CISC360 - Project 2

James Feister - jfeister@udel.edu

Introduction

Project 2 required we expand upon our current Gemini implementations.

- An assembler to convert gemini asm to bytecode.
- Loading and Parsing of gemini bytecode in the simulator
- BGE, BLE, BNE, HLT instructions
- Implementation of Direct and two way cache scheme
- Extra credit
  - Implement a SETHI and SETLO functionality for 32 bit numbers
  - Implement a JMP and RET functionality, for functions and recursion
  - Overflow detection on Mul and Div instructions
  - Extend cache to perform 4 memory block grabs with each fetch

Cache

The Gemini Simulator now supports cache. A user is allowed to select from 4 different cache schemes. Direct mapping one block, Direct mapping four block, Two way mapping one block, and Two way mapping four block. The current gui supports only a drop down selection of these scheme types but more may be created. The cache system was implemented to allow for easy modification of: Cache size, Cache set size (1 for direct mapping, 2 for two way mapping), Block size (number of blocks to pull from memory at a time). The test programs benefited greatly from the 4 block size locality bringing the number of requests from memory to only 4 for the initial values. Otherwise all their memory requests were fed from the cache.

Project 2 test1.s benefitted the most from the direct and two way four block implementations, by decreasing their main memory access to only 4 and the cache handled the rest of the requests. The one block for direct and two way only cut the main memory requests down to half.

Project 2 test2.s had the same results as Project 2 test1.s but had issues do to the random replacement on the cache. It may be better to have a smarter last used implementation so that cache slots to be replaced for a specific set will be more predictable.

Test_1.s demonstrated physical locality advantages with direct one block with 8 contiguous items, and two way four block with 4 memory block contiguous items per cache line. The two way set one block broke up the physical locality enough to cause more cache misses.

ByteCode

GeminiAssembler.pro is the project file for the Gemini Assembler (GASM). GASM will read Gemini assembly files, verify their content, and then output a bytecode version of the files. The Gemini Simulator has been modified to input the bytecode files and run them.

Layout of the bytecode line is [8 bit operand][8 bit value | memory access flag][16 bit value].

Extra Instructions JMP RET,BGE, BNE, BLE HLT

JMP and RET instructions with a 25 deep stack has been implemented. If the stack is overflowed or underflowed an exception will throw and the simulation will terminate. Program ‘**recursion.s**’ will demonstrate the usage of these instructions. BGE, BNE, BLE, and HLT were extra requirements of this project.

OverFlow Detection

Overflow detection of the 16-bit multiplication and division instructions has been implemented. Should this occur the OVF bit will be set to 1. Program ‘**overflow.s**’ will demonstrate this functionallity.

SETHI SETLO

The Gemini now supports two 32 bit registers accessible through SETHI #$[0|1] and SETLO#$[0|1] instructions. If a program specified the incorrect register an error will throw and the simulation will terminate. The added instructions to support these registers are:

- SETHI #$ [0 | 1]: Set hi order bits of either SL0 or SL1 with value from Accumulator
- SETLO #$ [0 | 1]: Set low order bits of either SL0 or SL1 with value from Accumulator
- LDHI #$[0 | 1]: Load accumulator with hi order bits of either SL0 or SL1
- LDLO #$[0 | 1]: Load accumulator with low order bits of either SL0 or SL1
- ADDSL : Add SL0 and SL1 puts result in SL0
- SUBSL : Subtract SL0 from SL1 puts result in SL0
- MULSL : Multiply SL0 by SL1 puts result in SL0
- DIVSL : Divide SL0 by SL1 puts result in SL0

Program ‘**sethi_sethl.s**’ will demonstrate these instructions

Test 1.c Conversion to Test 1.s

We were tasked to convert the following c code to gemini assembly.

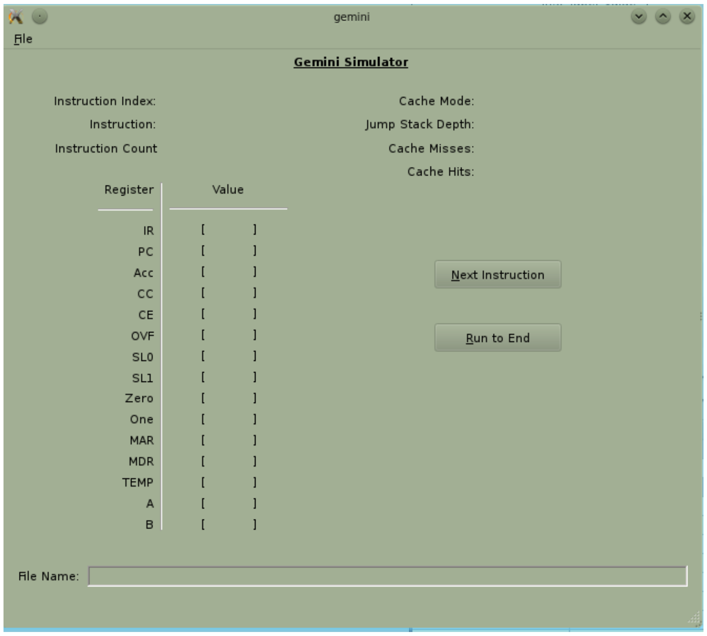
<table>
<tr>
<td>
<pre>
int main(void){
  int x = 0;
  int g = 0x0, h = 0x7, j = 0xa, k = 0xf, l = 0xf;
  int y = 10;
  for(int i = 0; i < y; i++){
    x += y * i;
    g = h & j & i | k;
    g = !g;
  } if(x & 0xa > y){
    x = 0;
  }
}
</pre>
</td>
<td>
<pre>
main:
  lda #$0 ! x
  sta $0
  lda #$0 ! g
  sta $1
  lda #$7 ! h
  sta $2
  lda #$10 ! j
  sta $3
  lda #$15 ! k
  sta $4
  lda #$15 ! l
  sta $5
  lda #$10 ! y
  sta $6
  lda #$0 ! i
  sta $7
for:
  lda $7
  sub $6
  be forend
  lda $7 ! i
  mul $6 ! * y
  add $0 ! += x
  sta $0 ! x = acc
  lda $2 ! h
  and $3 ! & j
  and $7 ! & i
  or $4 ! | k
  sta $1
  nota
  sta $1
  lda $7 ! load i
  add #$1
  sta $7
  ba for
forend:
  lda $0
  and #$10
  sub $6
  bl end
  lda #$0
  sta $0
end:
</pre>
</td>
</tr>
</table>

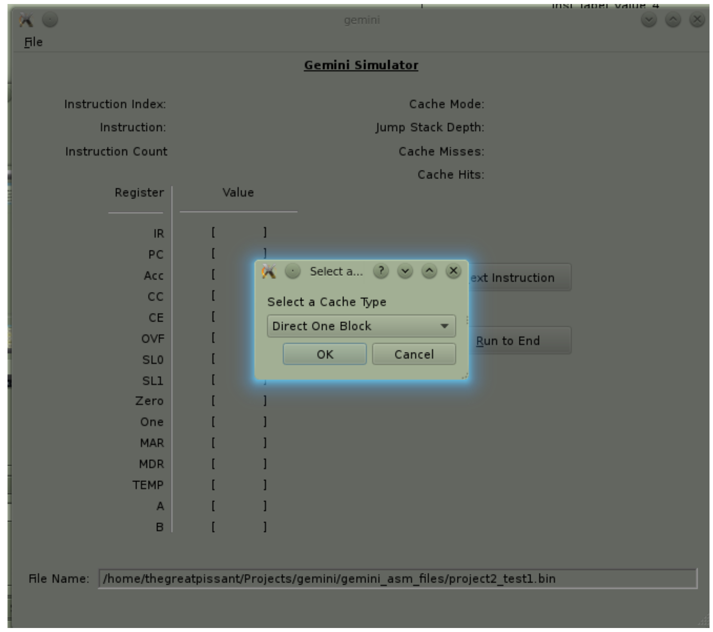
Gemini Assembler (GASM)

Gemini Assembler on startup

Gemini Assembler Finished Converting

Gemini simulator update GUI





Updates include:

- cache mode
- jump stack
- overflow
- sl0 and sl1 registers
- current filename loaded
- “Run to End” button to execute all of the program instructions.
- Cache selection dialog that appears after opening the gemini application.
- Default run with the ‘Direct One Block’ cache mode.
- User selectable cache mode for: ‘Direct One Block’, ‘Direct Four Block’, ‘Two way One Block’, and ‘Two Way Four Block’.

project2_test1.s

Gemini Assembly

```
main:
 ! Push an array onto the stack (from Stack[0] to Stack[15]
 ! Stack[0] = 10
 ! ...
 ! Stack[15] = 25
 lda #$10
 sta $0
 lda #$11
 sta $1
 lda #$12
 sta $2
 lda #$13
 sta $3
 lda #$14
 sta $4
 lda #$15
 sta $5
 lda #$16
 sta $6
 lda #$17
 sta $7
 lda #$18
 sta $8
 lda #$19
 sta $9
 lda #$20
 sta $10
 lda #$21
 sta $11
 lda #$22
 sta $12
 lda #$23
 sta $13
 lda #$24
 sta $14
 lda #$25
 sta $15
 ! Cache filled

 ! Program demonstrates extreme locality
  loop:
   lda $0
   add #$1
   sta $0  ! Stack[0]++

   ! Go through the array incrementing every other value
   ! (Every other value to keep this file sane...)
   lda $1
   add #$1
   sta $1

   lda $3
   add #$1
   sta $3

   lda $5
   add #$1
   sta $5

   lda $7
   add #$1
   sta $7

   lda $9
   add #$1
   sta $9

   lda $11
   add #$1
   sta $11

   lda $13
   add #$1
   sta $13

   lda $15
   add #$1
   sta $15

 check:
   lda $0  ! Place 10+ into the acc - do loop will run from 10 to 20...
   sub #$20  ! 20 - 10, 11 etc until 20-20 = 0
   be out
   ba loop

 out:
   lda $0
```

hexdump

```
0000000 0001 0801 000a 0101 0000 0200 000b 0101
0000010 0001 0200 000c 0101 0002 0200 000d 0101
0000020 0003 0200 000e 0101 0004 0200 000f 0101
0000030 0005 0200 0010 0101 0006 0200 0011 0101
0000040 0007 0200 0012 0101 0008 0200 0013 0101
0000050 0009 0200 0014 0101 000a 0200 0015 0101
0000060 000b 0200 0016 0101 000c 0200 0017 0101
0000070 000d 0200 0018 0101 000e 0200 0019 0101
0000080 000f 0200 0000 0100 0001 0301 0000 0200
0000090 0001 0100 0001 0301 0001 0200 0003 0100
00000a0 0001 0301 0003 0200 0005 0100 0001 0301
00000b0 0005 0200 0007 0100 0001 0301 0007 0200
00000c0 0009 0100 0001 0301 0009 0200 000b 0100
00000d0 0001 0301 000b 0200 000d 0100 0001 0301
00000e0 000d 0200 000f 0100 0001 0301 000f 0200
00000f0 0000 0100 0014 0401 0040 0901 0021 0801
*
0000104
```

Project2_test1.s 4 Cache runs

project2_test2.s

Gemini Assembly

```
main:
 ! Push some values on the stack randomly
 lda #$10
 sta $0
 lda #$11
 sta $1
 lda #$12
 sta $2

 lda #$13
 sta $16
 lda #$14
 sta $17
 lda #$15
 sta $18

 ! Program demonstrates some locality
  loop:
   lda $0
   add #$1
   sta $0  ! Stack[0]++

   lda $1
   add #$1
   sta $1

   lda $15
   add #$1
   sta $15

   lda $2
   add #$1
   sta $2

   lda $16
   add #$1
   sta $16

 check:
   lda $0  ! Place 10+ into the acc - do loop will run from 10 to 20...
   sub #$20  ! 20 - 10, 11 etc until 20-20 = 0
   be out
   ba loop

 out:
   lda $0
```

hexdump

```
0000000 0001 0801 000a 0101 0000 0200 000b 0101
0000010 0001 0200 000c 0101 0002 0200 000d 0101
0000020 0010 0200 000e 0101 0011 0200 000f 0101
0000030 0012 0200 0000 0100 0001 0301 0000 0200
0000040 0001 0100 0001 0301 0001 0200 000f 0100
0000050 0001 0301 000f 0200 0002 0100 0001 0301
0000060 0002 0200 0010 0100 0001 0301 0010 0200
0000070 0000 0100 0014 0401 0020 0901 000d 0801
*
0000084
```

Project2_test2.s 4 Cache runs

Test 1.s

Source Code

```
// Test 1 - Test basic operation

int main(void){
 int x = 0;
 int g = 0x0, h = 0x7, j = 0xa, k = 0xf, l = 0xf;
 int y = 10;
 for(int i = 0; i < y; i++){
   x += y * i;
   g = h & j & i | k;
   g = !g;
 }
 if(x & 0xa > y){
   x = 0;
 }
}
```

hexdump

```
0000000 0001 0801 0000 0101 0000 0200 0000 0101
0000010 0001 0200 0007 0101 0002 0200 000a 0101
0000020 0003 0200 000f 0101 0004 0200 000f 0101
0000030 0005 0200 000a 0101 0006 0200 0000 0101
0000040 0007 0200 0007 0100 0006 0400 0023 0901
0000050 0007 0100 0006 1000 0000 0300 0000 0200
0000060 0002 0100 0003 0500 0007 0500 0004 0600
0000070 0001 0200 0000 0700 0001 0200 0007 0100
0000080 0001 0301 0007 0200 0011 0801 0000 0100
0000090 000a 0501 0006 0400 0029 0a01 0000 0101
00000a0 0000 0200
00000a4
```

Gemini Assembly

```
main:
 lda #$0 ! x
 sta $0
 lda #$0 ! g
 sta $1
 lda #$7 ! h
 sta $2
 lda #$10 ! j
 sta $3
 lda #$15 ! k
 sta $4
 lda #$15 ! l
 sta $5
 lda #$10 ! y
 sta $6
 lda #$0 ! i
 sta $7
for:
 lda $7
 sub $6
 be forend
 lda $7 ! i
 mul $6 ! * y
 add $0 ! += x
 sta $0 ! x = acc
 lda $2 ! h
 and $3 ! & j
 and $7 ! & i
 or  $4 ! | k
 sta $1
 nota
 sta $1
 lda $7  ! load i
 add #$1
 sta $7
 ba for
forend:
 lda $0
 and #$10
 sub $6
 bl end
 lda #$0
 sta $0
end:
```

Test_1.s 4 Cache runs

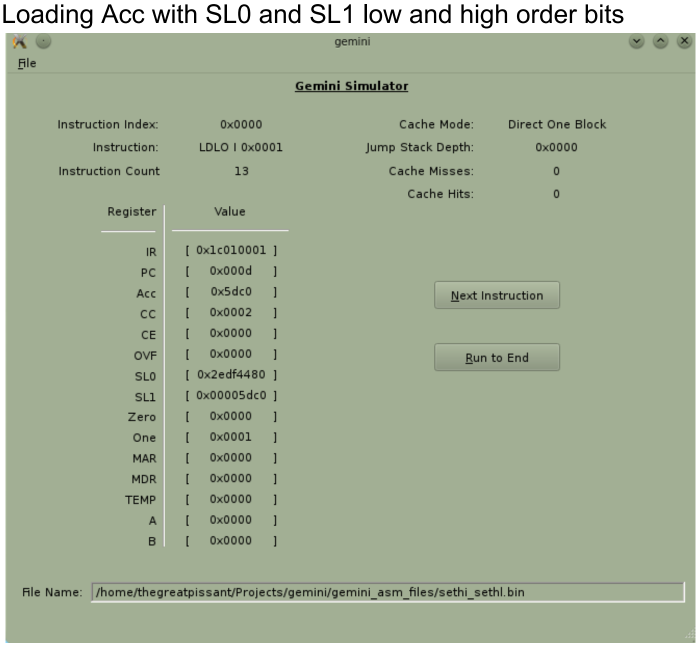
Extra credit Example programs

sethi_sethl.s Gemini Assembly

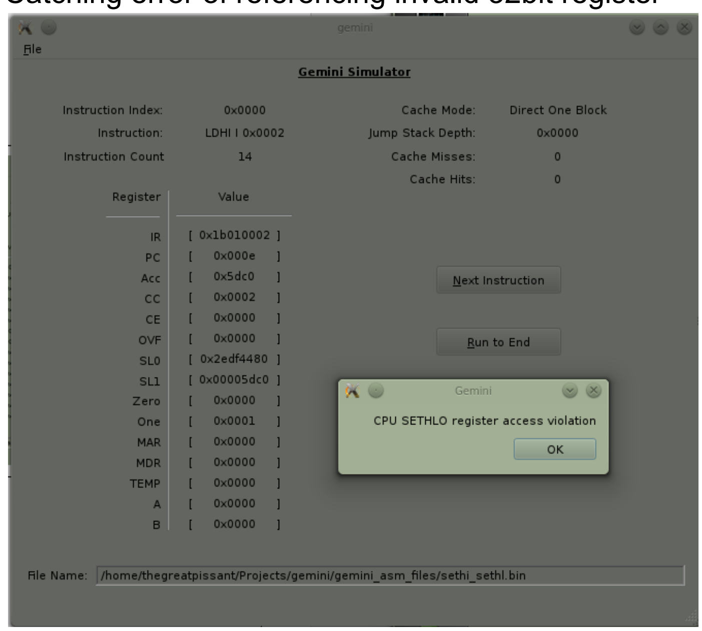
```
!example sethi and setlo
main:
    lda #$0
    sethi #$0
    sethi #$1
        lda #$32766
    setlo #$0
        lda #$24000
    setlo #$1
    mulsl
    ldhi #$0
    ldlo #$0
    ldhi #$1
    ldlo #$1
    ldhi #$2  ! error, this does not exist in cpu.
```

Result of multiplication

Loading Acc with SL0 and SL1 low and high order bits



Catching error of referencing invalid 32bit register



overflow.s Gemini Assembly

```
! Demonstrate the overflow detection for multiplication OVF
main:
      lda #$30000
      mul #$30000
```

Setting of OVF register

recursion.s Gemini Assembly

```
! Demonstrate recursion
main:
        lda #$0
    sta $0
    lda #$24
    sta $1
    jmp rec
    ba endrec
rec:
    lda $0
    sub $1
    be recdone
    lda $0
    add #$1
    sta $0
    jmp rec
recdone:
    ret
endrec:
    lda #$0
    sta $0
    lda #$25 ! here comes the stack overflow
    sta $1
    jmp rec
done:
```

Stack goes out to 25

Stack overflow at 26