All Exercises CAr

Student Solutions
Exercises Computer Architecture

1 | Chip & Die Fabrication

1.1 Fabrication

a) 71.8%
b) 235.5 dies
c) 169.1 good_dies
d) 1.18 CHF

fun/fabrication-01

1.2 Fabrication

a) $120\frac{\text{wafers}}{\text{ingot}}$
b) 250CHF
c) 0.796CHF
d) 209.3 dies
e) 158.23 dies
f) 2.05CHF

fun/fabrication-02

1.3 Fabrication

a) 200CHF
b) $\approx 600\frac{\text{dies}}{\text{wafer}}$
c) $1.06\frac{\text{CHF}}{\text{die}}$

fun/fabrication-03

2 | Moore's Law & Denard scaling

2.1 Dennard Scaling

a) $1.414 = \sqrt{2}$

b) 406pm equals to 16601 times smaller

fun/dennardscaling-01

2.2 Dynamic power consumption of a CMOS circuit is:

Two statements are true, one is false.

fun/dennardscaling-02

3 | Power Consumption

3.1 Cell phone battery life

a) $112.6h$

b) $9.19h$

fun/powerconsumption-01

1 | Processor Benchmark & Performance

1.1 Which of the following statements are correct?

Three statements are true one is false.

per/benchmark-01

1.2 What is the throughput?

One statement is true and three are false.

per/benchmark-02

1.3 What is the SPEC?

One statement is true and three are false.

per/benchmark-03

1.4 What is the goal of the EEMBC Benchmark?

One statement is correct and three are false

per/benchmark-04

1.5 Which of the following is an energy efficiency metric?

One statement is correct and three are false.

1.6 Both power consumption and performance per watt matters for an embedded system.

50/50 change. Think.

1.7 Processor performance

a) $30\mu s$
b) $2\frac{\text{cycles}}{\text{instruction}}$
c) $5\frac{\text{cycles}}{\text{instruction}}$
d) $292\mu s$
e) Processor B is 1.29 times faster than processor A.

1.8 Processor performance

a) $\text{CPI}_{\text{Avg\_A}} = 3.775\frac{\text{cycle}}{\text{instr}}$ & $\text{CPI}_{\text{Avg\_A}} = 2.52\frac{\text{cycle}}{\text{instr}}$
b) Computer B is 1.35 times faster than Computer A.
c) 2.69GHz

1.9 Processor performance

Execution_time = 8.75ms

1.10 Processor performance

Variant 2

1.11 Processor performance

a) CPU_A is better when
   a) $w_{p_1} > 90.\overline{90}\%$
   b) $w_{p_2} < 9.\overline{09}\%$
b) CPU_B is better when
   a) $w_{p_1} > 90\%$
   b) $w_{p_2} < 10\%$
c) CPU_C is better when
   a) $w_{p_1} > 50\%$
   b) $w_{p_2} < 50\%$

1.12 Processor performance

Central-Processing-Unit (CPU) A is the fastest!

1.13 Processor performance

The clock frequency of the CPU is 2 GHz

4.65

1.14 What is the best metric for comparinc performance?

One statement is true the others are false.

1.15 Processor performance

$T = 3.2\overline{3}\text{ms}$

1.16 Amdahl's Law

$S = 5.263\%$

1.17 Amdahl's Law

$f = 66.\overline{6}\%$

1.18 Amdahl's Law

Optimization A is 1.28 times better than Optimization B.
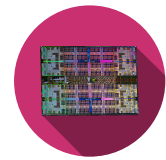
1 | Implementation

1.1 What is the main difference between a hard and a soft real-time system?

One statement is correct the other one is false.

1.2 What is an embedded system?

One statement is correct all others are false.

imp/implementation-02

1.3 Faster execution time means less energy.

One statement is correct the other false.

imp/implementation-03

1.4 Why are more and more SOC being developed insteead of CPU's?

All statements are either correct or false.

imp/implementation-04

1 | Instruction-Set Architecture

1.1 Simple C-Code to RISC-V Assembler

1.1.1 Students guide

a) You need the instruction: **add**
b) You need the instructions: **add**, **sub**
c) You need the instruction: **addi**
d) You need the instruction: **addi**
e) You need the instructions: **lui**, **addi**. Beware immediates overflow.
f) You need the instructions: **lui**, **addi**. Beware immediates overflow.

isa/c-to-riscv-01

1.2 Algorithmic C-Code to RISC-V Assembler

1.2.1 Students guide

a) One variant is with: **bne**, **add**, **sub**
b) One variant is with: **bne**, **add**, **j**, **sub**
c) One variant is with: **addi**, **bne**, **add**, **j**
d) One variant is with: **addi**, **bge**, **add**, **slli**, **j**
e) One variant is with: **lui**, **addi**, **lw**, **slli**, **sw**
f) One variant is with: **lui**, **ori**, **addi**, **bge**, **slli**, **add**, **lw**, **sw**, **j**
g) One variant is with: **addi**, **add**, **lb**, **beq**, **j**

isa/c-to-riscv-02

1.3 Machine code to RISC-V Assembler

1.3.1 Students guide

a)
```
0x41FE 83B3 = 0100 0001 1111 1110 1000 0011 1011 0011
```

op $= 51$, funct3 $= 0 \Rightarrow$ **add** or **sub** (R-Type Command)

funct7 = 01000000 $\Rightarrow$ **sub**

funct7	rs2	rs1	funct3	rd	op
0100 000	11111	11101	000	00111	0110011
32	31	29	0	7	51

```
sub t2, t4, t6
```

b) I-Type

isa/machinecode-to-riscv-01

1.4 Logic operations on registers

1.4.1 Students guide

a) **s3 = 0x46A1 0000**
b) **s4 = 0xFFFF 01B7**
c) **s5 = 0xB95E F1B7**

isa/riscv-execution-01

1.5 Logic operations on values

1.5.1 Students guide

a) **s3 = 0x3A75 0824**
b) -
c) -

isa/riscv-execution-02

1.6 RISC-V multiplication

1.6.1 Students guide

s4 = 0xE000 0000
s3 = 0x0000 0000

isa/riscv-execution-03

1.7 Division and modulo

1.7.1 Students guide

s3 = 0x0000 0005
s4 = 0x0000 0002

isa/riscv-execution-04

1.8 R-Type to Machine code

1.8.1 Students guide

a)
```
add x18, x18, x20
```

R-Type Command

funct7	rs2	rs1	funct3	rd	op
0	20	19	0	18	51
0000000	10100	10011	000	10010	0110011

```
0x0149 8933
```

b) -
c) **0x0092 9BB3**
d) -
e) -

isa/riscv-to-machinecode-01

1.9 I-Type to Machine code

1.9.1 Students guide

a)
```
addi x8, x9, 12
```

I-Type Command

$imm_{11:0}$	rs1	funct3	rd	op
12	9	0	8	19
0000 0000 1100	01001	000	01000	001 0011

```
0x00C4 8413
```

b) -
c) -
d) **0x01B0 1483**
e) -

isa/riscv-to-machinecode-02

1.10 S-Type to Machine code

1.10.1 Students guide

a) **0xFE79 AD23**
b) -
c)

```
sb x30, 0x2D(x0)
```

S-Type Command

imm$_{11:5}$	rs2	rs1	funct3	imm$_{4:0}$	op
0000 001	30	0	0	01101	35
0000 001	11110	00000	000	01101	010 0011

```
0x03E0 06A3
```

isa/riscv-to-machinecode-03

1.11   Real-time system

What is the main difference between a "hard" real-time system and a "soft" real-time system?

1.11.1 Students guide

One of those system types is considered as failed if it misses any timing. When/Why is it necessary to be so strict ?

isa/riscv-to-machinecode-04

1.12   U-Type to Machine code

1.12.1 Students guide

0x8CDE FAB7

isa/riscv-to-machinecode-05

1.13   J-Type to Machine code

1.13.1 Students guide

0x0FF8A 60EF

isa/riscv-to-machinecode-06

2 | Laboratory complement

To help you, feel free to use the RISC-V interpreter on https://course.hevs.io/car/riscv-interpreter/ as well as Ripes.

> ℹ️ Be careful with variable types!
> - **int** type is considered as a signed, 32 bits value.
> - **unsigned int** type is considered as an unsigned, 32 bits value.
> - If followed by a number (e.g.: **int16_t**), it means the value is on x bits (here 16). If preceded by a **u**, it is unsigned.
>
> **uint8_t** is an unsigned byte, whereas **int8_t** is a signed byte.

2.1 Basic calculations

2.1.1 Students guide

a)

```
# a = b + c;
# s0 = a, s1 = b, s2 = c

# b = 1, c = 2
addi s1, zero, 1
addi s2, zero, 2
add s0, s1, s2      # a = b + c
# s0 = 0x00000003
# s1 = 0x00000001
# s2 = 0x00000002

# b = -1, c = 2

...

# s0 = 0x00000001
# s1 = 0xffffffff
# s2 = 0x00000002

# b = -12, c = 2032

...

# s0 = 0x000007db
# s1 = 0xfffffff4
# s2 = 0x000007e7
```

b)

```
# a = b - c;
# d = (e + f) - (g + h);
# s0-s7 = a-h

...

# t0 = 0xfffffffb1
# t1 = 0x000007db
# s0 = 0xffffffff
# s1 = 0x00000002
# s2 = 0x00000003
# s3 = 0xfffff7d6
# s4 = 0xffffffff
# s5 = 0xfffffffb2
# s6 = 0x000007e7
# s7 = 0xfffffff4
```

isa/lab-basic-calc

2.2 Memory access

2.2.1 Students guide

```
# Check for sign extension comprehension

# uint16_t a = mem[3];
# mem[4] = a;
# t0 is a
```

```
lhu t0, 3(zero) # if lh, the last bit may be 1 -> extended -> wrong number
sw  t0, 4(zero)


# int16_t a = mem[3];
# mem[4] = a;
# t0 is a
??? t0, 3(zero) # if lh, the last bit may be 1 -> ???
??? t0, 4(zero)
```

isa/lab-memory

2.3 Basic algorithms

1. Transmit the 8-bit memory value at address 0x0000′1000 serially, bit by bit, into the Least Significant Bit (LSB) of memory address 0x0000′1001. The remaining bits of memory address 0x0000′1001 must be '**0**'. Calculate the baud rate in $\frac{\text{Instructions}}{\text{Bit}}$ for the entire transmission.

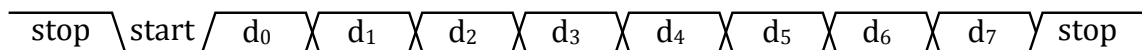| stop | start | d_0 | d_1 | d_2 | d_3 | d_4 | d_5 | d_6 | d_7 | stop |

Figure 5 - UART serial transmission

2. Multiply two 4-bit numbers together using one of the commands **bne** or **bge**. The algorithm works as follows: a multiplication is the same as adding the same number x times. For example: $2 * 9 = 9 + 9 = 18$.

2.3.1 Students guide

1. UART transmission idea:

```
# Serial UART Transmisstion
# setup
lui s2, 0x00001     # store UART base address
addi t0, zero, 0xA  # value to be send for testing
sb   t0, 0(s2)      # save to memory

# start
lui s2, 0x00001     # store UART base address
addi s1, zero, 0x1  # store mask bit
lb s0, 0(s2)        # get value from memory

# send stopbit
sb s1, 1(s2)        # send stopbit to memory
addi zero, zero, 0  # nop
addi zero, zero, 0  # nop

# send startbit
...

# algorithm iteration #1 to #8
...

# send stopbit
```

```
addi zero, zero, 0   # nop
sb s1, 1(s2)         # send stopbit to memory
```

2. Two numbers basic multiplication with loops idea:

```
# Input values
# a0, a1 = input, a2 = output
addi a0, zero, 9
addi a1, zero, 2

# init output to zero
addi a2, zero, 0

# check if a1 is zero
if a1 == 0 => goto end
decrement a1

accumulate:
  accumulate into a2
  decrement a1
  continue if not 0
```

isa/lab-basic-algos

2.4 Branching

2.4.1 Students guide

If / else

```
addi s0, zero, 1 # int a = 1
addi s1, zero, 2 # int b = 2

# if(a == b)
test1:
  bne s0, s1, test2 # imm = 12
# a == b
equal:
  c = 0
  goto end
# else if b > a
test2: # a < b === b >= a
  if a < b => goto a_smaller
# a > b
a_bigger:
  addi s2, zero, 1
  jal end # imm = 8
# a < b
a_smaller:
  addi s2, zero, 2

end:
# ...
```

2.4.2 Switch case

```
# a = s0, mem[2] = s1
lw s1, 2(zero)

# if(b == 0)
bne s1, zero, not0 # imm = 12

# b == 0
li s0, 17
jal end # imm = 48

# b != 0
not0:
  li t1, 3
  # if(b == 3)
  bne s1, t1, not3 # imm = 12

# b == 3
li s0, 33
jal end # imm = 32

# b != 3
not3:
  ...
  # if(b == 8)
  if(b == 8) goto is8_or_12 # imm = 20
  ...
  # if(b == 12)
  if(b == 12) goto is8_or_12 # imm = 12

# b != 8 | 12 (others)
li s0, 99
jal end # imm = 8

# b == 8 | 12
is8_or_12:
  li s0, 10

end:
# ...
```

2.4.3 While / Do While

```
// A : simple do-while
addi a5, zero, 10 # int a = 10;
while_entry:
  addi a5, a5, -1 # a--
  bne zero, a5, while_entry # imm = -4

// B : similar
addi a5, zero, 10 # int a = 10;
while_entry:
  addi a5, a5, -1 # a--
  if a >= 0 => goto while_entry
```

```
// C : uint32_t instead of int
...
```

2.4.4 For

```
# a is s0, i is s1, mem[0] = s2
lw s2, 0(zero) # loop target

# For the for to work, blte does not exist.
# Thus, since the loop decreases, a > b
#   === a-1 >= b (for signed only, else infinite loop)
# so better a >= b + 1
addi s2, s2, 1  # target + 1
li s1, 4        # i = 4
mv s0, zero     # a = 0

jal for_test # imm = 8

for_do:
  add s0, s0, s1
  addi s1, s1, -1 # MUST be at the end of for

for_test:
  bge  s1, s2, for_do # imm = -8
```

isa/lab-branch

2.5 Functions

2.5.1 Students guide

a) A function with context saving which can be optimized

```
# a is s0, b is s1
li s0, 1 # a = 1
mv a0, s0 # copy into a0 as funct. arg
jal ra, doubleIt # imm = undef.
mv s1, a0 # b = result

# DO NOT FORGET THE FOLLOWING
# a0 is a scratch register, and we called
a function
# so we are not sure if a0 is still s0
mv a0, s0
jal ra, doubleItOpti # imm = undef.
mv s1, a0 # b = result

# ...

doubleIt:
  # save context
  addi sp, sp, -4
  sw s0, 0(sp)
  # do a = a * 2
  mv s0, a0
  sll s0, s0, 1
  mv a0, s0
  # restore context
  lw s0, 0(sp)
  addi sp, sp, 4
  jalr zero, ra, 0 # or pseudo jr ra

# If 'a' should be a register
doubleItOpti:
  mv t0, a0
  sll t0, t0, 1
  mv a0, t0
  jalr zero, ra, 0 # or pseudo jr ra

# Most opti version
doubleItOpti2:
  # nothing to save since we can do it
with a0 directly
  sll a0, a0, 1
  jalr zero, ra, 0 # or pseudo jr ra
```

b) A function with too many arguments

```
# a to j in s0-s10
# res in s11
li s0 1
li s1 2
...
li s10 10

# prepare arguments
mv a0, s0
mv a1, s1
...
mv a7, s7
# still two args to pass -> stack
addi sp, sp, -8
# It is important that the caller
# reserves the space. Also, note
# the order in stack.
sw s8, 4(sp)
sw s9, 0(sp)
# call
jal ra, sum
# stack not needed anymore
addi sp, sp, 8

mv s1, a0 # b = result

# ...

sum:
  # do add with aX regs
  add a0, a0, a1
  add a0, a0, a2
  ...
  add a0, a0, a7
  # load i from over sp
  lw t0, 4(sp)
  add a0, a0, t0
  # load j from over sp
  lw t0, 0(sp)
  add a0, a0, t0
  jr ra
```

isa/lab-fcts

2.5.2 Students guide

2.5.2.1 Modulo

```
# RV32IM
# a is s0, b is s1, c is s2
li s0, 9
li s1, 7
remu s2, s0, s1

# RV32I
# Call a div algorithm and take remainder
# Or call a sub loop

# Modulo of power of 2
li s0, 9
li s1, 8
addi t0, s2, -1 # pow 2 - 1
and s2, t0, s0
```

2.5.2.2 °F -> °C

The main algorithm is:

```
begin:
    li s0, 550 # degrees farenheit

    li s1, 466034 # magic number
    mv s2, s0 # c = f

    # A: c = f - 32
    addi s2, s2, -32

    # B: c = c * 5
    # Variante 1 c*5 with shift
    # slli s3, s2, 2 # c * 4
    # add s2, s3, s2 # c + c (== * 5)
    # Variante 2 c*5 with function
    mv a0, s2
    li a1, 5
    # jal ra, mulFunct   # func variant malFunct
    # jal ra, sfmulFunct # func variant sfmulFunct
    jal ra, fmulFunct    # func variant fmulFunct
    mv s2, a0

    # C: c = c * 2^n / 9
    mv a0, s2
    mv a1, s1
    # jal ra, mulFunct   # func variant malFunct
    # jal ra, sfmulFunct # func variant sfmulFunct
    jal ra, fmulFunct    # func variant fmulFunct
    mv s2, a0

    # D: c >>= n
    srli s2, s2, 22

# End
```

```
nop
    j begin
```

The multiplication functions (from worst to best):

a)

```
# bad O(n_b)
mulFunct: # mulFunct(int a, int b)
    # add itself each time
    mv t0, a0
    addi a1, a1, -1

    mul_beg:
    bgeu zero, a1, mul_end # if 1 > b
    add a0, a0, t0
    addi a1, a1, -1
    j mul_beg

    mul_end:
    jr ra

# better O(n_min[a,b])
sfmulFunct: # sfmulFunct(int a, int b)
    swap a and b to loop less times
    loop to multiply
```

b)

```
# best
fmulFunct: # fmulFunct(int a, int b)
    swap a and b to loop less times

    mul_is_done:
        if b is 0 => goto fmul_end

        if b[0] is 0 => goto shift
    add:
        add t0, a0, t0
    shift:
        shift a0 left once
        shift a1 right once
        goto mul_is_done

    fmul_end:
        mv a0, t0
        jr ra
```

The test with n = 23 should work for small numbers but overflow with bigger:
- 100F = 37C; 400F = 204C
- 1000F = 25C -> WRONG

$$
\begin{aligned}
\text{nbBits}_{\text{max}_{\text{fahrenheit}}} + \text{nbBits}_{\text{mult5}} + \text{nbBits}_{\text{magicNumber}} &= \\
10(\text{max. 1000-32}) + 3 + (n - \text{nbBits}_{\text{div9}} + 1) &= \\
10 + 3 + (16 - 4 + 1) &= \\
&= 26 \text{ bits}
\end{aligned}
\tag{1}
$$

Because, following Equation 1, if n is 23 $\rightarrow$ equation gives 33 bits, but we are on 32.

isa/lab-adv-algos

1 | Architecture

1.1 Stack-Architecture

a) -
b) 7 explicit fetch and none implicit
c) 7 explicit fetch, 4 implicit fetch, 4 implicit store

arc/stack-01

1.2 Stack-Architecture

a) -
b) 7 explicit fetch
c) 7 explicit fetch (with compiler optimizations)
   7 explicit fetch, 1 implicit store, 1 implicit fetch (without compiler optimizations)
d) 7 explicit fetch, 3 implicit store, 3 implicit fetch (without compiler optimizations)

arc/stack-02

2 | Single-Cycle RISC-V

2.1 Single-Cycle Processor Operation

```
PCScr = '0'
RegWrite = '1'
ImmScr[1:0] = "xx"
ALUSrc = '0'
ALUControl[2:0] = "010"
MemWrite = '0'
ResultScr = '0'
```

arc/scr-01
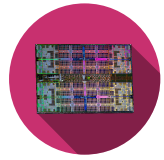
2.2 Extend Single Cycle with instruction `jal`

```
PCScr = '0'
RegWrite = '1'
ImmScr[1:0] = "xx"
ALUSrc = '0'
ALUControl[2:0] = "010"
MemWrite = '0'
ResultScr = '0'
```

arc/scr-02

2.3 Single Cycle Processor Performance

$$T_{\text{program\_single\_cycle}} = 75\text{sec} \tag{2}$$

arc/scr-03

3 | Multi-Cycle RISC-V

3.1 Multi Cycle Processor Performance

$$\begin{aligned}
\text{CPI}_{\text{load}} &= 5 \text{ cycles} \\
\text{CPI}_{\text{store}} &= 4 \text{ cycles} \\
\text{CPI}_{\text{branch}} &= 3 \text{ cycles} \\
\text{CPI}_{\text{jump}} &= 4 \text{ cycles} \\
\text{CPI}_{\text{alu}} &= 4 \text{ cycles} \\
\text{CPI}_{\text{average}} &= 4.14 \text{ cycles}
\end{aligned} \tag{3}$$

arc/mcr-01

3.2 Multi-Cycle Processor Performance

$$T_{\text{program\_single\_cycle}} = 155.25 \text{sec} \tag{4}$$

arc/mcr-02