Introduction to Computer Architecture exam (2 hours)

Renaud Pacalet - 2026-03-02

IMPORTANT: AT THE END OF THE EXAM, DO NOT TAKE THIS QUESTION PAPER AWAY, GIVE IT BACK TO THE EXAM SUPERVISOR. EXAM SUPERVISORS: PLEASE RETURN ALL QUESTION PAPERS WITH THE STUDENT PAPERS.

The text in black is the original one.

The text in red is examples of the expected correct answers. Only this text was expected, possibly in shorter form, nothing more.

The text in blue is extra comments about the expected correct answers.

Warning: the course changes frequently (content, vocabulary, examples…); some questions and answer proposals can thus be partly or completely out of scope. Warning: some questions can be answered in many different ways; the proposed answers are just examples and they are not exhaustive.

The exam is closed book, closed notes, except one A4-sized, double sided, cheat sheet. Calculators are allowed but communicating devices are strictly forbidden. Please number the different pages of your paper and indicate on each page your first and last names. You can write your answers in French or in English, as you wish. Precede your answers with the question's number. If some information or hypotheses are missing to answer a question, add them. If you consider a question as absurd and thus decide to not answer, explain why. If you do not have time to answer a question but know how to, briefly explain your ideas. Advice: quickly go through the document and answer the easy parts first.

The first question brings 6 points. The other questions bring 7 points each.

1. Design of a CMOS logic gate for Booth encoding

Booth encoding is a technique invented in 1950 by Andrew Donald Booth, to multiply signed integers. One of the steps of the Booth algorithm consists in converting a 3 bits input $a_2a_1a_0$ into 2 output bits M and S.

The input represents a signed integer $A \in \{-2,-1,0,1,2\}$, with an unusual redundant encoding: bit a_2 has weight -2 while bits a_1 and a_0 both have weight 1. So, $A = -2.a_2 + a_1 + a_0$. Examples:

- $a_2a_1a_0 = 110$ represents value $A = -2+1+0 = -1$,
- $a_2a_1a_0 = 111$ represents value $A = -2+1+1 = 0$,
- etc.

The value of output bits M and S is as follows:

$$M = \begin{cases} 1 \text{ when } A < 0 \\ 0 \text{ when } A \geq 0 \end{cases} \qquad S = \begin{cases} 1 \text{ when } A \in \{-2,2\} \\ 0 \text{ when } A \in \{-1,0,1\} \end{cases}$$

Example: input $a_2a_1a_0 = 110$ represents $A = -1$, so $M = 1$ and $S = 0$.

1. Write the 8 lines, 5 columns truth table of M and S as in the following example where the $a_2a_1a_0 = 110$ case is already filled:

a_2	a_1	a_0	M	S
0	0	0	...	...
0	0	1	...	...
...	...	...	...	...
1	1	0	1	0
1	1	1	...	...

Table 1: Booth encoding

Note: the extra column for integer A was not asked for:

A	a_2	a_1	a_0	M	S
0	0	0	0	0	0
1	0	0	1	0	0
1	0	1	0	0	0
2	0	1	1	0	1
−2	1	0	0	1	1
−1	1	0	1	1	0
−1	1	1	0	1	0
0	1	1	1	0	0

Table 2: Booth encoding

2. Write the boolean equations of M and S using the NOT, AND and OR operators and parentheses. Do not assume any precedence between the boolean operators, use parentheses to make your equation non ambiguous: NOT a_2 AND a_0 is ambiguous; (NOT a_2) AND a_0 is not.

$M = a_2$ AND (NOT (a_1 AND a_0))
$S =$ ((NOT a_2) AND a_1 AND a_0) OR (a_2 AND (NOT a_1) AND (NOT a_0))

3. Draw the CMOS schematic of a logic gate computing M using only N and P transistors. We assume that the inverses of the 3 input bits are also available and we denote them $\overline{a_2}$, $\overline{a_1}$ and $\overline{a_0}$. So, the inputs of your logic gate can be any of a_2, a_1, a_0, $\overline{a_2}$, $\overline{a_1}$ and $\overline{a_0}$.

   Reminder: CMOS logic gates are built from interconnected N and P transistors, between the ground voltage (logical 0) and the power supply voltage (logical 1). When its grid is set to 1, a N transistor lets the current pass between its two other terminals; when its grid is set to 0, it blocks the current. When its grid is set to 0, a P transistor lets the current pass between its two other terminals; when its grid is set to 1, it blocks the current. The network of N transistors, located between the ground and the output, is responsible for outputing 0 by connecting the output to the ground voltage. The network of P transistors, located between the power supply and the output, is responsible for outputing 1 by connecting

the output to the power supply voltage. Figure 1 shows an example CMOS logic gate: the inverter.

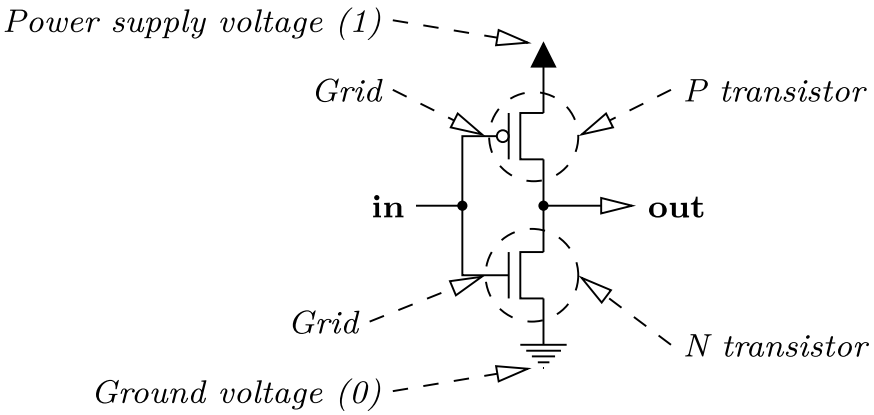


Figure 1: CMOS schematic of an inverter logic gate

The CMOS schematic is represented on Figure 2.

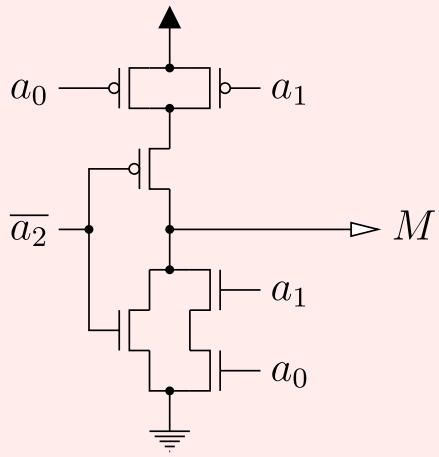


Figure 2: CMOS schematic of the logic gate computing M

From the truth table we can observe that M is 1 if and only if the a_2 input is 1 and at least one of a_1 or a_0 is 0. This immediately gives the network of P transistors between the power supply and the output: a P transistor which grid is $\overline{a_2}$ in series with a group of 2 P transistors in parallel which grids are a_1 and a_0 respectively. This way, if $\overline{a_2} = 0$ (that is, if $a_2 = 1$) and at least one of a_1 or a_0 equals 0, there is a path between the power supply and the output, which is forced to 1; in all other circumstances there is no such path and the output is not driven by the network of P transistors.

As always with CMOS logic the network of N transistors between the ground and the output is dual of the network of P transistors: a N transistor which grid is $\overline{a_2}$ in parallel with a group of 2 N transistors in series which grids are a_1 and a_0 respectively.

2. Binary representation of numbers

Notations: let r be a 32 bits register of a RISC-V 32 bits processor, $A \in \mathbb{Z}$ an integer, $B \in \mathbb{N}^*$ a positive integer, $V \in \mathbb{R}$ a real number. In the following we denote:

- $\mathrm{r}[i]$ bit number i of r, $\mathrm{r}[0]$ the Least Significant Bit (LSB) of r, and $\mathrm{r}[31]$ the Most Significant Bit (MSB) of r. Bits can be used in numeric expressions where they represent numbers 0 or 1:

$$\mathrm{r}[17] + \mathrm{r}[6] \in \{0, 1, 2\}$$

  They can also be used in boolean expressions where they represent "false" (0) or "true" (1):

$$(\mathrm{r}[17] \text{ OR } (\text{NOT } \mathrm{r}[6])) \in \{\text{false}, \text{true}\}$$

- $u(\mathrm{r})$ the content of r, considered as an unsigned number:

$$u(\mathrm{r}) = \sum_{i=0}^{i=31} \mathrm{r}[i].2^i$$

- $s(\mathrm{r})$ the content of r, considered as a signed number in two's complement representation:

$$s(\mathrm{r}) = -\ \mathrm{r}[31].2^{31} + \sum_{i=0}^{i=30} \mathrm{r}[i].2^i$$

- $A \bmod B$ the non-negative remainder of the integer division of A by B:

$$0 \leq A \bmod B < B \quad \text{and} \quad \exists K \in \mathbb{Z} \quad \text{such that} \quad A = K.B + (A \bmod B)$$

- $\lfloor V \rfloor$ the largest integer not greater than V:

$$\lfloor V \rfloor \in \mathbb{Z} \quad \text{and} \quad \lfloor V \rfloor \leq V < \lfloor V \rfloor + 1$$

1. Let r be a 32 bits register. What is the range of values that unsigned number $u(\mathrm{r})$ can take?

   The range of $u(\mathrm{r})$ is $\{0, ..., 2^{32} - 1\}$.

   The range of values that an n bits unsigned number can take is $\{0, ..., 2^n - 1\}$.

2. Let r and s be two 32 bits registers. What is the range of values that the sum $u(\mathrm{r}) + u(\mathrm{s})$ could take if it was computed with arbitrary precision?

   The range of $u(\mathrm{r}) + u(\mathrm{s})$ is $\{0, ..., 2^{33} - 2\}$.

   The sum of two unsigned numbers each in range $\{0, ..., 2^n - 1\}$ is in range $\{0 + 0, ..., 2^n - 1 + 2^n - 1\} = \{0, ..., 2^{n+1} - 2\}$.

3. How many bits, minimum, would be needed to represent $u(\mathrm{r}) + u(\mathrm{s})$ exactly in binary, as an unsigned number?

   We need 33 bits.

   The range of values that an n bits unsigned number can take is $\{0, ..., 2^n - 1\}$. As $\{0, ..., 2^{32} - 1\} \subsetneq \{0, ..., 2^{33} - 2\}$ 32 bits are not sufficient. As $\{0, ..., 2^{33} - 2\} \subseteq \{0, ..., 2^{33} - 1\}$ 33 bits are sufficient.

We execute the following RISC-V instructions:

```
add  t2,t1,t0   # t2 <-- t1 + t0
andi t3,t0,0xff # t3 <-- t0 AND 0xff
srli t4,t1,8    # t4 <-- t1 >> 8 (logical)
```

4. After the `add` instruction in the above assembly code, if $u(\text{t0})$ and $u(\text{t1})$ are too large, the result $u(\text{t2})$ is not equal to $u(\text{t0}) + u(\text{t1})$: this is what is called an "*overflow*" situation. It is possible to check for overflows by looking only at bits t0[31], t1[31] and t2[31].

   Express the necessary and sufficient condition on t0[31], t1[31] and t2[31], such that there is no overflow, that is, $u(\text{t2}) = u(\text{t0}) + u(\text{t1})$. Use the NOT, AND and OR operators and parentheses. Do not assume any precedence between the boolean operators, use parentheses to make your equation non ambiguous: NOT t0[31] AND t1[31] is ambiguous; (NOT t0[31]) AND t1[31] is not.

There is no overflow if and only if:

NOT (((NOT t2[31]) AND (t0[31] OR t1[31])) OR (t0[31] AND t1[31]))

Note: in the following we use the XOR boolean operator to simplify some expressions. We can write the addition out, as we would compute it by hand, and where we denote $c[i]$ the carry that propagate from rank $i-1$ to rank i ($c[0] = 0$):

c[32]	c[31]	c[30]	...	c[1]	c[0]
	t0[31]	t0[30]	...	t0[1]	t0[0]
	t1[31]	t1[30]	...	t1[1]	t1[0]
c[32]	t2[31]	t2[30]	...	t2[1]	t2[0]

There is an overflow if and only if the last output carry c[32] is equal to 1. By definition of the binary addition we have:

$$\text{t2[31]} = c[31] \text{ XOR t0[31] XOR t1[31]}$$
$$c[32] = (c[31] \text{ AND (t0[31] OR t1[31])) OR (t0[31] AND t1[31])}$$

We can rewrite the expression of t2[31] to obtain an expression of c[31] and reuse it in the expression of c[32]:

$$c[31] = \text{t2[31] XOR t0[31] XOR t1[31]}$$
$$c[32] = ((\text{t2[31] XOR t0[31] XOR t1[31]}) \text{ AND (t0[31] OR t1[31]))}$$
$$\text{OR (t0[31] AND t1[31])}$$

What is asked for is the necessary and sufficient condition such that there is **no** overflow, that is, the Boolean negation of c[32]:

$$\text{NOT } (((\text{t2[31] XOR t0[31] XOR t1[31]}) \text{ AND (t0[31] OR t1[31]))}$$
$$\text{OR (t0[31] AND t1[31]))}$$

Optionally, we can apply the rules of Boolean algebra to simplify the expression; the necessary and sufficient condition such that there is no overflow is:

NOT (((NOT t2[31]) AND (t0[31] OR t1[31])) OR (t0[31] AND t1[31])) ■

5. What is the mathematical relation between $u(\text{t0})$ and $u(\text{t3})$?

We have $u(\text{t3}) = u(\text{t0}) \bmod 256$.

Constant `0xff` (in hexadecimal), after extension to 32 bits, becomes the 32 bits word where the 8 Least Significant Bits (LSB) are equal to 1 and the 24 Most Significant Bits (MSB) are equal to 0. The RISC-V instruction `andi t3,t0,0xff` sets t3 to the bitwise AND between t0 and `0xff`. So, t3 has its 8 LSB equal to the 8 LSB of t0, and its 24 MSB equal to 0. We can rewrite the expressions of $u(\text{t3})$ and $u(\text{t0})$ as follows:

$$
\begin{aligned}
u(\text{t3}) &= \sum_{i=0}^{i=31} \text{t3}[i].2^i = \sum_{i=0}^{i=7} \text{t0}[i].2^i + \sum_{i=8}^{i=31} 0.2^i = \sum_{i=0}^{i=7} \text{t0}[i].2^i \\
u(\text{t0}) &= \sum_{i=0}^{i=31} \text{t0}[i].2^i \\
&= \sum_{i=0}^{i=7} \text{t0}[i].2^i + \sum_{i=8}^{i=31} \text{t0}[i].2^i \\
&= u(\text{t3}) + 2^8. \sum_{i=8}^{i=31} \text{t0}[i].2^{i-8} \\
&= u(\text{t3}) + 256. \sum_{i=0}^{i=23} \text{t0}[i+8].2^i
\end{aligned}
$$

So, $u(\text{t3})$ and $u(\text{t0})$ are congruent modulo 256. As $0 \leq u(\text{t3}) < 256$ we conclude that $u(\text{t3}) = u(\text{t0}) \bmod 256$ ■

6. What is the mathematical relation between $u(\text{t1})$ and $u(\text{t4})$?

We have $u(\text{t4}) = \left\lfloor \frac{u(\text{t1})}{256} \right\rfloor$

By definition, for any $A \in \mathbb{Z}$ and $B \in \mathbb{N}^*$, $\left\lfloor \frac{A}{B} \right\rfloor = \frac{A - (A \bmod B)}{B}$

The RISC-V instruction `srli t4,t1,8` sets t4 to t1 shifted to the right by eight bit positions; as the shift is logical (`srli`, not `srai`), the eight bits entering on the left during the shift are 0. So $\forall 0 \leq i \leq 23$, $\text{t4}[i] = \text{t1}[i+8]$ and $\forall 24 \leq i \leq 31$, $\text{t4}[i] = 0$. We can rewrite the expressions of $u(\text{t4})$ and $u(\text{t1})$ as follows:

$$
\begin{aligned}
u(\mathrm{t4}) &= \sum_{i=0}^{i=31} \mathrm{t4}[i].2^i = \sum_{i=0}^{i=23} \mathrm{t1}[i+8].2^i + \sum_{i=24}^{i=31} 0.2^i = \sum_{i=0}^{i=23} \mathrm{t1}[i+8].2^i \\
u(\mathrm{t1}) &= \sum_{i=0}^{i=31} \mathrm{t1}[i].2^i \\
&= \sum_{i=0}^{i=7} \mathrm{t1}[i].2^i + \sum_{i=8}^{i=31} \mathrm{t1}[i].2^i \\
&= (u(\mathrm{t1}) \bmod 256) + \sum_{i=0}^{i=23} \mathrm{t1}[i+8].2^{i+8} \\
&= (u(\mathrm{t1}) \bmod 256) + 2^8.\sum_{i=0}^{i=23} \mathrm{t1}[i+8].2^i \\
&= (u(\mathrm{t1}) \bmod 256) + 256.u(\mathrm{t4}) \Rightarrow \\
u(\mathrm{t4}) &= \frac{u(\mathrm{t1}) - (u(\mathrm{t1}) \bmod 256)}{256} \\
&= \left\lfloor \frac{u(\mathrm{t1})}{256} \right\rfloor \quad \blacksquare
\end{aligned}
$$

7. What is the mathematical relation between $s(\mathrm{t0})$ and $s(\mathrm{t3})$?

We have $s(\mathrm{t3}) = s(\mathrm{t0}) \bmod 256$

Constant `0xff` (in hexadecimal), after extension to 32 bits, is the 32 bits word where the 8 Least Significant Bits (LSB) are equal to 1 and the 24 Most Significant Bits (MSB) are equal to 0. The RISC-V instruction `andi t3,t0,0xff` sets t3 to the bitwise AND between t0 and `0xff`. So, t3 has its 8 LSB equal to the 8 LSB of t0, and its 24 MSB equal to 0. We can rewrite the expressions of $s(\mathrm{t3})$ and $s(\mathrm{t0})$ as follows:

$$
\begin{aligned}
s(\mathrm{t3}) &= -\mathrm{t3}[31].2^{31} + \sum_{i=0}^{i=30} \mathrm{t3}[i].2^i = -0.2^{31} + \sum_{i=0}^{i=7} \mathrm{t0}[i].2^i + \sum_{i=8}^{i=31} 0.2^i = \sum_{i=0}^{i=7} \mathrm{t0}[i].2^i \\
s(\mathrm{t0}) &= -\mathrm{t0}[31].2^{31} + \sum_{i=0}^{i=30} \mathrm{t0}[i].2^i \\
&= -\mathrm{t0}[31].2^{31} + \sum_{i=8}^{i=30} \mathrm{t0}[i].2^i + \sum_{i=0}^{i=7} \mathrm{t0}[i].2^i \\
&= 2^8.\left(-\mathrm{t0}[31].2^{23} + \sum_{i=8}^{i=30} \mathrm{t0}[i].2^{i-8}\right) + s(\mathrm{t3}) \\
&= 256.\left(-\mathrm{t0}[31].2^{23} + \sum_{i=0}^{i=22} \mathrm{t0}[i+8].2^i\right) + s(\mathrm{t3})
\end{aligned}
$$

This proves that $s(\mathrm{t3})$ and $s(\mathrm{t0})$ are congruent modulo 256. As $0 \leq s(\mathrm{t3}) < 256$ we can now conclude that $s(\mathrm{t3}) = s(\mathrm{t0}) \bmod 256$ $\blacksquare$

3. RISC-V assembly

The James Webb Space Telescope (JWST) embeds the Mid-Infrared Instrument (MIRI). MIRI can measure the temperature of exoplanets, stars, etc. It is controlled by a software running on a RISC-V 32 bits processor that implements the RV32I Instruction Set Architecture (ISA). Scientists asked for an update to add a configurable alarm feature. Your team is in charge of the software update. Members of your team already coded 3 functions:

- `temp` takes no parameter and returns the measured temperature in Kelvin degrees, as a 32 bits unsigned number.
- `alarm` takes no parameter and does not return any result. It just sends message "**ALARM: MAXIMUM TEMPERATURE REACHED!**" to ground control.
- `sleep` takes one 32 bits unsigned number parameter, waits for this number of microseconds, and does not return any result.

Your team leader asks you to write the code of function `monitor` that takes two 32 bits unsigned integers as inputs, denoted `tmax` and `delay` in the following, and returns one 32 bits unsigned integer as result: the temperature in Kelvin degrees. Function `monitor` shall loop. In each iteration of the loop it shall call `temp` and compare the returned temperature with `tmax`. If it is less than `tmax`, it shall call `sleep` with parameter `delay`, and iterate. Else, if the temperature is greater or equal `tmax`, it shall call `alarm` and return the temperature to its caller. A pseudo-code for `monitor` could be:

```
monitor(tmax, delay) {
  loop {
    t = temp()
    if t >= tmax then {
      alarm()
      return(t)
    }
    sleep(delay)
  }
}
```

Your code **must** fully comply with the ILP32 Application Binary Interface (ABI) seen during lectures and labs. Reminder: according the ABI, the size of a stack frame **must** be a multiple of 16 bytes; the general purpose **saved** registers are `sp`, `s0`, `s1`, …, `s11`; the general purpose **non-saved** registers are `ra`, `t0`, `t1`, …, `t6`, `a0`, `a1`, …, `a7`; `gp` and `tp` are reserved and shall not be used at all.

You do not know how functions `temp`, `alarm` and `sleep` are coded. All you know is that they are **fully** compliant with the ILP32 ABI. You do not know which registers they use, you **must** assume that they can modify **any** non-saved registers.

Use the provided RISC-V cheat sheet if you don't remember the RV32I ISA or the ILP32 ABI.

Code function `monitor` in RV32I assembly language. Comment each instruction on the same line after a `#` sign as in the following example:

```
1  addi t0,t1,0   # t0 = t1 (+0)
```

Listing 1 shows a possible source code of `monitor`. It uses saved registers `s0` to store `tmax` and `s1` to store `delay`, and preserve them across function calls. Near the end it also uses `s0` to store the temperature and preserve it from the call to `alarm`.

```
1  .text
2  monitor: # inputs tmax,delay in a0,a1, output temperature in a0
3    addi sp,sp,-16  # allocate 16 bytes stack frame
4    sw   ra,0(sp)   # save ra in stack frame
5    sw   s0,4(sp)   # save s0 in stack frame
6    sw   s1,8(sp)   # save s1 in stack frame
7    mv   s0,a0      # s0 <-- a0 (copy tmax from a0 to s0)
8    mv   s1,a1      # s1 <-- a1 (copy delay from a1 to s1)
9  loop:
10   call temp       # a0 <-- temp()
11   bgeu a0,s0,end  # if a0>=s0 (temp>=tmax, unsigned comparison) goto end
12   mv   a0,s1      # a0 <-- s1 (copy delay from s1 to a0)
13   call sleep      # sleep(delay)
14   j    loop       # goto loop
15 end:
16   mv   s0,a0      # s0 <-- a0 (copy temp from a0 to s0)
17   call alarm      # alarm()
18   mv   a0,s0      # a0 <-- s0 (copy temp from s0 to a0)
19   lw   ra,0(sp)   # restore ra from stack frame
20   lw   s0,4(sp)   # restore s0 from stack frame
21   lw   s1,8(sp)   # restore s1 from stack frame
22   addi sp,sp,16   # restore sp
23   ret             # return
```

Listing 1: The `monitor` function

Per the ABI, function inputs are passed in `a0`, `a1`, …, `a7` and results are returned in `a0` and `a1`. So, we receive our two inputs `tmax` and `delay` in `a0` and `a1`, and we must store the temperature in `a0` before returning.

As for any function that calls other functions we **must** allocate a stack frame and store register `ra` in it before calling any other function. This is because it contains our return address and any call to other functions overwrites it. If we do not first save it, our own return address is lost and we cannot return to our own caller.

We must also save `tmax` and `delay` somewhere because `a0` and `a1` are **non-saved** registers, and any call to `temp`, `spleep` or `alarm` could modify them. We could also save them in the stack frame, but this would imply load instructions in the loop, each time we need them again. Instead we save them in two **saved** registers, `s0` and `s1`, such that they remain immediately usable. Of course, as for any saved register that we decide to modify, we must first save `s0` and `s1` in the stack frame because, per the ABI, we must restore them before returning.

16 bytes are sufficient to store these three 4-bytes values. As always the stack grows toward the low addresses so we start our function by **subtracting** 16 from the stack pointer `sp` (Line 3). `sp` now contains the base address of the new stack frame.

We save registers `ra`, `s0` and `s1` in the stack frame, at offsets 0, 4 and 8 from the base address of the new stack frame, respectively (Lines 4 - 6).

We then copy `a0` and `a1` in `s0` and `s1`, respectively (Lines 7 - 8).

We can now call function `temp` that returns the measured temperature in `a0` (Line 10). We compare it with `tmax` in `s0` using the `bgeu` instruction (**unsigned** comparison), and branch to label `end` if the temperature is larger or equal `tmax`, else we continue (Line 11).

If the maximum temperature is not reached we must now call `sleep`. Per the ABI we must pass it its input parameter (`delay`) in `a0`, so we copy `s1` (that contains `delay`) to `a0` (Line 12), and call `sleep` (Line 13). Finally, we jump to label `loop` with the unconditional `j` instruction (Line 14)

If the maximum temperature is reached, at Line 11 we branch to label `end`. At this point the temperature we must return to the caller is in `a0`. Before returning we must call `alarm`, and before that we must save `a0` to preserve it from this function call. We use `s0` for that (Line 16), call `alarm` (Line 17), and copy back the temperature from `s0` to `a0` (Line 18).

Last but not least, we restore `ra`, `s0` and `s1` from the stack frame (Lines 19 - 21), we restore `sp` by adding 16, the opposite of what was subtracted when entering the function (Line 22), and return with pseudo-instruction `ret` (Line 23).

RISC-V Instruction-Set

Erik Engheim <erik.engheim@ma.com>

Arithmetic Operation

Mnemonic	Instruction	Type	Description
ADD rd, rs1, rs2	Add	R	rd ← rs1 + rs2
SUB rd, rs1, rs2	Subtract	R	rd ← rs1 - rs2
ADDI rd, rs1, imm12	Add immediate	I	rd ← rs1 + imm12
SLT rd, rs1, rs2	Set less than	R	rd ← rs1 < rs2 ? 1 : 0
SLTI rd, rs1, imm12	Set less than immediate	I	rd ← rs1 < imm12 ? 1 : 0
SLTU rd, rs1, rs2	Set less than unsigned	R	rd ← rs1 < rs2 ? 1 : 0
SLTIU rd, rs1, imm12	Set less than immediate unsigned	I	rd ← rs1 < imm12 ? 1 : 0
LUI rd, imm20	Load upper immediate	U	rd ← imm20 << 12
AUIP rd, imm20	Add upper immediate to PC	U	rd ← PC + imm20 << 12

Logical Operations

Mnemonic	Instruction	Type	Description
AND rd, rs1, rs2	AND	R	rd ← rs1 & rs2
OR rd, rs1, rs2	OR	R	rd ← rs1 | rs2
XOR rd, rs1, rs2	XOR	R	rd ← rs1 ^ rs2
ANDI rd, rs1, imm12	AND immediate	I	rd ← rs1 & imm12
ORI rd, rs1, imm12	OR immediate	I	rd ← rs1 | imm12
XORI rd, rs1, imm12	XOR immediate	I	rd ← rs1 ^ imm12
SLL rd, rs1, rs2	Shift left logical	R	rd ← rs1 << rs2
SRL rd, rs1, rs2	Shift right logical	R	rd ← rs1 >> rs2
SRA rd, rs1, rs2	Shift right arithmetic	R	rd ← rs1 >> rs2
SLLI rd, rs1, shamt	Shift left logical immediate	I	rd ← rs1 << shamt
SRLI rd, rs1, shamt	Shift right logical imm.	I	rd ← rs1 >> shamt
SRAI rd, rs1, shamt	Shift right arithmetic immediate	I	rd ← rs1 >> shamt

Load / Store Operations

Mnemonic	Instruction	Type	Description
LD rd, imm12(rs1)	Load doubleword	I	rd ← mem[rs1 + imm12]
LW rd, imm12(rs1)	Load word	I	rd ← mem[rs1 + imm12]
LH rd, imm12(rs1)	Load halfword	I	rd ← mem[rs1 + imm12]
LB rd, imm12(rs1)	Load byte	I	rd ← mem[rs1 + imm12]
LWU rd, imm12(rs1)	Load word unsigned	I	rd ← mem[rs1 + imm12]
LHU rd, imm12(rs1)	Load halfword unsigned	I	rd ← mem[rs1 + imm12]
LBU rd, imm12(rs1)	Load byte unsigned	I	rd ← mem[rs1 + imm12]
SD rs2, imm12(rs1)	Store doubleword	S	rs2 → mem[rs1 + imm12]
SW rs2, imm12(rs1)	Store word	S	rs2(31:0) → mem[rs1 + imm12]
SH rs2, imm12(rs1)	Store halfword	S	rs2(15:0) → mem[rs1 + imm12]
SB rs2, imm12(rs1)	Store byte	S	rs2(7:0) → mem[rs1 + imm12]

Branching

Mnemonic	Instruction	Type	Description
BEQ rs1, rs2, imm12	Branch equal	SB	if rs1 = rs2 pc ← pc + imm12
BNE rs1, rs2, imm12	Branch not equal	SB	if rs1 ≠ rs2 pc ← pc + imm12
BGE rs1, rs2, imm12	Branch greater than or equal	SB	if rs1 ≥ rs2 pc ← pc + imm12
BGEU rs1, rs2, imm12	Branch greater than or equal unsigned	SB	if rs1 >= rs2 pc ← pc + imm12
BLT rs1, rs2, imm12	Branch less than	SB	if rs1 < rs2 pc ← pc + imm12
BLTU rs1, rs2, imm12	Branch less than unsigned	SB	if rs1 < rs2 pc ← pc + imm12 << 1
JAL rd, imm20	Jump and link	UJ	rd ← pc + 4 pc ← pc + imm20
JALR rd, imm12(rs1)	Jump and link register	I	rd ← pc + 4 pc ← rs1 + imm12

Pseudo Instructions

Mnemonic	Instruction	Base instruction(s)
LI rd, imm12	Load immediate (near)	ADDI rd, zero, imm12
LI rd, imm	Load immediate (far)	LUI rd, imm[31:12] ADDI rd, rd, imm[11:0]
LA rd, sym	Load address (far)	AUIPC rd, sym[31:12] ADDI rd, rd, sym[11:0]
MV rd, rs	Copy register	ADDI rd, rs, 0
NOT rd, rs	One's complement	XORI rd, rs, -1
NEG rd, rs	Two's complement	SUB rd, zero, rs
BGT rs1, rs2, offset	Branch if rs1 > rs2	BLT rs2, rs1, offset
BLE rs1, rs2, offset	Branch if rs1 ≤ rs2	BGE rs2, rs1, offset
BGTU rs1, rs2, offset	Branch if rs1 > rs2 (unsigned)	BLTU rs2, rs1, offset
BLEU rs1, rs2, offset	Branch if rs1 ≤ rs2 (unsigned)	BGEU rs2, rs1, offset
BEQZ rs1, offset	Branch if rs1 = 0	BEQ rs1, zero, offset
BNEZ rs1, offset	Branch if rs1 ≠ 0	BNE rs1, zero, offset
BGEZ rs1, offset	Branch if rs1 ≥ 0	BGE rs1, zero, offset
BLEZ rs1, offset	Branch if rs1 ≤ 0	BGE zero, rs1, offset
BGTZ rs1, offset	Branch if rs1 > 0	BLT zero, rs1, offset
J offset	Unconditional jump	JAL zero, offset
CALL offset12	Call subroutine (near)	JALR ra, ra, offset12
CALL offset	Call subroutine (far)	AUIPC ra, offset[31:12] JALR ra, ra, offset[11:0]
RET	Return from subroutine	JALR zero, 0(ra)
NOP	No operation	ADDI zero, zero, 0

32-bit instruction format

	31 30 29 28 27 26 25	24 23 22 21 20	19 18 17 16 15	14 13 12	11 10 9 8 7	6 5 4 3 2 1 0
R	func	rs2	rs1	func	rd	opcode
I	immediate		rs1	func	rd	opcode
SB	immediate	rs2	rs1	func	immediate	opcode
UJ	immediate				rd	opcode

Register File

r0	r1	r2	r3
r4	r5	r6	r7
r8	r9	r10	r11
r12	r13	r14	r15
r16	r17	r18	r19
r20	r21	r22	r23
r24	r25	r26	r27
r28	r29	r30	r31

Register Aliases

zero	ra	sp	gp
tp	t0	t1	t2
s0/fp	s1	a0	a1
a2	a3	a4	a5
a6	a7	s2	s3
s4	s5	s6	s7
s8	s9	s10	s11
t3	t4	t5	t6

ra - return address
sp - stack pointer
gp - global pointer
tp - thread pointer

t0 - t6 - Temporary registers
s0 - s11 - Saved by callee
a0 - a7 - Function arguments
a0 - a1 - Return value(s)