CSE 141L
Spring, 2017

ISA DESIGN

April 24, 2017

Author: Chenxu Jiang
PID: A92114155
Author: Dingcheng Hu
PID: A92090168

1 Introduction

Our achitecture name is Hexac. Instead of single accumulator, which accumulator achitecture usually use, we use six accumulators to save the step of load to or store from the accumulator. Using six accumulators means accumulator instructions are six times of the normal accumulator instructions. We need to compensate the need for the large number of instructions by using only 4 accessible temporal registers. Small number of accessible registers saves us a lot of space for opcode and allows us to invent very specific and bizarre operations for the specified 3 tasks. Dynamic clock counts are reduced by six accumulators which save step for load/store and very specific operations for assigned tasks.

2 Instruction Format

Set Type Set 8-bit immediate constant to `$t0`

0	8 bit immediate

ex. `set 255`

0	1 1 1 1 1 1 1 1 1

1-Arg Reg Operation that takes 2-bit register as argument

1	opcode	reg

ex. `ldm0 $t0 ! load memory at $t0 to $acc0`

1	0 0 0 0 0 0	0 0

Branch Read relative jump address store at 2-bit lookup table

1	opcode	LT

ex. `bnzr loop ! Jump to loop if $t0 not 0`

1	0 1 1 1 1 0	0 0

Miscellaneous Weird specified operations that takes no arg

1	opcode	00

ex. `atos_2 ! swap add and sub if $acc2 negative`

1	1 0 1 1 0 0	00

3 Operations

Operation Table

Operation		Description	Format	Opcode
set	[imm]	Set 8-bit unsigned immediate value to $t0	Set	None
ldm0	[reg]	load word in memory location [reg] to $acc0	1-Arg	0x00
ldm1	[reg]	load word in memory location [reg] to $acc1	1-Arg	0x01
ldm2	[reg]	load word in memory location [reg] to $acc2	1-Arg	0x02
ldm3	[reg]	load word in memory location [reg] to $acc3	1-Arg	0x03
ldm4	[reg]	load word in memory location [reg] to $acc4	1-Arg	0x04
ldr1	[reg]	load word in [reg] to $acc1	1-Arg	0x05
ldr4	[reg]	load word in [reg] to $acc4	1-Arg	0x06
ldr5	[reg]	load word in [reg] to $acc5	1-Arg	0x07
add1	[reg]	add word in [reg] to $acc1, set carry to $flag	1-Arg	0x08
add4	[reg]	add word in [reg] to $acc4, set carry to $flag	1-Arg	0x09
add5	[reg]	add word in [reg] to $acc5, set carry to $flag	1-Arg	0x0a
addf0	[reg]	add word in [reg] to $acc0, $flag as init carry	1-Arg	0x0b
addf4	[reg]	add word in [reg] to $acc4, $flag as init carry	1-Arg	0x0c
sub1	[reg]	sub word in [reg] from $acc1, set carry to $flag	1-Arg	0x0d
sub3	[reg]	sub word in [reg] from $acc3, set carry to $flag	1-Arg	0x0e
sub4	[reg]	sub word in [reg] from $acc4, set carry to $flag	1-Arg	0x0f
sub5	[reg]	sub word in [reg] from $acc5, set carry to $flag	1-Arg	0x10
subf2	[reg]	sub word in [reg] from $acc2, $flag as init carry	1-Arg	0x11

Operation Table Continued

Operation		Description	Format	Opcode
str1	[reg]	store word in $acc1 to [reg]	1-Arg	0x12
str2	[reg]	store word in $acc2 to [reg]	1-Arg	0x13
str3	[reg]	store word in $acc3 to [reg]	1-Arg	0x14
stm0	[reg]	store word in $acc0 to memory location [reg]	1-Arg	0x15
stm1	[reg]	store word in $acc1 to memory location [reg]	1-Arg	0x16
stm3	[reg]	store word in $acc3 to memory location [reg]	1-Arg	0x17
stm4	[reg]	store word in $acc4 to memory location [reg]	1-Arg	0x18
stm5	[reg]	store word in $acc5 to memory location [reg]	1-Arg	0x19
mov	[reg]	copy word in $t0 to [reg]	1-Arg	0x1a
srl	[reg]	logical rshift [reg], set $flag to shift out	1-Arg	0x1b
sra	[reg]	arithmetical rshift [reg], set $flag to shift out	1-Arg	0x1c
srf	[reg]	right shift [reg], set shift in as $flag	1-Arg	0x1d
bnzr	[addr]	branch if $t0 not 0, jump to lookup table [addr]	Branch	0x1e
bnuzr	[addr]	branch if upper 4 bits of $t0 are not 0	Branch	0x1f
b1ne31	[addr]	branch if $acc1 is not equal to 31	Branch	0x20
dec1		decrease $acc1 by 1	Misc.	0x21
zero0		zero $acc0	Misc.	0x22
zero2		zero $acc2	Misc.	0x23
zero3		zero $acc3	Misc.	0x24
zero4		zero $acc4	Misc.	0x25
srl4		logical rshift $acc4, set shift out to $flag	Misc.	0x26
sra5		arithmetical rshift $acc5, set shift out to $flag	Misc.	0x27
srf5		rshift $acc5, set shift in as $flag	Misc.	0x28
sll1		logical lshift $acc1, set shift out to $flag	Misc.	0x29
slf0		lshift $acc0, shift in as $flag, shift out to $flag	Misc.	0x2a
slf2		lshift $acc2, shift in as $flag	Misc.	0x2b
atos_2		swap add and sub commands if $acc2 is negative	Misc.	0x2c
add3_n2		add 1 to $acc3, if $acc2 not negative	Misc.	0x2d
add4_n2		add 1 to $acc4, if $acc2 not negative	Misc.	0x2e
cp0_5		copy $acc0 to $acc5	Misc.	0x2f
add25		add $acc5 from $acc2, set carry to $flag	Misc.	0x30
sub25		sub $acc5 from $acc2, set carry to $flag	Misc.	0x31
subf25		sub $acc5 from $acc2, $flag as init carry	Misc.	0x32
inc_af0		$acc0 += $flag ^ $flip	Misc.	0x33
add0c25		add 1 to $acc0 if upper 4bit of $acc2, $acc5 match	Misc.	0x34
ld4m1		load word at memory location $acc1 to $acc4	Misc.	0x35
halt		signal stop execution	Misc.	0x36

4 Internal Operand

$acc0 - $acc5 Five accumulator registers, which can do addition/subtraction and load from/store to register or memory address in register. Some of the accumulator registers support shifts. $acc5 is the special one that many other accumulator register can directly read its value and do action with special misc. operations. They all can not be use as an argument in **1-Arg** type operations.

$flip One 1-bit register that convert adder/subtracter, set by `atos_2`. If it is set, the `add` operations became `sub` and `sub` became `add`, except inc/decrement for counter or specified.

$flag One 1-bit register that stores shift out and overflow carry bit. Shift and add/sub operation without "f" set the $flag and Shift and add/sub operations with "f" read the $flag.

$t0-$t3 Four accessible common registers, which can be used as argument in **1-Arg** operations. Can do shift and be used to store temporal value. $t0 is special that it can be set to 8-bit immediate value with `set`.

5 Control Flow

Only backward branches for for/while loop are supported. The if branches are saved by `atos_2`, since in assigned programs, the branch is only different on addition and subtraction. Check Operation table for detailed branch operations. The address is calculated by absolute address. Since the branch address is stored in loop-up table with 2 bit control. There is a limitation of one program can only have branches at most to 4 different address. There is no limitation on maximum branch distance.

6 Addressing Mode

Only absolute addressing is supported. The target's absolute address needs to be calculated and stores into temporal registers. Then accumulator registers would load from/store to the memory address stored in temporal registers.

Ex. Read memory location 128:

```
set       128          ! set 128 to $t0
ldm0      $t0          ! load from memory addr in $t0 to $acc0
stm0      $t0          ! store to memory addr in $t0 from $acc0
```

7 Main Memory

Since the largest given memory is 130, we need main memory to have width of 8-bit and depth of 256. Memory is not used a lot. Most operation is done in registers. Only parameter and return result need memory access.

8 Dynamic Instruction Count Optimization

1. We use six accumulator registers instead of one. This saves a lot of loads and stores of temporal values.
2. We use very specific branch condition, that all branches are calculated in one step instead of two. And we have `atos_2` instead of if branch, this have 50% to save a branch always command.
3. We invent a lot of bizarre commands that can only be used in assigned task. The commands may merge 2 commands into 1, without lost of too much genericity.

9 Cycle Time Optimization

1. For every command invented, we verified that the command can be done with at most 3 levels of basic gates, so that no commands are too slow compared to the others.
2. Our design forced us to use an adder/subtractor module, with a controller bit as switch. We are forced to implement a carry-look-ahead adder instead of ripple adder verilog automatically compiled to.
3. We minimize load and store from and to memory, the memory size is as small as possible which increases speed for load and store.

10 Competitor Claim

1. We have 6 accumulator registers and 4 regular accessibile register. And we can ensure that all operations can be done with value stored in registers. Each accumulator registers represents different value. We don't need to load and store different value to the same accumulator registers. Our competitors only have 4 accumulator registers. They need to do load and store to memory for temporal value and reuse the same register for different value. Therefore, in our design, we use much less load and store.
2. We can load 8-bit immediate value to register in one step. Our competiter need to load immediate value with multiple steps Many program would need immediate value. And we save more dynamic counts.
3. We have 6 bits for opcode and can support twices as much operations as our competiters' ISA. With more specific command we can do more with less dynamic counts.

11 Design with 2 More Bits

We would expose accumulator registers. Now accumulator registers cannot be used as arguments. With 2 more bits we are able to do that and we can then have less contrived self-invented instructions. And our ISA can be more generic and supports more programs. Now, we use contrived self-invented instructions for optimization of the only three assigned program.

12 Machine Classification

Hexac is an accumulator machine.

13 Example of Machine Code

Check **Instruction Format**

14 Cordic

14.1 Assembly Code

```
    set     24
    ldm3    $t0              ! load lsw of y to $acc3
    set     16
    ldm2    $t0              ! load msw of y to $acc2
    set     8
    ldm1    $t0              ! load lsw of x to $acc1
    set     0
    ldm0    $t0              ! load msw of x to $acc0
    ldr4    $t0              ! set msw of angle to 0 at $acc4
                             ! lsw of angle set to $acc5 later

                             ! prep loop
    cp0_5                    ! store msw of x to $acc5
    str1    $t1              ! store lsw of x to $t1
    str2    $t2              ! store msw of y to $t2
    str3    $t3              ! store lsw of y to $t3
    set     64               ! set $t0 to 0100 0000          (14)

  loop-msw:
    sra5                     ! shift x msw, set shift-out to $flag
    srf     $t1              ! shift x lsw, set shift-in to $flag

    sra     $t2              ! shift y msw, set shift-out to $flag
    srf     $t3              ! shift y lsw, set shift-in to $flag

    atos_2                   ! set adder to subtracter if $acc2 negative

    add1    $t3
    addf0   $t2              ! x new = x + (y >> 1) or x new = x - (y >> 1)

    sub3    $t1
    subf25                   ! y new = y - (x >> 1) or y new = y + (x >> 1)

    add4    $t0              ! sub if flipped

    cp0_5                    ! store new msw of x to $acc5
    str1    $t1              ! store new lsw of x to $t1
    str2    $t2              ! store new msw of y to $t2
    str3    $t3              ! store new lsw of y to $t3

    srl     $t0              ! shift t0
    bnzr    loop-msw         ! backward                      (16*7)

  end-loop-msw:              ! $t1 now x >> 8, $t3 now y >> 8
    sra     $t2              ! shift sign bit of y to $flag
    srl     $t1              ! x always positive, always shift in 0
    srf     $t3              ! shift in sign bit for the first loop
    mov     $t2              ! set $t0 = 2 to $t2, SOURCE of ZERO
    ldr5    $t0              ! set $acc5 to 0
    set     128              ! $t0 = 1000 0000               (6)
```

```
loop-lsw:
  atos_2                  ! set adder to subtracter if $acc2 negative

  add1    $t3             ! x new = x + (y >> 1) or x new = x - (y >> 1)
  inc_af_0                ! $flip? ($flag? do nothing: add 1) : (add $flag)

  sub3    $t1             ! y new = y - (x >> 1) or y new = y + (x >> 1)
  subf2   $t2             ! sub 0 and flag bit

  add5    $t0
  addf4   $t2             ! add to angle or subtract to angle if flipped

  str1    $t1             ! store lsw of x
  srl     $t1             ! shift logical, x always positive
  str3    $t3             ! store lsw of y
  sra     $t3             ! shift arithmetically, y can be negative

  srl     $t0             ! shift right angle
  bnuzr   loop-lsw        ! loop till $t0 = 0000 1000    (13*4)

end-loop:
  set     5
  stm0    $t0
  set     6
  stm1    $t0
  set     7
  stm4    $t0
  set     8
  stm5    $t0             !                              (8)

  halt                    ! store and stop         Total (193)
```

14.2 Dynamic Instruction Count

The instruction count would never vary, since there is no if branch that have different instruction counts for different branches. The dynamic instruction count for all executions is 193.

15 String Match

15.1 Assembly Code

```
  set     94
  ldr1    $t0           ! load 95 to $acc1, counter
  set     95
  ldm5    $t0           ! load last word to $acc5
  set     9
  ldm2    $t0           ! load word to match
  zero0                 ! zero $acc0, pattern match counter   (7)

loop:
  ld4m1                 ! load word at mem of counter number
  add0c25               ! add 1 to $acc0 if lower 4 bit of $acc2 $acc4 match
  srl4                  ! shift right $acc4
```

```
    srf5                ! shift right $acc5, shift in = shift out of $acc4
    add0c25
    srl4
    srf5
    add0c25
    srl4
    srf5
    add0c25
    srl4
    srf5
    add0c25
    srl4
    srf5
    add0c25
    srl4
    srf5
    add0c25
    srl4
    srf5
    add0c25
    srl4
    srf5                ! shift right $acc4 $acc5 8 times

    dec1                ! decrement counter
    b1ne31  loop        ! branch if counter reaches 31        (28*63)

  end-loop:
    add0c25             ! add 1 to $acc0 if lower 4 bit of $acc2 $acc4 match
    srf5                ! shift right $acc5, shift in = shift out of $acc4
    add0c25
    srf5
    add0c25
    srf5
    add0c25             ! check match extra 4 times,          (7)

    set     10
    stm0    $t0         ! store pattern counter               (2)

    halt                                                Total (1781)
```

15.2 Dynamic Instruction Count

The instruction count would never vary, since there is no if branch that have different instruction counts for different branches. The dynamic instruction count for all executions is 1781.

16 Division

16.1 Assembly Code

```
    set     128
    ldm0    $t0         ! $acc0 msw of dividend
    set     129
    ldm1    $t0         ! $acc1 lsw of dividend
    set     130
```

```
    ldm5    $t0             ! $acc5 divisor
    zero2                   ! zero $acc2 reminder                   (7)

    sub25                   ! $acc2 -= divisor, must negative
    sll1
    slf0
    slf2                    ! (reminder, msw Dividend, lsw Dividend) << 1
    add25                   ! $acc2 += divisor, since last op make $acc2 negative
    set     64              ! set $t0 = 0100 0000
    add3_n2 $t0             ! if $acc2 non negative and $t0 to msw of quotient
    set     32              ! set $t0 = 0010 0000                   (8)

  loop-msw:
    atos2                   ! transit adder to subtracter if remainder negative
    sll1
    slf0
    slf2                    ! (reminder, msw Dividend, lsw Dividend) << 1
    sub25                   ! if neg, $acc2 += divisor, if pos $acc2 -= divisor
    add3_n2 $t0             ! if $acc2 >=0 add $t0 to msw quotient, despite flipped

    srl     $t0             ! shift $t0
    bnzr    loop-msw        !                                       (8*6)

  end-loop-msw:
    set     128             ! set $t0 = 1000 0000                   (1)

  loop-lsw:
    atos2                   ! transit adder to subtracter if remainder negative
    sll1
    slf0
    slf2                    ! (reminder, msw Dividend, lsw Dividend) << 1
    sub25                   ! if neg, $acc2 += divisor, if pos $acc2 -= divisor
    add4_n2 $t0             ! if $acc2 >=0 and $t0 to lsw quotient, despite flipped

    srl     $t0             ! shift $t0
    bnzr    loop-lsw        !                                       (8*8)

  end-loop-msw:
    set     126
    stm3    $t0             ! store msw of quotient
    set     127
    stm4    $t0             ! lsw of quotient                       (4)

    halt                                                  Total (133)
```

16.2 Dynamic Instruction Count

The instruction count would never vary, since there is no if branch that have different instruction counts for different branches. The dynamic instruction count for all executions is 133.