TSEA26 Tutorial 1. Micro Architecture and Finite Length

Frans Skarman

November 10, 2021

Tutorial structure

- Some more theory
- Introduction to the Senior processor
- Lab hints
- Exercises

Administrative information

Labs:

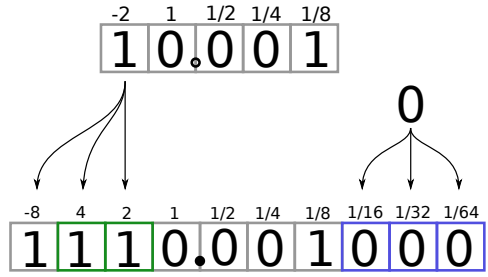
- In groups of 2 students
- No written report
- Demonstration during scheduled lab sessions
- Both of you must be prepared to answer questions about your design
- Mandatory to pass the course (3 of 6 hp)

If you miss a lab:

- Remote work is possible
  - `ssh -YC ssh.edu.liu.se`
  - thinlinc

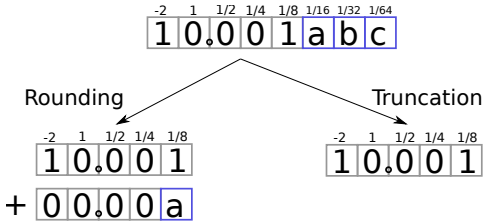
Extending 2's complement numbers

- More integer bits: sign extension
- More fractional bits: adding zeros



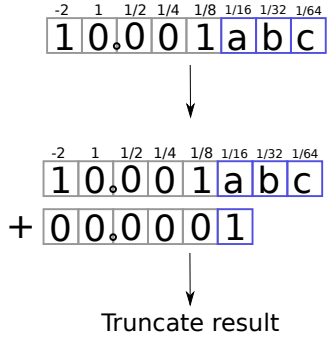
2's complement rounding and truncation

- Truncation just throws away bits
- Rounding introduces less error but requires an extra adder (usually)



2's complement rounding and truncation

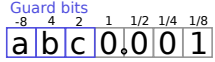
- Another alternative rounding approach (used in the labs)



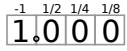
2's complement saturation

Bits to cut off are guard bits

- Used to protect against overflow
- Check sign bit
  - Positive overflow if positive and guard bits are not all 0
  - Negative overflow if negative and guard bits are not all 1
  - Fits otherwise

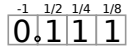


a=1, bc!=11



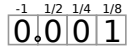
min

a=0, bc!=00



max

otherwise



Adders (signed/unsigned)

16-bit addition with carry in

Inferred:

```
// Compute A+B+c_i. Split output into 16 bit output and carry out
{c_o, result} <= A + B + {15'b0, c_i}
```

Explicit

```
// Add c_i on the left hand side
// Drop left hand side of result, split rest
// into 16 bit output and carry out
{c_o, result, x} <= {A, c_i} + {B, c_i}
// Alternatively
{c_o, result, x} <= {A, 1'b1} + {B, c_i}
```

VHDL behaves slightly differently

Multipliers

- Multiplication of N-bit number with K-bit number produces $N + K$-bit result
- Output fixed point is shifted similarly

Examples:

Integer multiplication:
$0111 \times 0111 = 00110001$ $(7 \times 7 = 35)$

Fixed point multiplication:
$0.11 \times 0.11 = 00.1001$
$(0.75 \times 0.75 = 0.5625)$

Senior

DSP with lots of bells and whistles

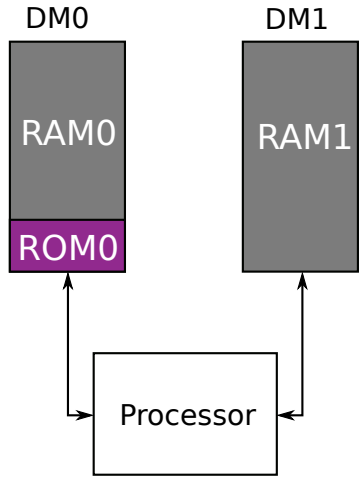
- 32 16-bit general purpose regs (`r0..r31`)
- 32 16-bit special purpose regs (`sr0..sr31`)
- 4 32-bit accumulator regs with 8 guard bits (`acr0..acr3`)

Special purpose registers in Senior

Mnemonic	Location	Address code	Specification
ar0	AG	00000	Address register 0
ar1	AG	00001	Address register 1
ar2	AG	00010	Address register 2
ar3	AG	00011	Address register 3
sp	AG	00100	Stack pointer
bot0	AG	00101	Bottom for AR0
top0	AG	00110	Top for AR0
step0	AG	00111	Step size for AR0
bot1	AG	01000	Bottom for AR1
top1	AG	01001	Top for AR1
step1	AG	01010	Step size for AR1
bitrev	AG	01011	Number of bits to reverse-6
fftbase	AG	01100	Base address for FFT addressing
fftstage	AG	01101	Current stage of FFT addressing
intaddr	AG	01110	Start address for interrupts
fl0	CP	01111	Flags, processor status register
fl1	CP	10000	Flags, core control register
loopn	CP	10001	Number of iterations in loop
loopb	CP	10010	Loop start address
loope	CP	10011	Loop end address
intmask	CP	10100	*Reserved(interrupt mask)*
guards01	MAC	10101	Guard for ACR0 and ACR1
guards23	MAC	10110	Guard for ACR2 and ACR3

Memory in Senior

- Senior has 4 memories
  - RAM0
  - ROM0
  - RAM1
  - Program Memory
- Accessed using `ld` `st`, and special instructions
- RAM0 and ROM0 share the same bus and address space
- RAM1 is independent



Move load and store instructions

Instructions for setting values of registers and memory:[1]

- Copy values from register to register: `move`
- Load from memory 0 or 1: `ld`X
- Store in memory 0 or 1: `st`X
- Clear register: `clear`
- Set register to constant: `set`
- And a few other, see section 2 of manual for details

Check the delay between instructions and insert NOPs or re-order where needed

[1] See manual for more details

Conditional execution

Some instructions support being executed conditionally

```
; Compare r16 with r17
cmp r16,r17
; If they were equal, copy r1 to r0
move.eq r0,r1
    ^^^ Conditional
```

Not all instructions support this, look for `[.cdt]` in the manual

Short and long arithmetic

Short arithmetic instructions operate on normal registers (`r0-r31`)

- `add`
- `sub`
- `mul`
- `...`

Long arithmetic instructions operate the accumulator registers (`acr0..acr4`)

- `addl`
- `subl`
- `mac`
- `...`

Delay slots

Pipelining means jumps take extra cycles. Normally the hardware inserts NOPs while waiting

```
start:
    addi r0,1 ; No effect on jump
    cmp r16,0
    jump.eq some_label
    ;
    nop  ; Inserted by hardware
    nop  ; And always executed
    nop  ; regardless
    ;      of jump taken or not
    addi r0,1
some_label:
    ; ...
```

This is quite wasteful... Delay slots solve this

Delay slots

Pipelining means jumps take extra cycles. Normally the hardware inserts NOPs while waiting

```
start:
    addi r0,1 ; No effect on jump
    cmp r16,0
    jump.eq some_label
    ;
    nop   ; Inserted by hardware
    nop   ; And always executed
    nop   ; regardless
    ;       of jump taken or not
    addi r0,1
some_label:
    ; ...
```

This is quite wasteful... Delay slots solve this

```
start:
    cmp r16,0
    jump.eq ds1 some_label
    addi r0,1 ; Moved from before jump
    ;
    nop   ; Only 2 nops
    nop   ; inserted
    ;
    addi r0,1
some_label:
    ; ...
```

$0 - 3$ delay slots available

The repeat instruction

Tight loops have lots of overhead

```
start:
    set r0,32 ; Init loop
loop:
    ld0  r16,ar0 ; 1 useful insn
    subi r0,1    ; Decrement loop count
    jump.ne loop ; Jump (3 cycles)
    ; do something with r16
```

DSP $\Rightarrow$ accelerate slow instructions

The repeat instruction

Tight loops have lots of overhead

```
start:
    set r0,32 ; Init loop
loop:
    ld0  r16,ar0 ; 1 useful insn
    subi r0,1    ; Decrement loop count
    jump.ne loop ; Jump (3 cycles)
    ; do something with r16
```

DSP $\Rightarrow$ accelerate slow instructions

`repeat`: repeat a loop a fixed amount of times

```
start:
    repeat 32,loop_end
    ld0  r16,ar0
loop_end:
    ; do something with r16
```

The repeat instruction

- Saves $4n + 1$ cycles in this case
- Uses 3 special registers `loopn`, `loopb` and `loope`
- Nesting not possible

Convolution instruction (convXX)

Perform a convolution step in a single instruction

```
convss acr1, (ar0++), (ar1++%)
                       ^^^^^^^ Data 2 address
                 ^^^^^ Data1 address
          ^^^^ Target accumulator
```

- Signed or unsigned convss, convsu, convus, convuu
- Retreive data at arX
- Multiply values
- Add to accumulator register

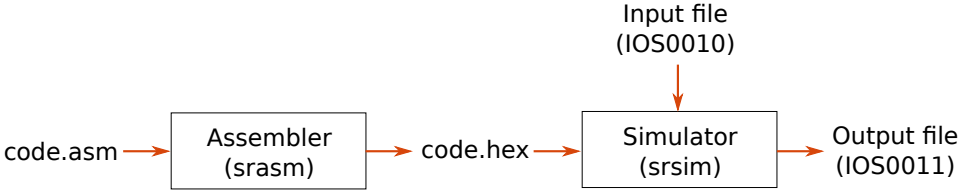
Addressing modes

Retreive data and update address register in the same instruction

- `arX` No change
- `arX++` Load data, then increment by `stepX`
- `arX++%` Load data, then increment with modulo
  - if `arX` $\geq$ `topX` then `arX` = `botX`
  - Useful for ring buffers

Senior assembler and simulator

- Assembly code includes
  - Assembly instructions: `ld, add, cmp ...`
  - Symbolic names for memory locations (labels)
  - Assembler directives `.skip 31`, `.df 0.125`
- Senior assembler (`srasm`): translates assembly code into executable binary code (hex files)
- Senior simulator (`srsim`): executes hex files and provides a debugging environment



The in and out instructions

Used by simulator for input/output

```
; Output value of r0 to new line in IOS0011
out 0x11,r0

; Tell simulator to end simulation
out 0x13, r0 ; Dummy register needed here

; Read a line of IOS0010 to r0
in r0, 0x10
```

Using srsim

```
# Assemble code into code.hex
srasm code.asm
# Start simulator in debug mode
srsim code.hex
```

- `h`: help menu
- `r<n>`: run n instructions and break
- `l`: list instructions around program counter
- `p`: print content of registers
- `g`: run the whole program to completion

Start simulator and run program with `srsim -r <hex file>`

Exercises

Exercises 1, 2, and 3 from Introductory exercises in TSEA26 exercise collection.