Lecture 4, Jan 30, 2024

Addressing Modes

- *Inherent addressing*: address, data, or register is inherently specified by the instruction
  - Used by instructions that either don't take any operands, or operands are fixed by the instruction
  - e.g. `NOP`
- *Immediate addressing*: a constant value is explicitly stored within the code and used by the instruction
  - e.g. `MOV R0, 0xAF`
  - For word sizes above 1 byte, this will have to respect endianness
  - Best used to implement constants, since it is faster but not modifiable
- *Register addressing*: operands are registers
  - e.g. `MOV R0, R1`
  - Generally accumulator based or RISC architectures restrict where accumulators and general purpose registers can appear as operands
  - Best used to implement transfers or variable assignment
  - Limits us to the width of our registers
- *Direct addressing*: specifies a memory location to access (i.e. a constant pointer)
  - e.g. `MOV [0xA000], 0xAF`
  - The addresses themselves are stored within the instruction and are constant
  - Best used to implement global variables, constant loading, memory-mapped I/O, etc.
  - If we only use direct addressing, we need to make all variables global in scope and pre-assign fixed addresses to every variable
- *Indirect addressing*: the memory location in the operand contains the address of the data to be used by the instruction (i.e. a true pointer)
  - e.g. `MOV R0, @[0xA000]`
    - The memory at `0xA000` is read, and interpreted as a memory address, and then data from that address is put into `R0`
  - Best used to implement pointers, arrays, etc, to implement data structures and pass data to subroutines
  - However, this uses 2 or more memory accesses per instruction, which are slow
- The below addressing modes are outside the minimum required set, so some microcontrollers might not support them, especially RISC
- *Register indirect addressing*: the register in the operand contains the address of the data to be used
  - e.g. `MOV R0, @R1` or `MOV R0, @@R1` (dereference twice, usually only on CISC)
  - Allows us to dynamically create and destroy new pointers
  - Allows doing math on pointers for e.g. array access with variable indices
  - Address space is again limited to register width, so we might not be able to access all the memory
  - Can become a vector for attack if a malicious actor can put data into the register
- *Indexed addressing*: a family of addressing modes to used to add indexing many of the previous modes
  - Relative addressing: relative to current program counter, e.g. `BRR 0x05`
    - Useful for flow control
  - Base register: adding offset to register value, e.g. `MOV R0, @[R1 + 5]`, `MOV R0, @[R1 + R2]`
    - This is very useful for array dereferencing and data structures in general
- Other exotic modes (these are much less common)
  - Base plus index plus offset: effective address is the sum of all 3 (for supporting 2D arrays)
  - Scaled: effective address is the base address plus a scale times an index (for addressing arrays)
  - Register auto-increment/decrement

Example 1: Factorial

- Example: compute the factorial of a number stored in `R0`, assume 3 general purpose registers `R0` to `R2` are available and ignore checking pre-conditions
- Assume for the purposes of this example that we cannot modify `R0`

- Translate from C code:

```c
int n;
int fact = 1;
for (int i = 1; i <= n; i ++) {
    fact *= i;
}
```

- Create variable assignment table:

Variable	Assignment	Scope
`n`	`R0`	Global
`fact`	`R1`	Global
`i`	`R2`	Local to `for`

- Code generation:

```
    MOV     R1, 0x01    ; int fact = 1; // note assumption that registers are 8-bit
    MOV     R2, 0x01    ; int i = 1;
loop:
    CMP     R2, R0
    BGT     end         ; if (i > n) { break; }
    MUL     R1, R2      ; fact = fact * i;
    INC     R2          ; i ++;
    BRA     loop
end:
```

- Note we used labels here for the relative branch instructions; if not, we would have to compute the offsets ourselves
  - Assuming each instruction takes 2 bytes, the `BGT` would have an offset of `0x06` while the `BRA` would have `0xF6` (-10)

Example 2: Square Root

- Example: calculate the square root of a number S in `R0` by the following procedure:
  - Set $x_n = S/2$
  - Iterate $x_{n+1} = \frac{1}{2}\left(x_n + \frac{S}{x_n}\right)$
- Code generation:

```
    MOV     R1, R0      ; int x_n = S;
    DIV     R1, 2       ; x_n = x_n / 2;
loop:
    MOV     R2, R0      ; int temp = S;
    DIV     R2, R1      ; temp = temp / x_n;
    ADD     R1, R2      ; x_n = x_n + temp;
    DIV     R1, 2       ; x_n = x_n / 2;
    BRA     loop
```