trong bước tiền xử lý. **Mục đích:** Tính toán thành phần đầu tiên của phép nhân 64-bit và lưu vào vị trí thấp nhất của kết quả.

2.2.5 Phép nhân A_high * B_low

```
multu $s1, $s2
mflo $t2
mfhi $t3
lw $t4, result_1
addu $t4, $t4, $t2
sw $t4, result_1
```

```
sltu $t5, $t4, $t2
addu $t3, $t3, $t5
sw $t3, result_2
```

Kết quả của A_high * B_low đóng góp vào result_1 và result_2 (dịch 32 bit so với A_low * B_low). Carry được kiểm tra và truyền lên result_2 để đảm bảo tính chính xác của phép cộng 128-bit. **Mục đích:** Cộng dồn thành phần thứ hai của phép nhân vào kết quả 128-bit.

2.2.6 Phép nhân A_low * B_high

```
multu $s0, $s3
mflo $t2
mfhi $t3
lw $t4, result_1
addu $t4, $t4, $t2
sw $t4, result_1
sltu $t5, $t4, $t2
lw $t6, result_2
addu $t6, $t6, $t3
addu $t6, $t6, $t5
sw $t6, result_2
sltu $t5, $t6, $t3
sw $t5, result_3
```

Kết quả của A_low * B_high cũng đóng góp vào result_1 và result_2, tương tự A_high * B_low. Carry được truyền từ result_1 lên result_2 và từ result_2 lên result_3 để xử lý phép cộng 128-bit. **Mục đích:** Cộng dồn thành phần thứ ba vào kết quả.

2.2.7 Phép nhân A_high * B_high

```
multu $s1, $s3
mflo $t2
mfhi $t3
lw $t4, result_2
addu $t4, $t4, $t2
sw $t4, result_2
sltu $t5, $t4, $t2
lw $t6, result_3
addu $t6, $t6, $t3
addu $t6, $t6, $t5
sw $t6, result_3
```

Kết quả của A_high * B_high đóng góp vào result_2 và result_3 (dịch 64 bit so với A_low * B_low). Carry được truyền lên result_3 để hoàn tất phép nhân 128-bit. **Mục đích:** Hoàn thành việc cộng dồn thành phần cuối cùng của phép nhân.

2.3 Kiểm tra overflow

Đoạn mã trong nhãn check_overflow kiểm tra xem kết quả phép nhân 128-bit có thể được biểu diễn chính xác trong phạm vi 64-bit có dấu (từ -2^{63} đến 2^{63} - 1) hay không. Nếu kết quả vượt quá phạm vi này, chương trình báo overflow và đặt kết quả về 0. Đặc biệt, đoạn mã xử lý trường hợp đặc biệt -2^{63} (0x80000000_00000000), một giá trị hợp lệ dù có bit cao.

2.3.1 Kiểm tra trường hợp đặc biệt: Kết quả là -2^{63}

```
# Special case for -2^63
lw $t1, result_0
lw $t2, result_1
lw $t3, result_2
lw $t4, result_3

li $t5, 0
bne $t1, $t5, not_min_int
li $t5, 0x80000000
bne $t2, $t5, not_min_int
li $t5, 0
bne $t3, $t5, not_min_int
bne $t4, $t5, not_min_int

j no_overflow
```

Giá trị -2^{63} trong dạng 64-bit có dấu được biểu diễn là 0x80000000_00000000, với result_0 = 0x00000000, result_1 = 0x80000000, và các phần cao hơn (result_2, result_3) bằng 0. Đây là một trường hợp đặc biệt vì -2^{63} là giá trị nhỏ nhất trong phạm vi 64-bit có dấu, và việc nhân -2^{63} với -1 tạo ra kết quả hợp lệ (-2^{63}) nhưng cần được kiểm tra riêng để tránh báo sai overflow. **Mục đích:** Xác nhận trường hợp đặc biệt -2^{63} là hợp lệ, tránh xử lý sai trong bước kiểm tra overflow tiếp theo.

2.3.2 Kiểm tra dấu của kết quả

```
not_min_int:
lw $t0, result_1
srl $t1, $t0, 31 # Bit  du   ca  result_1
beqz $t1, check_positive_overflow
```

Bit dấu của result_1 là bit 63 của kết quả 64-bit thấp (phần quan trọng của phép nhân 64-bit có dấu). Nếu bit dấu là 0, kết quả được coi là dương; nếu là 1, kết quả là âm. Điều này quyết định cách kiểm tra overflow trong các bước tiếp theo. **Mục đích:** Xác định dấu của kết quả để áp dụng quy tắc kiểm tra overflow tương ứng (dương hoặc âm).

2.3.3 Kiểm tra overflow cho kết quả âm

```
#  Kt   qu   m
li $t4, 0xFFFFFFFF
```

```
lw $t2, result_2
lw $t3, result_3
bne $t2, $t4, overflow_detected
bne $t3, $t4, overflow_detected
j no_overflow
```

Trong biểu diễn số có dấu, nếu kết quả âm, các bit từ 64 đến 127 (tương ứng result_2 và result_3) phải là 1 (0xFFFFFFFF) để đảm bảo mở rộng dấu đúng. Nếu result_2 hoặc result_3 không phải 0xFFFFFFFF, kết quả vượt quá phạm vi 64-bit có dấu, gây overflow. **Mục đích:** Đảm bảo kết quả âm có mở rộng dấu chính xác, phát hiện overflow nếu không thỏa mãn.

2.3.4 Kiểm tra overflow cho kết quả dương

```
check_positive_overflow:
lw $t2, result_2
lw $t3, result_3
bnez $t2, overflow_detected
bnez $t3, overflow_detected
```

Nếu kết quả dương, các bit từ 64 đến 127 (result_2 và result_3) phải là 0 để đảm bảo mở rộng dấu đúng. Nếu result_2 hoặc result_3 khác 0, kết quả vượt quá phạm vi 64-bit có dấu (2^{63} - 1), gây overflow. **Mục đích:** Đảm bảo kết quả dương không vượt quá phạm vi hợp lệ.

2.3.5 Không phát hiện overflow

```
no_overflow:
j print_result
```

Nếu tất cả kiểm tra đều thỏa mãn (kết quả là -2^{63} hoặc mở rộng dấu đúng), kết quả được coi là hợp lệ và sẵn sàng để in. **Mục đích:** Chuyển sang bước in kết quả khi không có lỗi.

2.3.6 Xử lý khi phát hiện overflow

```
overflow_detected:
li $v0, 4
la $a0, str_overflow
syscall
sw $zero, result_0
sw $zero, result_1
sw $zero, result_2
sw $zero, result_3
```

Khi phát hiện overflow, chương trình báo lỗi và đặt kết quả về 0 để đảm bảo không sử dụng giá trị sai. **Mục đích:** Thông báo lỗi và đặt lại kết quả để xử lý an toàn.

2.4 Điều chỉnh dấu kết quả

Đoạn mã này nằm trong bước điều chỉnh dấu của kết quả phép nhân 64-bit có dấu. Mục tiêu là xác định dấu của kết quả dựa trên dấu của hai số đầu vào (được lưu trong $s6 và $s7) và, nếu kết quả âm, chuyển đổi kết quả 128-bit sang dạng bù 2 để đảm bảo biểu diễn chính xác. Quá trình này bao gồm việc đảo bit và cộng 1, với việc xử lý carry qua các phần của kết quả.

2.4.1 Kiểm tra dấu của kết quả

```
# Kiem tra dau cua ket qua
xor $t0, $s6, $s7
beqz $t0, check_overflow # Neu dau giong nhau, ket qua duong
```

Phép XOR xác định dấu của kết quả dựa trên quy tắc: hai số cùng dấu cho kết quả dương, hai số khác dấu cho kết quả âm. Nếu kết quả dương, không cần chuyển đổi bù 2, và chương trình chuyển trực tiếp sang bước kiểm tra overflow. **Mục đích:** Xác định dấu của kết quả để quyết định xem có cần chuyển đổi sang dạng bù 2 hay không.

2.4.2 Chuyển đổi kết quả sang dạng bù 2 (phần thấp: result_0 và result_1)

```
# Ket qua am, lay bu 2
lw $t1, result_0
lw $t2, result_1
not $t1, $t1    # Lay bu 1
not $t2, $t2
addiu $t1, $t1, 1 # Cong 1 de lay bu 2
sltiu $t5, $t1, 1 #  Kim  tra carry
beqz $t5, skip_carry_1
addiu $t2, $t2, 1
skip_carry_1:
sw $t1, result_0
sw $t2, result_1
```

Quá trình lấy bù 2 gồm hai bước: đảo bit (bù 1) và cộng 1. Việc cộng 1 có thể tạo ra carry, cần được truyền từ result_0 sang result_1. Lệnh sltiu (set less than unsigned immediate) kiểm tra carry bằng cách so sánh $t1 với 1, vì nếu $t1 = 0 sau khi cộng, điều này ngụ ý có tràn. **Mục đích:** Chuyển đổi hai phần thấp của kết quả sang dạng bù 2 nếu kết quả âm, đảm bảo biểu diễn đúng giá trị âm.

2.4.3 Chuyển đổi kết quả sang dạng bù 2 (phần cao: result_2 và result_3)

```
lw $t3, result_2
lw $t4, result_3
not $t3, $t3
not $t4, $t4
sltiu $t5, $t2, 1 #  Kim  tra carry  t   phn   thp
beqz $t5, skip_carry_2
addiu $t3, $t3, 1
```

```
sltiu $t5, $t3, 1
beqz $t5, skip_carry_2
addiu $t4, $t4, 1
skip_carry_2:
sw $t3, result_2
sw $t4, result_3
```

Quá trình lấy bù 2 được áp dụng cho hai phần cao của kết quả, với carry được truyền từ result_1 sang result_2 và từ result_2 sang result_3. Việc kiểm tra và truyền carry đảm bảo rằng toàn bộ kết quả 128-bit được chuyển đổi chính xác sang dạng bù 2. **Mục đích:** Hoàn tất việc chuyển đổi kết quả sang dạng bù 2 cho các phần cao nếu kết quả âm.

2.5 In kết quả

Đoạn mã trong nhãn print_result chịu trách nhiệm in kết quả của phép nhân 64-bit có dấu dưới dạng số hexadecimal 128-bit. Nếu kết quả âm, một ký tự dấu trừ ('-') sẽ được in trước. Kết quả được chia thành bốn phần 32-bit (result_0 đến result_3) và được in lần lượt, với các phần được phân cách bằng ký tự 'x' để dễ đọc.

2.5.1 In chuỗi thông báo "Tich = "

```
print_result:
li $v0, 4
la $a0, str_product
syscall
```

Syscall 4 in chuỗi ASCII kết thúc bằng null từ địa chỉ được cung cấp trong $a0. Chuỗi "Tich = " được in để thông báo rằng giá trị tiếp theo là kết quả của phép nhân. **Mục đích:** Cung cấp tiêu đề cho kết quả, giúp người dùng nhận biết giá trị được in là kết quả của phép nhân.

2.5.2 Kiểm tra và in dấu âm nếu cần

```
# In dau am neu can
xor $t0, $s6, $s7
beqz $t0, print_result_value

li $v0, 11
li $a0, 45  # Ky tu '-'
syscall
```

Phép XOR xác định dấu của kết quả dựa trên quy tắc: hai số cùng dấu cho kết quả dương, hai số khác dấu cho kết quả âm. Syscall 11 in ký tự đơn dựa trên mã ASCII được cung cấp trong $a0. Mục đích: In dấu trừ nếu kết quả âm, đảm bảo hiển thị đúng dạng số có dấu.

2.5.3 In giá trị kết quả dạng hexadecimal

```
print_result_value:
```

```
lw $a0, result_3
li $v0, 34 # In  s   dng  hex
syscall

li $v0, 11
li $a0, 'x'
syscall

lw $a0, result_2
li $v0, 34
syscall

li $v0, 11
li $a0, 'x'
syscall

lw $a0, result_1
li $v0, 34
syscall

li $v0, 11
li $a0, 'x'
syscall

lw $a0, result_0
li $v0, 34
syscall
```

Kết quả 128-bit được chia thành bốn phần 32-bit (result_3 đến result_0), được in từ phần cao nhất đến phần thấp nhất. Syscall 34 in số nguyên 32-bit dưới dạng hexadecimal, thường có định dạng "0x" theo sau bởi 8 chữ số hex (ví dụ: 0x1A2B3C4D). Ký tự 'x' được thêm vào giữa các phần để tạo định dạng dễ đọc, ví dụ: 0x00000001x00000002x00000003x00000004. **Mục đích:** Hiển thị toàn bộ kết quả 128-bit dưới dạng hexadecimal, đảm bảo dễ đọc và rõ ràng.

3 Kiểm thử

Trong mục này em sẽ thực hiện với 1 số trường hợp mà em cho rằng là giá trị biến biên của chương trình:

1. Nhân với 0.
2. Nhân với 1.
3. Nhân các số dương nhỏ.
4. Nhân các số âm nhỏ.
5. Nhân số dương lớn gần giới hạn.
6. Nhân số âm lớn gần giới hạn.
7. Nhân số dương và âm lớn (dấu hỗn hợp).
8. Các trường hợp gây tràn số.
9. Trường hợp đặc biệt -2^{63}.

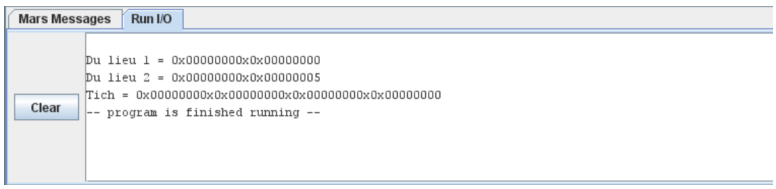
3.1 Nhân với 0

Đầu vào:

```
dulieu1_high: 0x00000000, dulieu1_low: 0x00000000
dulieu2_high: 0x00000000, dulieu2_low: 0x00000005
```

Kết quả mong đợi: 0x0000000000000000

Output chương trình thực tế:

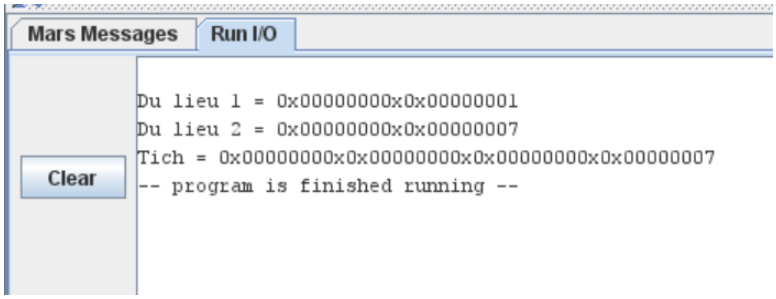


=> Đúng với kết quả mong đợi.

3.2 Nhân với 1

```
dulieu1_high: 0x00000000, dulieu1_low: 0x00000001
dulieu2_high: 0x00000000, dulieu2_low: 0x00000007
```

Kết quả mong đợi: 0x0000000000000007
Output chương trình thực tế:

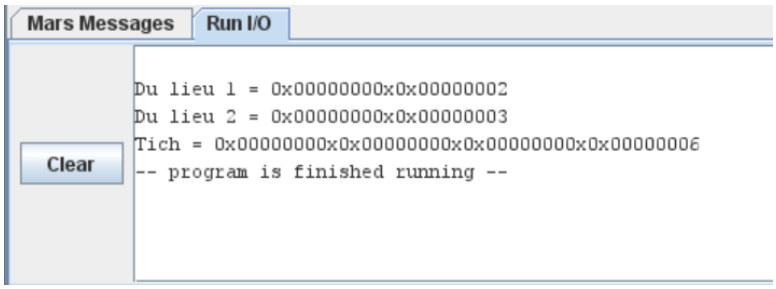


=> Đúng với kết quả mong đợi.

3.3 Nhân các số dương nhỏ

```
dulieu1_high: 0x00000000, dulieu1_low: 0x00000002
dulieu2_high: 0x00000000, dulieu2_low: 0x00000003
```

Kết quả mong đợi: 0x0000000000000006
Output chương trình thực tế:

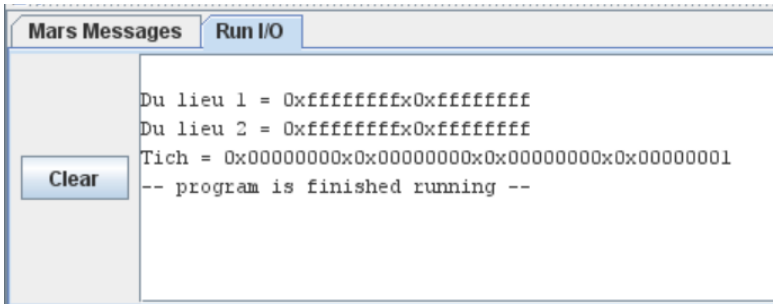


=> Đúng với kết quả mong đợi.

3.4 Nhân các số âm nhỏ

```
dulieu1_high: 0xFFFFFFFF, dulieu1_low: 0xFFFFFFFF
dulieu2_high: 0xFFFFFFFF, dulieu2_low: 0xFFFFFFFF
```

Kết quả mong đợi: 0x0000000000000001
Output chương trình thực tế:

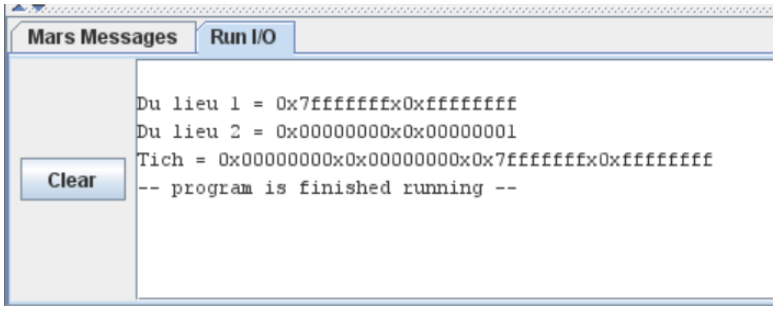


=> Đúng với kết quả mong đợi.

3.5 Nhân số dương lớn

```
dulieu1_high: 0x7FFFFFFF, dulieu1_low: 0xFFFFFFFF
dulieu2_high: 0x00000000, dulieu2_low: 0x00000001
```

Kết quả mong đợi: 0x7FFFFFFFFFFFFFFF
Output chương trình thực tế:

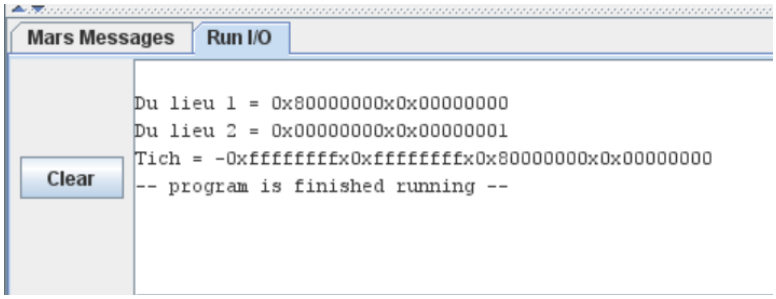


=> Đúng với kết quả mong đợi.

3.6 Nhân số âm lớn

```
dulieu1_high: 0x80000000, dulieu1_low: 0x00000000
dulieu2_high: 0x00000000, dulieu2_low: 0x00000001
```

Kết quả mong đợi: 0x8000000000000000
Output chương trình thực tế:

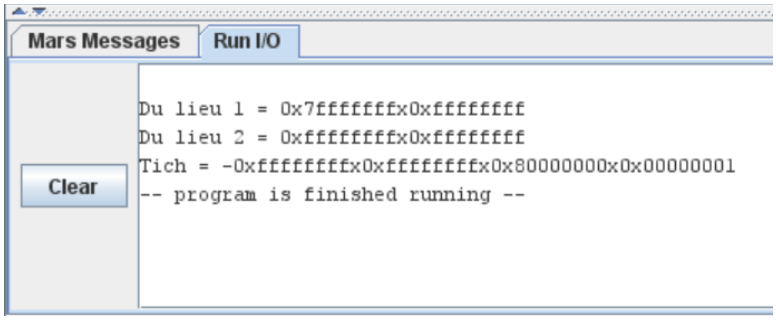


=> Đúng với kết quả mong đợi.

3.7 Nhân số dương và âm lớn (dấu hỗn hợp)

```
dulieu1_high: 0x7FFFFFFF, dulieu1_low: 0xFFFFFFFF
dulieu2_high: 0xFFFFFFFF, dulieu2_low: 0xFFFFFFFF
```

Kết quả mong đợi: 0x8000000000000001
Output chương trình thực tế:

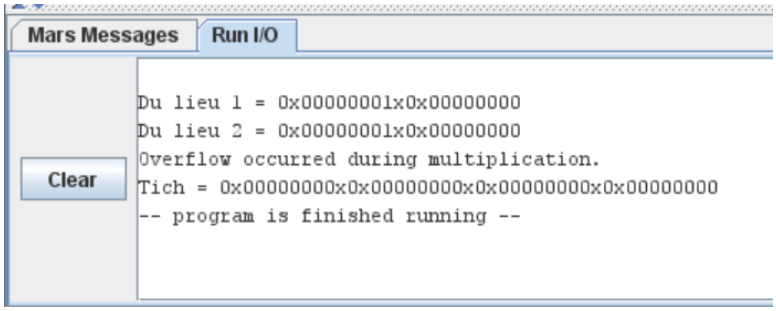


=> Đúng với kết quả mong đợi.

3.8 Gây tràn số với số dương lớn

```
dulieu1_high: 0x00000001, dulieu1_low: 0x00000000
dulieu2_high: 0x00000001, dulieu2_low: 0x00000000
```

Kết quả mong đợi: Báo "Overflow detected", kết quả đặt về 0.
Output chương trình thực tế:

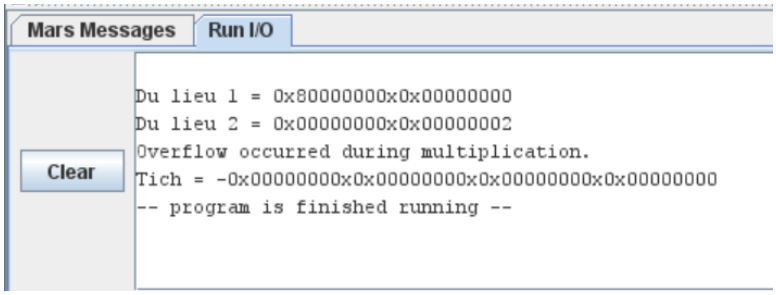


=> Đúng với kết quả mong đợi.

3.9 Gây tràn số với số âm lớn

```
dulieu1_high: 0x80000000, dulieu1_low: 0x00000000
dulieu2_high: 0x00000000, dulieu2_low: 0x00000002
```

Kết quả mong đợi: Báo "Overflow detected", kết quả đặt về 0.
Output chương trình thực tế:

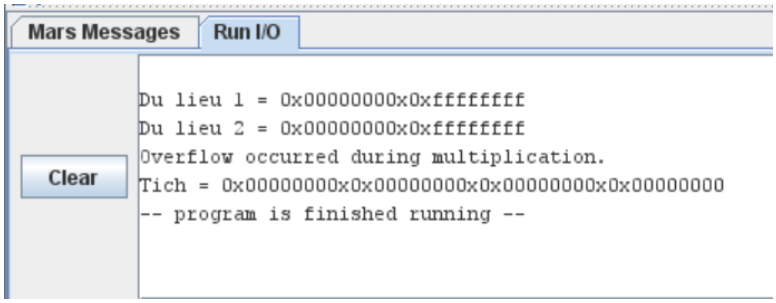


=> Đúng với kết quả mong đợi.

3.10 Nhân hai số gây tràn với carry-over

```
dulieu1_high: 0x00000000, dulieu1_low: 0xFFFFFFFF
dulieu2_high: 0x00000000, dulieu2_low: 0xFFFFFFFF
```

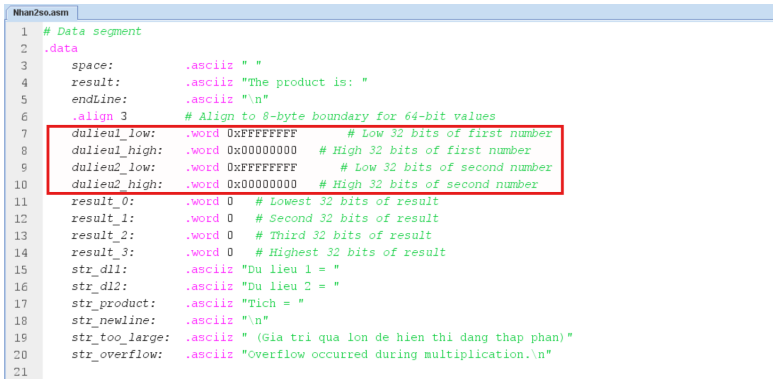
Kết quả mong đợi: Báo "Overflow detected", kết quả đặt về 0.
Output chương trình thực tế:



=> Đúng với kết quả mong đợi.

4 Hướng dẫn sử dụng chương trình

Trong file này, em có đính kèm với source code MIPS với tên file là Nhan2so.asm. Thầy chỉ cần vào phần mềm MARS, và mở file này. Và thay đổi các giá trị dữ liệu kiểm thử cho chương trình bằng mã hexa, với low là 32 bit thấp của số và high là 32 bit cao có bao gồm bit dấu.



Để chuyển thập phân sang hexa thì em có tìm được 1 số trang có cho phép đổi thập phân sang hexa Tại đây.

File Nhan2so.asm.

Tài liệu