Outline

• Architectural models
• A little about the 68HC11
  – Memory map
  – Registers
  – A little bit of assembly (never did us any harm)
• The program counter
• The stack
  – Procedure calls
  – Local variables
• ‘for’ and ‘while’ loops
• Stack problems
Architectural Models

• von Neumann architecture (Pentium / 68HC11)
  – Shared data and program space
  – Contents addressable by location
  – Execution occurs in a sequential fashion
    • Unless told to do otherwise

• Harvard architecture
  – Separate program and data space (PIC)

• And others…
  – SIMD, etc.
Example Architecture

• For this lecture
  – Simple 16-bit von Neumann architecture
  – 8-bit data / 16-bit address CPU (68HC11)
  – Full 64K of memory
  – All memory locations exist
  – Input / Output devices
    • Keyboard
    • Hard disc drive
### 68HC11 Memory Map

<table>
<thead>
<tr>
<th>Start</th>
<th>End</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>00FF</td>
<td>Page Zero RAM (fast)</td>
</tr>
<tr>
<td>0100</td>
<td>0FFF</td>
<td>RAM</td>
</tr>
<tr>
<td>1000</td>
<td>103F</td>
<td>Special Control Registers</td>
</tr>
<tr>
<td>1040</td>
<td>EFFF</td>
<td>RAM</td>
</tr>
<tr>
<td>F000</td>
<td>FFFF</td>
<td>ROM</td>
</tr>
</tbody>
</table>

- Special Control registers are movable and the ROM / RAM boundary is implementation dependent.
Instruction Syntax

• Assembler syntax
  
  `<label> <opcode> <parameters>`

• Parameters
  
  # immediate
  $ hex value
  % binary
  @ octal
  ‘C’ ASCII character

• Example

  DOTHIS
  
  LDAA #$44
  LDAB %#01000100
Instruction Types

• Instruction types
  – Arithmetic
    • ABA
  – Logic
    • ANDA #$F6
  – Load and Store
    • LDAA $EC0B
  – Transfer control
    • JSR $6000
  – Special
    • WAI
Addressing Modes

• Inherent
  – No arguments needed
    • ABA

• Immediate
  – Value specified immediately after opcode
    • LDAA #$08

• Direct
  – Supplied memory location on “zero” (demand) page
    • LDAA $59

• Extended
  – Supplied memory location
    • LDAA $0600
Addressing Modes

- Indexed (INDX and INXY)
  - Relative to the X or Y registers
    - LDAA $05,X

- Relative
  - Used in branch instructions
    - BNE $06

- Mixed mode
  - BRCLR 03,X #$A8 $FF00
    - If !(*(X+3) & 0xA8) then PC=$FF00
Registers

• Fast access memory locations
  – No need to place address on address bus

• 68HC11 is typical and has few
  – 8 bit
    • A, B
    • Flags (Condition Codes)
  – 16 bit
    • D (A and B combined)
    • X, Y
    • SP, PC

• What happens to program variables?
The Program Counter

• Address of next instruction to execute

• CPU is in a loop
  – Fetch instruction from where PC points
  – Decode instruction
  – Increment PC (by instruction size)
  – Obey instruction
    • Fetch more data (and increment PC)
    • Manipulate the data
The Stack

• Just a location (high) in memory
• Special assembly instructions manipulate it
  – PSHA, PULB, LDS, INS, DES, TSX
• 68HC11
  – SP points to the “top” of the stack
  – PSHA: store A where SP points then SP=SP-1
  – PULA: SP=SP+1, load A with where SP points

- What is the effect of pushing A?
- What is the effect of pulling A?
- What happens when the stack gets “big”?
Subroutines

- Sample 68HC11 code

MAIN:
  JSR HERE ; 6 cycles
  RTS

HERE:
  RTS ; 5 cycles

<table>
<thead>
<tr>
<th>Cycle</th>
<th>JSR (DIR)</th>
<th>JSR (EXT)</th>
<th>JSR (IND,X)</th>
<th>JSR (IND,Y)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Addr</td>
<td>Data</td>
<td>R/W</td>
<td>Addr</td>
</tr>
<tr>
<td>1</td>
<td>OP</td>
<td>9D</td>
<td>1</td>
<td>OP</td>
</tr>
<tr>
<td>2</td>
<td>OP + 1</td>
<td>dd</td>
<td>1</td>
<td>OP + 1</td>
</tr>
<tr>
<td>3</td>
<td>00dd</td>
<td>(00dd)</td>
<td>1</td>
<td>OP + 2</td>
</tr>
<tr>
<td>4</td>
<td>SP</td>
<td>Rtn lo</td>
<td>0</td>
<td>hll</td>
</tr>
<tr>
<td>5</td>
<td>SP - 1</td>
<td>Rtn hi</td>
<td>0</td>
<td>SP</td>
</tr>
<tr>
<td>6</td>
<td>SP - 1</td>
<td>Rtn hi</td>
<td>0</td>
<td>SP - 1</td>
</tr>
<tr>
<td>7</td>
<td>SP - 1</td>
<td>Rtn hi</td>
<td>0</td>
<td>SP - 1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cycle</th>
<th>RTS (INH)</th>
<th>Addr</th>
<th>Data</th>
<th>R/W</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>OP</td>
<td>39</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>OP + 1</td>
<td>-</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>SP</td>
<td>-</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>SP + 1</td>
<td>Rtn hi</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>SP + 2</td>
<td>Rtn lo</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
High Level Language Calls

• Is this procedure call free?

    int getval(int p1, int p2, int p3, int p4) {
        return p2;
    }

    int main(int argc, char **argv) {
        return getval(6, 7, 8, 9);
    }

• How long does it take?
Procedure Calls

• C version

```c
int getval(void) {
    return 0;
}
int main(int argc, char **argv) {
    return getval();
}
```

• Hand assembly version

```assembly
.GETVAL:
    LDD #0
    RTS
.MAIN:
    JSR _GETVAL
    TSX
    RTS
```

• Compiler assembly version

```assembly
.GETVAL:
    LDD #0
    RTS
.MAIN:
    TSX
    JSR _GETVAL
    TSX
    RTS
```
Calling Conventions

• Different in different languages (and compilers)
  – C
    • Caller cleans up
  – Pascal
    • Callee cleans up (saves one instruction per call)

• The C calling convention (on 68HC11)
  – Place parameters onto the stack (reverse order)
  – Jump to the subroutine
  – Fix the stack

• The C returning convention (on 68HC11)
  – Load result into register D
  – Return
High Level Procedure Calls

• C Version

```c
int getval(int p1, int p2, int p3, int p4) {
    return p2;
}

int main(int argc, char **argv) {
    return getval(6, 7, 8, 9);
}
```
Assembly Version

_main:
;   argv  ->  12,x
;   argc  ->  10,x
    pshx
    pshx
    pshx
    pshx
    tsx
    ldd  #6
    std  0,x
    ldd  #7
    std  2,x
    ldd  #8
    std  4,x
    ldd  #9
    std  6,x
    jsr _getval
    tsx
    pulx
    pulx
    pulx
    pulx
    rts

_getval:
;   p4  ->  8,x
;   p3  ->  6,x
;   p2  ->  4,x
;   p1  ->  2,x
    tsx
    ldd  4,x
    rts

 argv
 X+12
 X+10
 X+8
 X+6
 X+4
 X+2

 argc
 X+12
 X+10
 X+8
 X+6
 X+4
 X+2

 X
 X+8
 X+6
 X+4

 0009
 0008
 0007
 0006

 0009
 0008
 0007
 0006

 Return

 SP  X
 SP  X
 SP  X
 SP  X
Local Variables

• Local variables go on the stack
  – We have already seen this

```c
int main(int argc, char **argv) {
  char byte;
  int integer;

  byte = 1;
  integer = 2;

  return byte + integer;
}
```

• What happens if there are many?
• What if they are large?
Loops

- **for loops are really while loops**

  ```
  for (count = 0; count < 0xFF; count++)
    x++;
  ```

- **is equivalent to**

  ```
  count = 0;
  while (count < 0xFF) {
    x++;
    count++;
  }
  ```

```
Loops and Variables

• What is the output of this program?

```c
int main(int argc, char **argv) {
    int pos, y[1], result;

    result = 2;
    for (pos = 0; pos <= 1; pos++)
        y[pos] = 100;

    printf("pos:%d result:%d\n", pos, result);
    return 0;
}
```
Simplified Version

• C Version

int main(int argc, char **argv) {
    int pos, y[1], result;

    pos = 0;
    result = 2;
    y[1] = 100;

    printf("pos:%d result:%d\n", pos, result);
    return 0;
}

• Assembly Version

;   result -> 0,x
;       y -> 2,x
;       pos -> 4,x

;   pos = 0
    ldd #0
    std 4,x
;   result = 2
    ldd #2
    std 0,x
;   y[1] = 100
    ldd #100
    std 4,x
Buffer-Overrun Attack

- Y is just a pointer into the stack
- Indices are just arithmetic expressions
- \( y[n] = (\text{char}*)y[0] + n \times \text{sizeof} * y \)
- What is the result of \( y[2]=100 \)?

- How is this exploited in a buffer overrun attack on a program?
Out of Scope (Dangling Pointers)

• What happens to the stack here?

```c
char *doit(void) {
    char *x, buffer[2];

    x = buffer;
    return x;
}

int main(int argc, char **argv) {
    char *answer = doit();
    answer[2] = 'G';
    return 0;
}
```
Stack Problems

• What happens if the stack gets too large?
• How does this exhibit itself in your program?

• What happens if you overwrite the stack?
• How does this exhibit itself in your program?

• What happens when a pointer goes out of scope?
• How does this exhibit itself in your program?

• What happens when there are multiple threads?
Behaviours To Look For

- Crash when a function returns
  - The return address on the stack might be corrupt
- A passed parameter is different from that passed
  - The parameter might have been overwritten
- Variables changing values without reason
  - A dangling pointer might be altering it
- Crash in (otherwise reliable) system routine
  - Stack overflow might be occurring
- Memory allocation has side-effects
  - Corruption in heap can trickle down to the stack
- Bugs that go away in the debugger
  - The stack will (probably) be different
Conclusion

• Stack overflow
  – Is the compiler always right?
  – The architecture can introduce errors
  – Some errors aren’t logic errors
  – Non-logic errors are hard to identify
  – Don’t always assume your program is wrong!

• Stack corruption can easily occur
  – This is a logic error!

• Stack problems aren’t always easy to find