Acos
Control work.

Malchenko Oleg, 193

№ 1

Explain:

- Integer and floating-point values are represented in binary.
- unsigned, 2's complement signed, single and double floating point. Give an explanation. with an example, how a signed and unsigned integer values are added.

1) Integers:

Integers are represented in a binary format, like bina-ry words of certain length.

- Unsidned integers have the following form:

$$X_{10}(\geq 2^n) = \sum_{i=0}^{n} b_2^{(i)} 2^i = b_2^{(n)} \cdot 2^n + \dots + b_2^{(0)} 2^0 = (b_2^{(n)} \dots b_2^{(0)})_2,$$ where:

$b_2^{(k)} \in \{0,1\}\ \forall k \in [n]$, that is the MSB of X_2 is $b_2^{(n)}$ and LSB is: $b_2^{(0)}$

- 2's complement signed:

$$X_{10}(<2^n) = -b_2^{(n)} 2^n + \sum_{i=0}^{n-1} b_2^{(i)} 2^i = -b_2^{(n)} 2^n + b_2^{(n-1)} 2^{n-1} + \dots + b_2^{(0)} \cdot 2^0 = (b_2^{(n)} \dots b_2^{(0)})_2,$$

where MSB $= b_2^{(n)}$ is a sign bit, that is: if MSB $=1$, then X is negative, if MSB $=0$, then it's positive.

- The actual size (in bits) of an integer depends on the system's architecture: 32 or 64 bits.

- Single and double floats: floating point values have quite a more complicated structure:

IEEE fp. formate suffices: $X_f = (-1)^{b_2^{(n-1)}} \cdot (1 + [\mathrm{Frac}[b_2^{(i)} \cup \ldots]]) \cdot 2^{[\mathrm{Exp}(b_2^{(k)} \cup \ldots) - B]}$

Where:

- for single precision:
  - $: b_2^{(n-1)} = b_2^{(31)}$, (sign bit)
  - $: \mathrm{Fraction} = (b_2^{(22)} \ldots b_2^{(0)})_2$ | - responsible for floating (fraction) part.
  - $: \mathrm{Exponent} = (b_2^{(30)} \ldots b_2^{(23)})_2$ | - responsible for integer part of X_f.
- for double:
  - $: b_2^{(n-1)} = b_2^{(63)}$ sign.
  - $: \mathrm{Frac} = (b_2^{(51)} \ldots b_2^{(0)})_2$
  - $: \mathrm{Exp} = (b_2^{(62)} \ldots b_2^{(51)})_2$

  | double is more presise

- $\pm\infty$: $\mathrm{Exp} = 11\ldots1_2$, $\mathrm{Frac} = 00\ldots0_2$,
- NaN : $\mathrm{Exp} = 1\ldots\ldots1_2$, $\mathrm{Frac} \neq 0\ldots0_2$.
- Normalized float formaters: $\overline{a,bcd\ldots} \cdot 10^n$, example: $1.234 \cdot 10^{-6}$

1) about Addition:

- signed Addition: $\bar{a}_2 + \bar{b}_2 = ((\tilde{a}_2^{(n)} + \tilde{b}_2^{(n)})(\tilde{a}_2^{(n-1)} + b_2^{(n-1)}) \ldots (\tilde{a}_2^{(0)} + b_2^{(0)})$ +rem$^{(n-1)}$ +rem$^{(n-1)}$ rem(0) – remainder. , where the sign bit is preserved, if remnants don't sum into a greater value:

  example:
$$\begin{array}{r} 1\,0001\,0001_2 \\ +\,0\,0110\,1001_2 \\ \hline 1\,0111\,1010_2 \end{array}$$
  rem.

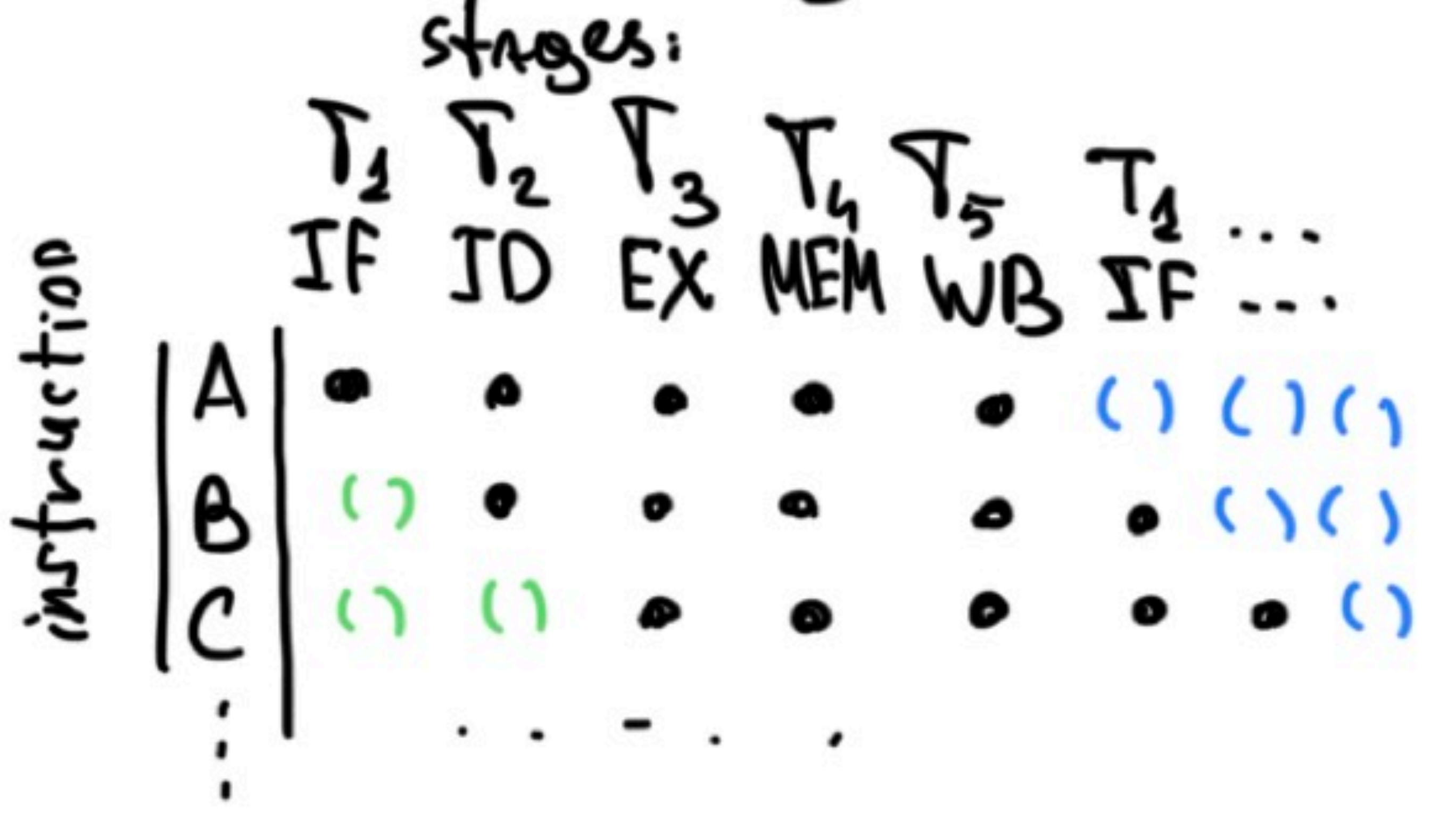
- unsigned: An overflow can occur here, for example.
$$\begin{array}{r} 1111 \\ +\,0001 \\ \hline \boxed{1}\,0000 \end{array}$$
  overflow.

  | Addition shares same algebraic rule an a ring $\mathbb{Z}_2[0,1]$.

№2 Describe consept of pipeline. List five stages and explain. Advantage? Disadvantage?

Pipeline is: like a conveier - inputting elements, processing, proceed, parallelity nees:

1) In order ~~not~~ not to waste time and performance, instructions are pipelined simultaneously in a sequence, s.t. if the first instruction has passed a certain stage, the next one takes its spot immediately.

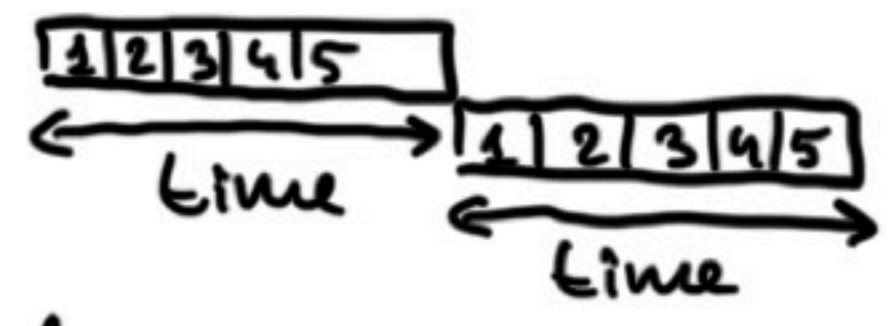


the stages of RISC-V pipeline are:
- IF - fetching an instruction. (from memory)
- ID - decoding + reading registers (to opcodes and offsets).
- EX - executing operation (or calculating address)
- MEM - accessing the memory to be written into
- WB - writing back the result into register.

2) Advantages:

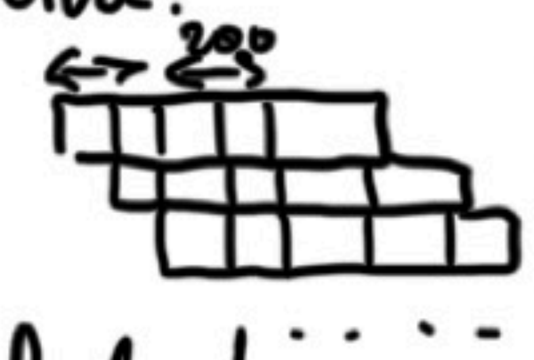
- CPU performance is better then of linear instruction access (one at a time)

Single cycle:



||time| = ~800 sec. $\cdot 10^{-12}$

Pipeline:



| time = ~ 200 sec. $\cdot 10^{-12}$.

- fixed latency:

  time for each instruction is fixed and non increasing.

2) Disadvantages:

- Hazards may occur:
  - structure, (conflict in resource accessing) it's busy | load and store access failed, one at a time
  - data (data has not been processed yet, but is needed for next stage)
  - control (decision on action depends on previous instruction, which is still processing).

  Solutions:
  - stalling (bubbles insertion) nop.
  - adding hardware,
  - forwarding after load/read.
  - preprocessing for branching.

- Complexity for multiple instructions, if compiler has no automatic handler.

N°3 Interrupts:

- interrupts happen in an external controller, which <u>arise</u>, when faced an unexpected (sequence) of events, requiring change in control flow.
- Interrupts may be raised artificially by user, when he needs to manually influence the control flow, or by compiler or an external devicing controller, ex: timer tool, which interrupts if a certain time for sequence of processes have passed (int. may be exceptions)
- Interrupt handler: is called when specific condition is met: an interrupt arises: uepc (pc) is saved and program proceeds to handler, the sequence of instructions for handling with an interrupt is called |⇒ interrupt is processed, handler returns: uret to uepc, skipping the [illegible] (if given) and proceeding further, unless faced another interrupt.

  Registers: (uepc (return interrupt pc), timeh, utvec, uie, ustatus (for errors), accessed by atomic read/write instr).