CS 126 Lecture A2: TOY Programming

Outline

- **Review and Introduction**
- Data representation
- Dynamic addressing
- Control flow
- TOY simulator
- Conclusions
What We Have Learned About TOY

- What’s TOY, what’s in it, how to use it.
  - von Neumann architecture
- Data representation
  - Binary and hexadecimal
- TOY instructions
  - Instruction set architecture
- Example TOY programs
  - Simple machine language programming

What We Haven’t Learned

- How to represent data types other than positive integers?
- How to represent complex data structures at machine level?
- How to make function calls at machine level?
- What’s the relationship among TOY, C programming, and “real” computers?
Outline

- Review and Introduction
- Data representation
- Dynamic addressing
- Control flow
- TOY simulator
- Conclusions

Represent Negative Numbers Using “Two’s Complement”

<table>
<thead>
<tr>
<th>0000000000000011</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1111111111111100</td>
<td>bits flipped</td>
</tr>
<tr>
<td>·1111111111111101</td>
<td>-3</td>
</tr>
</tbody>
</table>

- Represent $-N$ with an $n$-bit 2’s complement: $2^n - N$
- To calculate $-N$, start with $N$, flip bits, and add 1
Examples

000000000000100  4
0000000000000011  3
0000000000000010  2
000000000000001  1
000000000000000  0
111111111111111  -1
111111111111110  -2
111111111111101  -3
111111111111100  -4

Leading bit is sign

Arithmetic

• Addition is carried out as if all numbers were positive

  Addition usually works

  111111111111101  -3
  000000000000100  4
  000000000000001  1

  Overflow:

  carry in to sign with no carry out

  011111111111111  2^15  -1
  000000000000001  1
  100000000000000 a negative number

• Subtraction \(-N\) is done with addition of \(N\)
Nice and Not-So-Nice Properties

- Nice properties
  - 0 is 0
  - -0 and +0 are the same
- Not-so-nice property
  - Can represent one more negative number than positive numbers
  - With n bits, can represent:
    \[ 2^{n-1} - 1 \] positive numbers (\(2^{n-1} - 1\) is maximum)
    0
    \[ 2^{n-1} \] negative numbers (-\(2^{n-1}\) is minimum)
  - A2-3 of course reader is wrong! (Replace 16s with 15s)
- Alternatives other than 2’s complement exist

Other Primitive Data Types

- big integers: could use “multiple precision”
- multiple words per integer
- required for multiply, divide?
- real numbers: could use “floating point”
- like scientific notation

- “double” type, “long long” type (for most compilers)
- character strings: could use ASCII code
  - 8 bits/character (packed/unpacked)
Outline

• Review and Introduction
• Data representation
• Dynamic addressing
• Control flow
• TOY simulator
• Conclusions

The Need for Dynamic Addressing

int a; ① int *p; ③
int a[100]; ② p = (int *) malloc(sizeof *p);

• All we have so far: “hard-wired” addresses inside instructions (R1<-MEM[D0])
• Many cases where guessing address at compile-time is impossible
  - case 1: possible for compiler to hard-wire address of a
  - case 2: difficult for compiler to hard-wire address of a[i]
  - case 3: impossible for compiler to guess address at p
• Solution:
  - Compute address at run time
  - Put address in a register
  - Augment instruction format to use address register
Indexed Addressing

For all Format 2 instructions

INDEX bit: leading bit of 2nd digit

INDEX = 1
  r1, r2: 3rd, 4th digits as in Format 1
  add r1 and r2 to get address

INDEX = 0
  take address as before

• Example: A923 means \texttt{MEM[R[2]+R[3]] \leftarrow R[1]} (9 is binary 1001)
Why “Stealing” One Bit is OK

- We only have 8 registers
- Only three bits are necessary
- But 4 bits allocated to dest register field
- So we can “steal” 1 bit

C Program for Fibonacci Array

```c
#include <stdio.h>

int a[16];
int n, i, j, k;

n = 15;
a[0] = 1;
a[1] = 1;
i = 0;
j = 1;
k = 2;

int main()
{
    do {
        a[k] = a[i]+a[j];
i++;
j++;
k++;
n--;
} while (n > 0);

for (i = 0; i < 16; i++) {
    printf("%d ", a[i]);
}
printf("\n");
}
```

- We will see how to implement the line in red using indexed addressing in TOY
TOY Version of Fibonacci Program

```
10: B10E  R1 <- 000E
11: B001  R0 <- 0001
12: B230  R2 <- 0030
14: A031  mem[31] <- 1
15: B300  R3 <- 0
16: B401  R4 <- 1
17: B502  R5 <- 2
18: 9E23  R6 <- mem[R2 + R3]
19: 9F24  R7 <- mem[R2 + R4]
1A: 1667  R6 <- R5 + R7
1B: AE25  mem[R2 + R5] <- R6
1C: 1330  R3++
1D: 1440  R4++
1E: 1550  R5++
1F: 7118  to 18 if --R1 > 0
```

• Self-modifying programs
• Special purpose computer → general purpose computer → stored program computer → self-modifying stored program computer
• Are some machines intrinsically more powerful than others?? Stay tuned.

Food for Thought
Outline

• Review and Introduction
• Data representation
• Dynamic addressing
• **Control flow**
• TOY simulator
• Conclusions

Branches and Looping

Press GO, computer either
* executes some instructions and halts
* gets caught in a loop

"infinite loop"

puzzles and/or panics programmers
(it will happen to you!)

Often more complicated
The Halting Problem

• Why doesn’t the compiler detect infinite loops and tell me?

  Can’t know whether or not a program will loop, in general

  Profound implications (stay tuned)

  ▶ Control structures (for and while) help manage branching, avoid looping

  ▶ Can always stop TOY by pulling the plug

Function Calls

• Functions can be written and used by different people

  Issues:
  • how to pass parameter values
  • how to know where to return (may have multiple calls)

  Adhere to calling conventions to get function to perform computation with different parameter values

  To implement functions (one possibility)
  • assume parameter values in register
  • assume return value in register
  • use indexed jump to return
**Example Function**

Ex: function to compute $a$ to the $b$-th power

- $a$ in R0
- $b$ in R2
- addr in R4
- result in R3

Implementation computes $a$ to the $b$-th power by looping $b$ times

- multiplying R3 by a each time

Takes care of $b==0$

---

**Example Caller**

Ex: program that calls the function on the previous slide twice to compute $x^4 + y^5$

- $x$ in mem loc 1E
- $y$ in mem loc 1F

<table>
<thead>
<tr>
<th>Line</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>B000</td>
<td>R0 &lt;- 0</td>
</tr>
<tr>
<td>11</td>
<td>911E</td>
<td>R1 &lt;- x</td>
</tr>
<tr>
<td>12</td>
<td>B204</td>
<td>R2 &lt;- 4</td>
</tr>
<tr>
<td>13</td>
<td>8420</td>
<td>R3 &lt;- $x^4$ (using function)</td>
</tr>
<tr>
<td>14</td>
<td>1530</td>
<td>R5 &lt;- R3</td>
</tr>
<tr>
<td>15</td>
<td>911F</td>
<td>R1 &lt;- y</td>
</tr>
<tr>
<td>16</td>
<td>B205</td>
<td>R2 &lt;- 5</td>
</tr>
<tr>
<td>17</td>
<td>8420</td>
<td>R3 &lt;- $y^5$ (using function)</td>
</tr>
<tr>
<td>18</td>
<td>1535</td>
<td>R5 &lt;- $x^4 + y^5$</td>
</tr>
</tbody>
</table>
Function Call Demo

The Use of Registers vs. Memory for Function Calls

- Stack is implemented using main memory
- Review:
  - Call: push environment (registers and PC)
  - Call: push function parameters
  - Inside a function: look for parameters on the stack
  - Return: restores environment by popping stack
- Registers can still be used as optimizations
Outline

• Review and Introduction
• Data representation
• Dynamic addressing
• Control flow
• **TOY simulator**
• Conclusions

Availability

```bash
cp ~cs126/toy/toy.c .
c c toy.c -o TOY
TOY < myprog.toy
```

Example programs also available

```bash
TOY < ~cs126/toy/horner.toy
```

• Better yet, download java version from announcement page
• Edit “toy.html”, reopen it in browser
#include <stdio.h>
short int R[8], mem[256]; pc = 16;
main()
{
    int i, inst, op, addr, r0, r1, r2;
    for (i = 0; i < 256; i++) mem[i] = 0;
    for (i = pc; i < 256; i++)
        if (scanf("%X", &mem[i]) == EOF) break;
    do
    {
        inst = mem[pc++];
        op = (inst >> 12) & 0XF;
        addr = inst & 0XFF;
        r0 = (inst >> 8) & 0X7;
        r1 = (inst >> 4) & 0X7; r2 = inst & 0X7;
        if (inst & 0X0800)
            addr = (R[r1]+R[r2]) & 0X0FF;
    }
}

TOY Simulator (Part 2: execute)
switch (op)
{ 
    case 0: break;
    case 1: R[r0] = R[r1] + R[r2]; break;
    case 2: R[r0] = R[r1] - R[r2]; break;
    case 3: R[r0] = R[r1] * R[r2]; break;
    case 4: printf("%X\n", R[r0]); break;
    case 5: pc = addr; break;
    case 6: if (R[r0]>0) pc = addr; break;
    case 7: if (--R[r0]) pc = addr; break;
    case 8: R[r0] = pc; pc = addr; break;
    case 9: R[r0] = mem[addr]; break;
    case 10: mem[addr] = R[r0]; break;
    case 11: R[r0] = addr; break;
    case 12: R[r0] = R[r1] ^ R[r2]; break;
    case 13: R[r0] = R[r1] & R[r2]; break;
    case 14: R[r0] = R[r0] >> addr; break;
    case 15: R[r0] = R[r0] << addr; break;
}
Outline

• Review and Introduction
• Data representation
• Dynamic addressing
• Control flow
• TOY simulator
• Conclusions
  - Relationships among machine language programming, C programming, TOY machine, and “other” machines
Engineering and Theoretical Implications of Simulator

- Translate SIMULATOR to TOY program? why not?? ✔

**BOOTSTRAPPING**
- build ‘first’ machine
- implement simulator
- modify simulator to try new designs
  (still going on!)

- Theoretically, any von Neumann machine can simulate any other von Neumann machine--all of them have the same “power”!! (More later)

What We Have Learned

- Two’s complement
  - How to represent negative numbers
  - How to perform addition and subtraction
  - Understand overflow

- How to use indexed addressing to access data structures

- Function calls
  - Passing parameters in registers
  - Save and restore PC