Computer Architecture
(ENE1004)

Assignment – 2: Implementing Function Calls

Specification: Shuffle 32 Bits

- The goal of HW#2 is the same as that of HW#1
  - Again, you are required to implement the **shuffle32** function in HW#2
  - However, in HW#1, there have been no requirements in implementing the function
- However, you must implement **shuffle32** in the following fashion
  - You implement **shuffle32** as a function that calls **shuffle16** twice
  - And, you implement **shuffle16** as a function that calls **shuffle8** twice
  - And, you implement **shuffle8** as a function that calls **shuffle4** twice
  - Thus, you should write the four functions, **shuffle4**, **shuffle8**, **shuffle16**, and **shuffle32**
- For students who fail to make a successful submission for HW#1, you are allowed to try HW#1 again; implement only **shuffle32** in any way you like

Hint (1)

- Note that you write new nested procedures
  - At the beginning of a function, you should save register values in the stack “if necessary”
  - Before returning to the caller (at the end of the function), you should restore register values from the stack for the caller
- First, implement **shuffle4**
  - Input ($a0): a value as original bit positions {3 2 1 0}
  - Output ($v0): the shuffled value as original bit positions {3 1 2 0}
  - Simply switch the two bit values, bit 2 and bit 1
- Next, implement **shuffle8**
  - Input ($a0): a value as original bit positions {7 6 5 4 3 2 1 0}
  - Output ($v0): the shuffled value as original bit positions {7 3 6 2 5 1 4 0}
  - (i) Divide the 8 bits into four 2-bit chunks and switch the two middle chunks
    - {(7 6) (5 4) (3 2) (1 0)} → {(7 6) (3 2) (5 4) (1 0)}
  - (ii) Call **shuffle4** using the first 4-bit chunk – {(7 6 3 2) (5 4 1 0)}
    - shuffle4 (7 6 3 2) (5 4 1 0) → (7 3 6 2) (5 4 1 0)
  - (iii) Call **shuffle4** using the second 4-bit chunk – {(7 6 3 2) (5 4 1 0)}
    - shuffle4 (7 3 6 2) (5 4 1 0) → (7 3 6 2) (5 1 4 0)

Hint (2)

- Then, implement **shuffle16**
  - Input ($a0): a value as original bit positions {15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0}
  - Output ($v0): the shuffled value as original bit positions {15 7 14 6 13 5 12 4 11 3 10 2 9 1 8 0}
  - (i) Divide the 16 bits into four 4-bit chunks and switch the two middle chunks
    - {(15 14 13 12) (11 10 9 8) (7 6 5 4) (3 2 1 0)} → {(15 14 13 12) (7 6 5 4) (11 10 9 8) (3 2 1 0)}
  - (ii) Call **shuffle8** using the first 8-bit chunk - {(15 14 13 12 7 6 5 4) (11 10 9 8 3 2 1 0)}
    - (15 14) (13 12) (7 6) (5 4) → (15 14) (7 6) (13 12) (5 4)
    - shuffle4 (15 14 7 6) (13 12 5 4) → (15 7 14 6) (13 12 5 4)
    - shuffle4 (15 7 14 6) (13 12 5 4) → (15 7 14 6) (13 5 12 4)
  - (iii) Call **shuffle8** using the second 8-bit chunk – {(15 7 14 6 13 5 12 4) (11 10 9 8 3 2 1 0)}
    - (11 10) (9 8) (3 2) (1 0) → (11 10) (3 2) (9 8) (1 0)
    - shuffle4 (11 10 3 2) (9 8 1 0) → (11 3 10 2) (9 8 1 0)
    - shuffle4 (11 3 10 2) (9 8 1 0) → (11 3 10 2) (9 1 8 0)
- Finally, implement **shuffle32**
  - In the similar way, divide and conquer
  - This function calls **shuffle16** twice

Evaluation

- Test cases is also given; exactly the same as those of HW#1

```
 i testcase[i]          testcase[i] in binary         shuffled result in binary      result
 0 0xffffffff 11111111111111111111111111111111 11111111111111111111111111111111 0xffffffff
 1 0xffff0000 11111111111111110000000000000000 10101010101010101010101010101010 0xaaaaaaaa
 2 0x0000ffff 00000000000000001111111111111111 01010101010101010101010101010101 0x55555555
 3 0xff00ff00 11111111000000001111111100000000 11111111111111110000000000000000 0xffff0000
 4 0x00ff00ff 00000000111111110000000011111111 00000000000000001111111111111111 0x0000ffff
 5 0xf0f0f0f0 11110000111100001111000011110000 11111111000000001111111100000000 0xff00ff00
 6 0x0f0f0f0f 00001111000011110000111100001111 00000000111111110000000011111111 0x00ff00ff
 7 0xcccccccc 11001100110011001100110011001100 11110000111100001111000011110000 0xf0f0f0f0
 8 0x33333333 00110011001100110011001100110011 00001111000011110000111100001111 0x0f0f0f0f
 9 0xaaaaaaaa 10101010101010101010101010101010 11001100110011001100110011001100 0xcccccccc
10 0x55555555 01010101010101010101010101010101 00110011001100110011001100110011 0x33333333
11 0x00000000 00000000000000000000000000000000 00000000000000000000000000000000 0x00000000
12 0xffff0000 11111111111111110000000000000000 10101010101010101010101010101010 0xaaaaaaaa
13 0xaaaaaaaa 10101010101010101010101010101010 11001100110011001100110011001100 0xcccccccc
14 0xcccccccc 11001100110011001100110011001100 11110000111100001111000011110000 0xf0f0f0f0
15 0xf0f0f0f0 11110000111100001111000011110000 11111111000000001111111100000000 0xff00ff00
16 0xff00ff00 11111111000000001111111100000000 11111111111111110000000000000000 0xffff0000
17 0x12345678 00010010001101000101011001111000 00010011000111000001111101100000 0x131c1f60
All done!
```

- Submission
  - Upload your file, named “name_id.asm”, to LMS
  - Due by Jun 20 (Tue) at midnight
- Never cheat! If you cheat, you will get an F
  - Do your best; try to submit your best version even if your program does not work well