Lecture 6, Feb 13, 2024

Code Reuse

Inline Functions and Macros

- *Inline functions* or *macros* are code blocks defined once and duplicated by the assembler or compiler each time it's used
- Since no function call is involved, they have no overhead, can be easily optimized, and is simple to implement
- However duplicating them each time means inefficient code memory usage, so we should keep them short
- Usually they will make explicit assumptions about the program state (e.g. operands are in certain registers)
  - Should only contain re-locatable code (i.e. no jumping to fixed addresses within the code block)
  - Also cannot contain recursion!
- Best used for code that are executed often (e.g. inside a loop), but called/used in few other places

Subroutines

- *Subroutines* are defined and compiled once; to execute them, code execution jumps to the memory they occupy and jumps back after they're done
  - The code does not strictly need to be relocatable
  - Reduces code memory usage since only one copy is needed
  - However, calling and passing arguments introduces some overhead, which could amount to a significant amount if the subroutine itself is short
  - Also harder to optimize since the subroutine code needs to be as generic as possible, so each call cannot be optimized by itself
- A call instruction saves the current PC on the stack and jumps to the subroutine
  - A return instruction loads the saved PC and jumps to that location, returning to the point before the subroutine call
- A safe subroutine has to back up any registers it uses on the stack as to not interfere with any calling code
  - Some CISC microcontrollers might do this automatically on call instructions
  - This is uncommon today, since it introduces large overhead and backs up all registers, even the ones that aren't used in the subroutine
    - This is still done commonly for interrupts
  - At the start of the function, we push any registers we need to use, then at the end of the function we pop in opposite order to restore them
    - Sometimes we may back up SFRs as well
- Variable passing can be accomplished in a number of ways:
  - Using registers can work if we need few parameters, and saves overhead
  - Using memory (fixed locations) require agreement on locations, and struggles for variable-length data and recursion
  - Using the stack is the best and most general option, which is preferred for an automatic implementation by compilers
    - To pass variables or return values on the stack, we need a special addressing mode that allows us to access earlier points in the stack, since the top of the stack will store the PC

Example: Call/Return Compilation Example

```
int max(int a, int b) {
    if (a > b)
        return a;
    else
        return b;
```

```
}
void main(void) {
    int c = 1, d = 6;
    int e = max(c, d);
}
```

- Assume:
  - `main()` starts at `0x0100`, `max()` starts at `0xF000`
  - Instructions with register-only operands are one byte long, while others are one byte plus any immediates
  - We have 3 registers `R0`, `R1`, `R2`
  - `int`s and registers are 16-bit
  - Branching instructions use 8-bit offsets
  - Stack grows upwards, SP points to next available address
  - Big-endian system
  - Stack-based call and return
    - Note: in reality the compiler will likely use registers for argument passing, or optimize this into an inline function/macro due to its short length
- Variable assignment:
  - In `main()`:
    - `c` →`R0`
    - `d` →`R1`
    - `e` →`R2`
    - All variables have scope ending only when program terminates
  - In `max()`:
    - `a` →`R0`
    - `b` →`R1`
    - Both variables have scopes that exist through the entire function, and only in the function
    - Since `c` and `d` are still in scope at this point, we need to backup the registers
- Code for `main()`:

```
0x0100      MOV     R0, #0x0001
0x0103      MOV     R1, #0x0006
0x0106      PUSH    R0           ; First arg
0x0107      PUSH    R1           ; Second arg
0x0108      ADD     SP, #0x0002  ; Space for retval
0x010B      CALL    0xF000
0x010E      POP     R2           ; Retrieve return value
0x010F      SUB     SP, #0x0004  ; Clear args from stack
```

- Code for `max()`:

```
0xF000      PUSH    R0                ; Back up registers
0xF001      PUSH    R1
0xF002      MOV     R0, [SP - 12]     ; Load arg a
0xF005      MOV     R1, [SP - 10]     ; Load arg b
0xF008      CMP     R0, R1            ; if (a <= b) goto else;
0xF009      BLE     #0x0008
0xF00B      MOV     [SP - 8], R0      ; Return a (note 2 bytes for offset)
0xF00E      BRA     #0x0003
0xF010      MOV     [SP - 8], R1      ; Return b
0xF013      POP     R1                ; Restore registers
0xF014      POP     R0
0xF015      RET
```

- Stack contents are shown in the table below
  - By the start of the function call SP is at `0x2008`, right after PC
  - By the end of the call the SP is back to `0x2005`, exactly where it was before the call
  - At the end of the program, the stack pointer should be back to `0x2000` if we cleaned up properly

Address	Data	Comment
`0x200B`	...	`R1` backup, low byte
`0x200A`	...	`R1` backup, high byte
`0x2009`	...	`R0` backup, low byte
`0x2008`	...	`R0` backup, high byte
`0x2007`	`0x0E`	PC, low byte
`0x2006`	`0x01`	PC, high byte
`0x2005`	?	Space for return
`0x2004`	?	Space for return
`0x2003`	`0x06`	`R1`, low byte
`0x2002`	`0x00`	`R1`, high byte
`0x2001`	`0x01`	`R0`, low byte
`0x2000`	`0x00`	`R0`, high byte