CSE 5350 Computer Architecture II © DL, UTA, 2000's- 2022

The CAT computer series (Computer Architecture Two) is a very simple, somewhat functional assembler and simulator-VM for the CAT ISAs. They are currently implemented in Python 2.7 to be portable to a large number of platforms. While the reference implementations have no intentional implementation defects, they have not been well tested, and probably have such types of defects; on the other hand they deliberately have functional problems: some models have unimplemented or partially implemented instructions, simulator and VM execution and algorithm and trace portions are incomplete.

Didn't need to or wanted to finish implementing Store instruction, can handle more registers easily, among other features, has memory management registers (!), interrupts, and procedure/function calls.

ISA: Instructions:

```
ld    reg  addr        # load reg with contents of mem at this addr
ldi   reg  imval       # load imval part of instruction into reg
add reg reg/*reg       # add second reg to first, second reg can be a pointer to mem value
inc  reg               # reg value +1
dec reg
bnz reg addr           # if reg contents are zero go to address
hlt                    # stop execution
st    reg addr         # store reg to mem addr
dw  val                # define word – initialize memory contents
sub reg reg/*reg       # sub second reg to first, second reg can be a pointer to mem value
brl  reg addr          # store ret addr in reg  go to address
ret  reg               # go back to here – register has addr
int   reg addr         # store ret addr in reg, go to addr
traps illegal instructions, arithmetic overflow, has system calls
end                    # end of assembly unit
```

(This has been modified in the code – trust the code, not documentation!)
Everything is a 24 bit word (??): registers, data, instructions.
8 general purpose regs, some more than others (R7 and R6 are memory segments, R5 sometimes branch/link return address)
Many words of memory (dozens of KWords)
5 bit opcode, 4 bit reg field, 14 bit address-offset/immediate

CAT1.1

- Implement store
- Calculate CPI, MIPS (IPS), "clock" speed, memory references, IC, benchmarks
- Add Benchmark for:

```
For i = 1 to N
   Sum += A[i] – B[i]
```

- Some questions:

What are the numbers on a line, are they absolute addresses or relative?
How does memory "management" work? Can Instructions and data share the same space? Do they?
Can DWs be mixed within an instruction stream? What happens?
What is an "overflow"? What happens? When does it occur – example?
Is "add *r3 *r4" legal? What happens?
What does "go" do?
How would you implement a "bz" (branch if reg is zero) instruction?
How would you implement a "bzl" (branch and link if reg is zero) instruction?
(why is this somewhat difficult)?

Can you add stack instructions: Push, Add?

Benchmark program; sum an array

```
 go     100
100     ld   2 200        ; r2 has value of counter
        ldi  3 201        ; r3 points to first value
        ldi  1 0          ; r1 contains sum
103     add  1 *3         ; r1 = r1 + next array value
        inc  3
        dec  2
        bnz  2 3
        sys  1 16
16      dw   0
200     dw   5
        dw   3
        dw   2
        dw   0
        dw   8
        dw   100
        end
```

```
; sum an array
       go   0
0      ld   2 .count     ; r2 has value of counter
       ldi  3 .vals     ; r3 points to first value
       ldi  1 0       ; r1 contains sum
.loop  add  1 *3      ; r1 = r1 + next array value
       inc  3
       dec  2
       bnz  2 .loop
       sys  1 16
       dw   0
.count dw   5
.vals  dw   3
       dw   2
       dw   0
       dw   8
       dw   100
16     dw   0
       end
```

a.out:

```
go 0
0x1c8009    0
0x24c00a    1
0x244000    2
0x46c00    3
0x10c000    4
0xc8000    5
0x308003    6
0x404010    7
0x0    8
0x5    9
0x3    10
0x2    11
0x0    12
0x8    13
0x64    14
0x0    15
0x0    16
0x0    17
0x0    18
0x0    19    …
```

Old Versions, for reference:

```
#! python
# (c) DL, UTA, 2009(?) - 2017
import  sys, string
wordsize = 24                                       # everything is a word
numregbits = 3                                      # actually +1, msb is indirect bit
opcodesize = 5
memloadsize = 1024                                  # change this for larger programs
numregs = 2**numregbits
opcposition = wordsize - (opcodesize + 1)           # shift value to position opcode
reg1position = opcposition - (numregbits + 1)       # first register position
reg2position = reg1position - (numregbits + 1)
memaddrimmedposition = reg2position                 # mem address or immediate same place as reg2
startexecptr = 0;
def regval ( rstr ):                                # help with reg or indirect addressing
    if rstr.isdigit():
       return ( int( rstr ) )
    elif rstr[0] == '*':
       return ( int ( rstr[1:] ) + (1<<numregbits) )
    else:
       return 0                                     # should not happen
mem = [0] * memloadsize                             # this is the memory load executable
# instruction mnemonic, type: (1 reg, 2 reg, reg+addr, immed, pseudoop), opcode
opcodes = {'add': (2, 1),'sub': (2, 2),              # ie, "add" is a type 2 instruction, opcode = 1
           'dec': ( 1, 3), 'inc': ( 1, 4 ),
           'ld': (3, 7), 'st': (3, 8), 'ldi': (3, 9),
           'bnz': (3, 12), 'brl': (3, 13),
           'ret': ( 1, 14 ),
           'int': (3, 16), 'sys': (3, 16),           # syscalls are same as interrupts
           'dw': (4, 0), 'go':(3, 0), 'end': (0, 0) }  # pseudo ops
curaddr = 0                                          # start assembling to location 0
#for line in open(sys.argv[1], 'r').readlines():     # command line
infile = open("in.asm", 'r')
# Build Symbol Table
symboltable = {}
for line in infile.readlines():           # read our asm code
   tokens = string.split( string.lower( line ))      # tokens on each line
   firsttoken = tokens[0]
   if firsttoken.isdigit():                          # if line starts with an address
       curaddr = int( tokens[0] )                   # assemble to here
       tokens = tokens[1:]
   if firsttoken == ';':                             # skip comments
      continue
   if firsttoken == 'go':                            # start execution here
      startexecptr = ( int( tokens[ 1 ] ) & ((2**wordsize)-1))  # data
      continue
   if firsttoken[0] == '.':
      symboltable[firsttoken] = curaddr
   curaddr = curaddr + 1
print symboltable
infile.close()
infile = open("in.asm", 'r')
for line in infile.readlines():           # read our asm code
   tokens = string.split( string.lower( line ))      # tokens on each line
   firsttoken = tokens[0]
   if firsttoken.isdigit():                          # if line starts with an address
       curaddr = int( tokens[0] )                   # assemble to here
       tokens = tokens[1:]
   if firsttoken == ';':                             # skip comments
      continue
   if firsttoken == 'go':                            # start execution here
      startexecptr = ( int( tokens[ 1 ] ) & ((2**wordsize)-1))  # data
      continue
   if firsttoken[0] == '.':
      symaddr = symboltable[firsttoken]
      tokens = tokens[1:]
   memdata = 0                                          # build instruction step by step
   instype = opcodes[ tokens[0] ] [0]
   memdata = ( opcodes[ tokens[0] ] [1] ) << opcposition   # put in opcode
   if instype == 4:                                     # dw type
      memdata = ( int( tokens[ 1 ] ) & ((2**wordsize)-1))  # data is wordsize long
   elif instype == 0:                                   # end type
```

```
      memdata = memdata
   elif instype == 1:                                      # dec,inc type, one reg
      memdata = memdata + (regval( tokens[1] ) << reg1position)
   elif instype == 2:                                      # add, sub type, two regs
      memdata = memdata + ( regval( tokens[1] ) << reg1position ) + ( regval( tokens[2] ) << reg2position)
   elif instype == 3:                                      # ld,st type
      token2 = tokens[2]
      if token2.isdigit():
         memaddr = int( tokens[2] )
      else:
         memaddr = symboltable[ token2 ]
      memdata = memdata + ( regval( tokens[1] ) << reg1position ) + memaddr
   mem[ curaddr ] = memdata                                # memory image at the current location
   curaddr = curaddr + 1
outfile = open("a.out", 'w')                               # done, write it out
outfile.write( 'go ' + '%d' % startexecptr )               # start execution here
outfile.write( "\n" )
for i in range(memloadsize):                               # write memory image
   outfile.write( hex( mem[ i ] ) + "    " + '%d'%i)
   outfile.write( "\n" )
outfile.close()
```

```
#! python
# (c) DL, UTA, 2009(?) - 2017
import  sys, string, time
wordsize = 24                                      # everything is a word
numregbits = 3                                     # actually +1, msb is indirect bit
opcodesize = 5
addrsize = wordsize - (opcodesize+numregbits+1)    # num bits in address
memloadsize = 1024                                 # change this for larger programs
numregs = 2**numregbits
regmask = (numregs*2)-1                            # including indirect bit
addmask = (2**(wordsize - addrsize)) -1
nummask = (2**(wordsize))-1
opcposition = wordsize - (opcodesize + 1)          # shift value to position opcode
reg1position = opcposition - (numregbits +1)          # first register position
reg2position = reg1position - (numregbits +1)
memaddrimmedposition = reg2position                # mem address or immediate same place as reg2
realmemsize = memloadsize * 1                      # this is memory size, should be (much) bigger than a
program
#memory management regs
codeseg = numregs - 1                              # last reg is a code segment pointer
dataseg = numregs - 2                              # next to last reg is a data segment pointer
#ints and traps
trapreglink = numregs - 3                          # store return value here
trapval     = numregs - 4                          # pass which trap/int
mem = [0] * realmemsize                            # this is memory, init to 0
reg = [0] * numregs                                # registers
clock = 1                                          # clock starts ticking
ic = 0                                             # instruction count
numcoderefs = 0                                    # number of times instructions read
numdatarefs = 0                                    # number of times data read
starttime = time.time()
curtime = starttime
def startexechere ( p ):
    # start execution at this address
    reg[ codeseg ] = p
def loadmem():                                     # get binary load image
  curaddr = 0
  for line in open("a.out", 'r').readlines():
    token = string.split( string.lower( line ))      # first token on each line is mem word, ignore rest
    if ( token[ 0 ] == 'go' ):
        startexechere(  int( token[ 1 ] ) )
    else:
        mem[ curaddr ] = int( token[ 0 ], 0 )
        curaddr = curaddr = curaddr + 1
def getcodemem ( a ):
    # get code memory at this address
    memval = mem[ a + reg[ codeseg ] ]
    return ( memval )
def getdatamem ( a ):
```

```
    # get code memory at this address
    memval = mem[ a + reg[ dataseg ] ]
    return ( memval )
def getregval ( r ):
    # get reg or indirect value
    if ( (r & (1<<numregbits)) == 0 ):               # not indirect
       rval = reg[ r ]
    else:
       rval = getdatamem( reg[ r - numregs ] )       # indirect data with mem address
    return ( rval )
def checkres( v1, v2, res):
    v1sign = ( v1 >> (wordsize - 1) ) & 1
    v2sign = ( v2 >> (wordsize - 1) ) & 1
    ressign = ( res >> (wordsize - 1) ) & 1
    if ( ( v1sign ) & ( v2sign ) & ( not ressign ) ):
      return ( 1 )
    elif ( ( not v1sign ) & ( not v2sign ) & ( ressign ) ):
      return ( 1 )
    else:
      return( 0 )
def dumpstate ( d ):
    if ( d == 1 ):
        print reg
    elif ( d == 2 ):
        print mem
    elif ( d == 3 ):
        print 'clock=', clock, 'IC=', ic, 'Coderefs=', numcoderefs,'Datarefs=', numdatarefs, 'Start Time=',
starttime, 'Currently=', time.time()
def trap ( t ):
    # unusual cases
    # trap 0 illegal instruction
    # trap 1 arithmetic overflow
    # trap 2 sys call
    # trap 3+ user
    rl = trapreglink                            # store return value here
    rv = trapval
    if ( ( t == 0 ) | ( t == 1 ) ):
       dumpstate( 1 )
       dumpstate( 2 )
       dumpstate( 3 )
    elif ( t == 2 ):                          # sys call, reg trapval has a parameter
       what = reg[ trapval ]
       if ( what == 1 ):
           a = a        #elapsed time
    return ( -1, -1 )
    return ( rv, rl )
# opcode type (1 reg, 2 reg, reg+addr, immed), mnemonic
opcodes = { 1: (2, 'add'), 2: ( 2, 'sub'),
            3: (1, 'dec'), 4: ( 1, 'inc' ),
            7: (3, 'ld'),  8: (3, 'st'), 9: (3, 'ldi'),
           12: (3, 'bnz'), 13: (3, 'brl'),
           14: (1, 'ret'),
           16: (3, 'int') }
startexechere( 0 )                                  # start execution here if no "go"
loadmem()                                           # load binary executable
ip = 0                                              # start execution at codeseg location 0
# while instruction is not halt
while( 1 ):
   ir = getcodemem( ip )                            # - fetch
   ip = ip + 1
   opcode = ir >> opcposition                       # - decode
   reg1   = (ir >> reg1position) & regmask
   reg2   = (ir >> reg2position) & regmask
   addr   = (ir) & addmask
   ic = ic + 1
                                                    # - operand fetch
   if not (opcodes.has_key( opcode )):
      tval, treg = trap(0)
      if (tval == -1):                              # illegal instruction
         break
   memdata = 0                                      #     contents of memory for loads
   if opcodes[ opcode ] [0] == 1:                   #     dec, inc, ret type
      operand1 = getregval( reg1 )                  #       fetch operands
```

```
   elif opcodes[ opcode ] [0] == 2:                 #     add, sub type
      operand1 = getregval( reg1 )                  #       fetch operands
      operand2 = getregval( reg2 )
   elif opcodes[ opcode ] [0] == 3:                 #     ld, st, br type
      operand1 = getregval( reg1 )                  #       fetch operands
      operand2 = addr
   elif opcodes[ opcode ] [0] == 0:                 #     ? type
      break
   if (opcode == 7):                                # get data memory for loads
      memdata = getdatamem( operand2 )
   # execute
   if opcode == 1:                     # add
      result = (operand1 + operand2) & nummask
      if ( checkres( operand1, operand2, result )):
         tval, treg = trap(1)
         if (tval == -1):                           # overflow
            break
   elif opcode == 2:                   # sub
      result = (operand1 - operand2) & nummask
      if ( checkres( operand1, operand2, result )):
         tval, treg = trap(1)
         if (tval == -1):                           # overflow
            break
   elif opcode == 3:                   # dec
      result = operand1 - 1
   elif opcode == 4:                   # inc
      result = operand1 + 1
   elif opcode == 7:                   # load
      result = memdata
   elif opcode == 9:                   # load immediate
      result = operand2
   elif opcode == 12:                  # conditional branch
      result = operand1
      if result <> 0:
         ip = operand2
   elif opcode == 13:                  # branch and link
      result = ip
      ip = operand2
   elif opcode == 14:                   # return
      ip = operand1
   elif opcode == 16:                   # interrupt/sys call
      result = ip
      tval, treg = trap(reg1)
      if (tval == -1):
        break
      reg1 = treg
      ip = operand2
   # write back
   if ( (opcode == 1) | (opcode == 2 ) |
         (opcode == 3) | (opcode == 4 ) ):     # arithmetic
        reg[ reg1 ] = result
   elif ( (opcode == 7) | (opcode == 9 )):     # loads
        reg[ reg1 ] = result
   elif (opcode == 13):                        # store return address
        reg[ reg1 ] = result
   elif (opcode == 16):                        # store return address
        reg[ reg1 ] = result
   # end of instruction loop
# end of execution
```