Computer Organization: Homework 2 Report

Eric Quezada

November 2, 2024

Student	Eric Quezada
Professor	Dr. Christoph Lauter
Course	CS 3432: Computer Organization
Assignment	Homework 2

Section 1: A Magic Function

```
uint32_t magic(uint32_t x, uint32_t y) {
    uint32_t a = x, b = y, res = 0, t;
    for (int i = 0; i < 32; i++) {
        t = a & 1;
        if (t != 0) res += b;
        a >>= 1;
        b <<= 1;
    }
    return res;
}
```

Section 1.1 Explanation of the Function `magic()`

The `magic()` function performs multiplication between two unsigned 32-bit integers, `x` and `y`, using iterative shifting. For each value of `x`, a shifted left version of `y` is added to the result. This simulates the multiplication of values yielding the product of `x` and `y`. Therefore a more appropriate name for the function "magic" should be "multiply".

Section 1.2 Translating Multiply to "simple" C

```
/* This function does a bitwise multiplication of two variables a and b and
 * returns the result to user */

uint32_t multiply(uint32_t x, uint32_t y) {
  uint32_t a;
  uint32_t b;
  uint32_t res;
  uint32_t t;
  int i;

  a = x;
  b = y;
  res = (uint32_t) 0;
  i = 0;
  start:
    if (i>=32) goto end;
        t = a & ((uint32_t) 1);
```

```
        if (t == 0) goto shift;
            res = res + b;
            goto shift;
  shift:
    a = a >> 1;
    b = b << 1;
    i = i + 1;
    goto start;
  end:
    return res;
}
```

Section 1.3 Multiply Function Test Cases

```
Result of 13 * 9 = 117
Result of 7 * 15 = 105
Result of 0 * 9 = 0
Result of 255 * 255 = 65025
Result of 12 * 12 = 144
```

Section 1.4 Further Translating Multiply to RISC-V Assembly

`Preamble` The preamble of the program sets the environment and establishes some attributes and settings for the assembly code. This includes the following lines:

```
.option nopic           # We do not write PIC code
.attribute arch, "rv32i2p0_m2p0_a2p0_f2p0_d2p0_c2p0" # Requirements for our code
.attribute unaligned_access, 0 # We do not use unaligned accesses
.attribute stack_align, 16   # We align the stack
.text                    # This is where the program starts
.align 1                # Align this function
.globl program          # Label program is a globally callable function
.type program, @function # Label program is a function
```

`Prolog` The prolog of the `program` function sets up the function's stack frame and saves the necessary registers. The prolog consists of the following lines:

```
program:                            # Label for the program function. The function starts here
    addi sp, sp, -32               # Move stack pointer up (to lower addresses)
    sw ra, 28(sp)                  # Store return address on stack
    sw s0, 24(sp)                  # Store frame pointer on stack
    addi s0, sp, 32                # Set frame pointer to stack pointer of caller

# The following lines of code is the Multiply function translated to RISC-V assembly:

multiply_unsigned:
    addi    sp,sp, -32
    sw      ra,28(sp) # Store return address on stack
    sw      s0,24(sp)
    addi    s0,sp,32
    mv      a2, x0 # temporary variable for res we kill the arguments
    mv      a3, x0 # temporary variable for i

multiples_unsigned_loop:
    slti a4, a3, 32 # i < 32
    beq a4, x0, multiples_unsigned_done
    andi a4, a0, 1
    beq a4, x0, multiples_unsigned_after_if
    add a2, a2, a1
```

```
multiples_unsigned_after_if:
   srli   a0, a0, 1
   slli   a1, a1, 1
   addi   a3, a3, 1
   j  multiples_unsigned_loop


multiples_unsigned_done:
   mv  a0, a2

      lw    ra,28(sp)
      lw    s0,24(sp)
      addi    sp,sp,32
      jr    ra
      .size    multiply_unsigned, .-multiply_unsigned
```

Section 2. More Magic: Division

```c
void divide(uint32_t *quot, uint32_t *rem, uint32_t a, uint32_t b) {
    int i;
    uint32_t q, r;
    q = (uint32_t) 0;
    r = (uint32_t) 0;
    for (i=32-1;i>=0;i--) {
        r <<= 1;
        r |= (a >> i) & ((uint32_t) 1);
        if (r >= b) {
            r -= b;
            q |= ((uint32_t) 1) << i;
        }
    }
    *quot = q;
    *rem = r;
}
```

Section 2.1 Explanation of the Function `divide()`

The `divide()` function performs division between two unsigned 32-bit integers: `a`, `b`, and returns the quotient and remainder via pointer arguments `quot` and `rem`, respectively. The implementation simulates long division.

Section 2.2 Translating Divide to "Simple" C

```c
/* This function does a division of two variables a and b and returns the
   result
 * to user in the form of: Quotient of x / y = result, Remainder =
    result_of_remainder
 */
void divide(uint32_t *quot, uint32_t *rem, uint32_t a, uint32_t b) {
  int i;
  uint32_t q, r;

  q = (uint32_t) 0;
  r = (uint32_t) 0;
  i = 32 - 1;

start_loop:
```

```
  if (i < 0) goto end_loop;
  r = r << 1;
  uint32_t shifted_a = a >> i;
  uint32_t isolated_bit = shifted_a & ((uint32_t) 1);
  r = r | isolated_bit;
  if (r >= b) goto subtract;
  goto next_iteration;

subtract:
  r = r - b;
  q = q | (((uint32_t) 1) << i);

next_iteration:
  i = i -1;
  goto start_loop;

end_loop:
  *quot = q;
  *rem = r;
}
```

Section 2.3 Divide Function Test Cases

```
10 / 3 = Quotient: 3, Remainder: 1
15 / 4 = Quotient: 3, Remainder: 3
20 / 5 = Quotient: 4, Remainder: 0
13 / 9 = Quotient: 1, Remainder: 4
0 / 5 = Quotient: 0, Remainder: 0
7 / 2 = Quotient: 3, Remainder: 1
8 / 3 = Quotient: 2, Remainder: 2
```

2.4 Further Translating Divide to RISC-V Assembly

```
# The following lines of code is the Divide function translated to RISC-V assembly:
divide:
    addi    sp, sp, -32         # Allocate stack space
    sw      ra, 28(sp)          # Store return address on stack
    sw      s0, 24(sp)          # Store frame pointer on stack
    addi    s0, sp, 32          # Set frame pointer

    # Initialize q (quotient) and r (remainder) to 0
    li      t0, 0               # t0 = q (quotient)
    li      t1, 0               # t1 = r (remainder)
    li      t2, 0               # t2 = i (bit counter)

divide_loop:
    # Check if i < 32
    li      t3, 32              # Set constant 32
    bge     t2, t3, divide_done  # If i >= 32, end loop

    # r <<= 1
    slli    t1, t1, 1           # Shift remainder (t1) left by 1

    # r |= (a >> (31 - i)) & 1
    li      t4, 31              # Load 31
    sub     t4, t4, t2          # Calculate (31 - i)
```

```
    srl     t5, a0, t4          # Shift a0 (dividend) right by (31 - i) to get current bit
    andi    t5, t5, 1           # Mask with 1 to isolate bit
    or      t1, t1, t5          # OR it with r (t1)

    # if (r >= b)
    blt     t1, a1, skip_subtract  # Skip if r < b

    # r -= b
    sub     t1, t1, a1          # Subtract b from remainder (t1)

    # q |= 1 << (31 - i)
    li      t4, 1               # Load 1
    sll     t4, t4, t4          # Shift left by (31 - i)
    or      t0, t0, t4          # OR with quotient (t0)

skip_subtract:
    addi    t2, t2, 1           # i++
    j       divide_loop         # Repeat loop

divide_done:
    # Store result in output pointers
    sw      t0, 0(a0)           # *quot = q
    sw      t1, 0(a1)           # *rem = r

    # Restore stack and return
    lw      ra, 28(sp)          # Load return address
    lw      s0, 24(sp)          # Restore frame pointer
    addi    sp, sp, 32          # Deallocate stack space
    jr      ra                  # Return to caller
    .size divide, .-divide
```

Section 3. Using the Magic: Converting Integers to Digits

```
void convert(char *str, uint32_t a) {
    uint32_t t, r;
    char *curr;
    char *i, *d;
    char c;
    t = a;
    if (t == ((uint32_t) 0)) {
        curr = str;
        *curr = '0';
        curr++;
        *curr = '\0';
        return;
    }
    curr = str;
    while (t != ((uint32_t) 0)) {
        r = t % ((uint32_t) 10);
        t /= (uint32_t) 10;
        *curr = r + '0';
        curr++;
    }
    *curr = '\0';
```

```
    curr--;
    i = str;
    d = curr;
    while (i < d) {
        c = *i;
        *i = *d;
        *d = c;
        i++;
        d--;
    }
}
```

Section 3.1 Explanation of the Function

The `convert()` function converts an unsigned 32-bit integer `a` into its string representation in the character array `str`. It initializes `t` to `a` and points `curr` to `str`. If `a` is zero, it stores '0' in `str` and returns.

The function loops until `t` is zero, calculating the last digit (`r`) with `t` modulo 10, updating `t`, converting `r` to a character, and storing it in `str`. After adding a null terminator, it reverses the string by swapping characters with pointers `i` and `d`.

Section 3.2 Translating Convert to "Simple" C

```
void convert(char *str, uint32_t a) {
    uint32_t t, r;
    char *curr;
    char *i, *d;
    char c;

    t = a;
    if (t == ((uint32_t) 0)) goto zero_case;

    curr = str;

convert_loop:
    if (t == ((uint32_t) 0)) goto end_convert_loop;
    r = t % ((uint32_t) 10);
    t = t / ((uint32_t) 10);
    *curr = r + '0';
    curr = curr + 1;
    goto convert_loop;

end_convert_loop:
    *curr = '\0'; // Null-terminate the string
    curr = curr - 1;

    i = str;
    d = curr;

reverse_loop:
    if (i >= d) goto end_reverse_loop;
    c = *i;
    *i = *d;
    *d = c;
    i = i + 1;
    d = d - 1;
    goto reverse_loop;

end_reverse_loop:
```

```
    return;

zero_case:
    curr = str;
    *curr = '0';
    curr = curr + 1;
    *curr = '\0';
    return;
}
```

Section 3.3 Convert Function Test Cases

```
The number 0 as a string is: "0"
The number 1 as a string is: "1"
The number 10 as a string is: "10"
The number 123456789 as a string is: "123456789"
The number 4294967295 as a string is: "4294967295"
The number 255 as a string is: "255"
The number 999999999 as a string is: "999999999"
The number 1000000000 as a string is: "1000000000"
The number 15 as a string is: "15"
```

3.4 Further Translating Convert to RISC-V Assembly

```
# The following lines of code is the Convert function translated to RISC-V assembly:

convert:
    # Save registers on the stack
    addi    sp, sp, -48         # Allocate stack space
    sw      ra, 44(sp)          # Store return address
    sw      s0, 40(sp)          # Store frame pointer
    addi    s0, sp, 48          # Set frame pointer

    # Initialize variables
    mv      t0, a1              # t0 = t = a
    mv      t1, a0              # t1 = curr = str

    # Handle case where t == 0
    beqz    t0, zero_case

convert_loop:
    beqz    t0, end_convert_loop  # If t == 0, end loop

    # r = t % 10
    li      t2, 10               # Load 10 into t2
    remu    t3, t0, t2           # t3 = r = t % 10

    # t = t / 10
    divu    t0, t0, t2           # Update t = t / 10

    # *curr = r + '0'
    addi    t3, t3, '0'          # Add ASCII '0' to convert to char
    sb      t3, 0(t1)            # Store the character in *curr

    # curr = curr + 1
```

```
    addi    t1, t1, 1            # Move to the next character in str
    j       convert_loop          # Continue loop

end_convert_loop:
    sb      x0, 0(t1)            # Null-terminate the string
    addi    t1, t1, -1            # Move curr to the last character

    # Set up for reversing
    mv      t4, a0               # t4 = i = str
    mv      t5, t1               # t5 = d = curr

reverse_loop:
    bge     t4, t5, end_reverse_loop  # If i >= d, end reverse loop

    # Swap *i and *d
    lb      t3, 0(t4)            # Load *i into t3
    lb      t6, 0(t5)            # Load *d into t6
    sb      t6, 0(t4)            # *i = *d
    sb      t3, 0(t5)            # *d = *i

    # Move i and d
    addi    t4, t4, 1            # i++
    addi    t5, t5, -1           # d--
    j       reverse_loop          # Continue reversing

end_reverse_loop:
    # Restore registers and return
    lw      ra, 44(sp)           # Load return address
    lw      s0, 40(sp)           # Restore frame pointer
    addi    sp, sp, 48           # Deallocate stack space
    jr      ra                   # Return

zero_case:
    # Handle the case for a == 0
    li      t0, 48               # Load ASCII code for '0' into t0
    sb      t0, 0(a0)            # Store '0' in *str
    addi    a0, a0, 1            # Move to next char
    sb      x0, 0(a0)            # Null-terminate the string

    # Restore registers and return
    lw      ra, 44(sp)           # Load return address
    lw      s0, 40(sp)           # Restore frame pointer
    addi    sp, sp, 48           # Deallocate stack space
    jr      ra                   # Return

    .size convert, .-convert
```