LAST TIME

- Enhanced our processor design in several ways
- Added branching support
  - Allows programs where work is proportional to the input values
- Added a large memory
  - Small, in-processor memory is now called the *register file*
  - Move data from main memory into registers for computation
- Two architectures:
  - Load/Store Architecture (RISC)
  - Multiple operand types (CISC)
Programs with Loops

- Can implement more interesting programs now
  - e.g. multiplication, using our processor’s simple instructions

```c
int mul(int a, int b) {
    int p = 0;
    while (a != 0) {
        if (a & 1 == 1)
            p = p + b;
        a = a >> 1;
        b = b << 1;
    }
    return p;
}
```

**Control Instruction Operation Table**

<table>
<thead>
<tr>
<th>Control</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0001</td>
<td>ADD A B</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>0111</td>
<td>BRZ A Addr</td>
</tr>
<tr>
<td>1000</td>
<td>AND A B</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>1100</td>
<td>SHL A</td>
</tr>
<tr>
<td>1110</td>
<td>SHR A</td>
</tr>
</tbody>
</table>

**Register Value Table**

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>A</td>
</tr>
<tr>
<td>1</td>
<td>B</td>
</tr>
<tr>
<td>2</td>
<td>Tmp</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>P</td>
</tr>
</tbody>
</table>
Building Blocks

- Multiply function is a useful building block for other programs!
- Example: discriminant of quadratic fn. \( ax^2 + bx + c \)
  
  ```c
  int discriminant(int a, int b, int c) {
    return b * b - 4 * a * c;
  }
  ```

- We know we can implement this in our instruction set
- Would like to reuse our `mul()` function for this
  - Can implement \( 4 \times (\ldots) \) by shifting left by 2 bits
  - Still need two multiplies to implement this function
    
    ```c
    int discriminant(int a, int b, int c) {
      return mul(b, b) - mul(a, c) << 2;
    }
    ```

- How do we do this?
How do we use `mul()` as a subroutine?

Need to know how `mul()` takes its arguments, and returns its result

- Decided that R0 and R1 were inputs, and R7 is the product
  - Just pass `mul()` our inputs, then get result out of R7

Need a way to transfer control to `mul()`

- ...then, `mul()` has to get back to our code somehow
  - `Hmm...`

Is this the whole picture?

- No! `mul()` also uses R2, R3, R4 internally
  - The calling code needs to avoid using these registers
**Subroutines**

- Three major problems we need to solve:
  - Need a way to pass arguments and return values between a caller and the subroutine
  - Need a way to transfer control from a caller to the subroutine, then return back to caller
  - Need to isolate subroutine’s state from caller’s state

- First problem is primarily a design issue
  - Figure out a convention, then stick with it

- The second and third points are the harder ones
SUBROUTINES AND CALLERS

- Our program:
  ```c
  int discriminant(int a, int b, int c) {
    return mul(b, b) - mul(a, c) << 2;
  }
  ```

- Need to invoke our `mul()` function twice
  - Hard part is not jumping to the `mul()` function...
  - Need to get back to wherever we called it from!

- Can we do this with our current processor architecture?
  - No! 😞
    - Processor only supports constants for branch addresses

---

<table>
<thead>
<tr>
<th>Control</th>
<th>Operation</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0001</td>
<td>ADD</td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>0011</td>
<td>SUB</td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0111</td>
<td>BRZ</td>
<td>A</td>
<td>Addr</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1100</td>
<td>SHL</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>1110</td>
<td>SHR</td>
<td>A</td>
<td></td>
</tr>
</tbody>
</table>
**Subroutine Return-Addresses**

- **Problem:**
  - Can only load constants into the program counter
  - \( PC = PC + 1 \)
  - \( PC = \text{branch_addr} \)

- **Need ability to specify branch address in a register as well**

- **To call a subroutine:**
  - Pass the return address in another register
  - Subroutine jumps back to return-address at end
Updated logic:

```c
if (Opcode == BRZ && RegA == 0) {
    if (BranchMode == 1)
        ProgCtr = RegB;
    else
        ProgCtr = BranchAddr;
}
```
Now we can use **mul()** as a subroutine
- **mul()** convention: expect the return-address in R6

To call our subroutine:
- Move return-address into R6, then jump to **MUL** address
  ```
  ...  # Set up other args
  MOV RET, R6
  BRZ 0, MUL
  RET:
  ...  # Result is in R7!
  ```
- Note: Introduced **MOV** instruction
- (Could write **ADD RET, 0, R6** but that is a bit silly...)

**mul()** parameters:

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>A</td>
</tr>
<tr>
<td>1</td>
<td>B</td>
</tr>
<tr>
<td>6</td>
<td>return addr</td>
</tr>
<tr>
<td>7</td>
<td>P (output)</td>
</tr>
</tbody>
</table>

**MUL:**
```
XOR R7, R7, R7
```  

**WHILE:**
```
BRZ R0, DONE
AND R0, 1, R2
BRZ R2, SKIP
ADD R7, R1, R7
```  

**SKIP:**
```
SHR R0, R0
SHL R1, R1
BRZ 0, WHILE
```  

**DONE:**
```
BRZ 0, R6
```
COMPUTING THE DISCRIMINANT

What about our discriminant function?

```c
int discriminant(a, b, c) {
    return mul(b, b) - mul(a, c) << 2;
}
```

Still a huge pain to implement!!

- Only have 8 registers
- `mul()` now uses 7 registers
  - (...if our instructions could encode constants, we would use only 5...)

Actually, why should callers of `mul()` have to care what registers `mul()` uses internally?!

- **Abstraction**: Subroutine’s caller shouldn’t have to understand subroutine’s internals in order to use it
**Subroutines and Registers**

- In fact, we would really only like to think about:
  - How to pass arguments to subroutine
  - How to get return-value back from subroutine
- Ideally, would like subroutines to use registers however they want to
  - Somehow, save registers at start of subroutine call
  - Restore registers when subroutine returns to caller
- If a complex subroutine runs out of registers:
  - Save values of some registers, then reuse them
  - When finished, can restore old values of registers

- Can implement these features with a [stack](#)
Stacks

A Last In, First Out (LIFO) data structure

Components:
- A region of memory
- A stack pointer SP
  - Invariant: SP always points to top of stack

Two operations:
- **PUSH** \( \text{Reg} \) – pushes \( \text{Reg} \) onto stack
  - \( \text{SP} = \text{SP} - 1 \)
  - \( \text{Memory}[\text{SP}] = \text{Reg} \)
- **POP** \( \text{Reg} \) – pops top of stack into \( \text{Reg} \)
  - \( \text{Reg} = \text{Memory}[\text{SP}] \)
  - \( \text{SP} = \text{SP} + 1 \)

IA32 convention: stack grows “downward”
- Pushing a value decrements SP
- Popping a value increments SP
- Will use this convention in our examples
USING THE STACK

Can simplify our subroutine implementations
- Pass arguments, return-values via registers
- Subroutines will save and restore other registers they use
- Subroutines must leave stack in the same state they found it

Example: Updated `mul()`
- R0, R1 are arguments
- R6 is return address
- R7 is result
- Function uses R2, so save it at start, then restore at end

```
MUL:
PUSH R2
XOR R7, R7, R7

WHILE:
BRZ R0, DONE
AND R0, 1, R2
BRZ R2, SKIP
ADD R7, R1, R7

SKIP:
SHR R0, R0
SHL R1, R1
BRZ 0, WHILE

DONE:
POP R2
BRZ 0, R6
```
**Discriminant Function**

- Our discriminant function:
  ```c
  int discriminant(int a, int b, int c) {
    return b * b - 4 * a * c;
  }
  ```

- Register usage:
  - `a = R0`
  - `b = R1`
  - `c = R2`
  - Result into R7

- Example code for function:

```assembly
DISCR:
  PUSH R0          # Save A
  MOV  R1, R0      # R0 = B, R1 = B
  MOV  RET1, R6    # Set up for call
  BRZ  0, MUL      #     mul(B, B)

RET1:
  POP  R0          # Restore A
  PUSH R7          # Save B*B
  MOV  R2, R1      # R1 = C
  MOV  RET2, R6    # Set up for call
  BRZ 0, MUL       #     mul(A, C)

RET2:
  POP R1           # Restore B*B
  SHL R7, R7       # Multiply A*C by 4
  SHL R7, R7
  SUB R1, R7, R7   # R7 = B*B - 4*A*C
  DONE
```
DISCRIMINANT FUNCTION (2)

- Significantly easier to implement than before...
- Computed $b^2$ first
  - Needed to save $a$ before calling `mul()`
- Saved result of first multiply operation
  - Pushed R7 onto stack
  - Popped into R1

An example of using stack to save and restore intermediate values

DISCR:

```assembly
PUSH R0          # Save A
MOV R1, R0      # R0 = B, R1 = B
MOV RET1, R6    # Set up for call mul(B, B)
BRZ 0, MUL

RET1:
POP R0          # Restore A
PUSH R7          # Save B*B
MOV R2, R1      # R1 = C
MOV RET2, R6    # Set up for call mul(A, C)
BRZ 0, MUL

RET2:
POP R1           # Restore B*B
SHL R7, R7      # Multiply A*C by 4
SHL R7, R7
SUB R1, R7, R7  # R7 = B*B - 4*A*C
DONE```

16
ARGUMENTS AND RETURN-ADDRESS

- There’s no reason not to pass the arguments and return address on the stack as well!

- Code for calling \texttt{mul(b, b)}:
  
  ```
  MOV R1, R0 # R0 = B, R1 = B
  MOV RET1, R6 # Set up for call
  BRZ 0, MUL # mul(B, B)
  
  RET1:
  ```

- Instead, introduce two new instructions:
  - \texttt{CALL Addr}
    - Pushes PC of \textit{next} instruction onto stack
    - Then sets PC = Addr
  - \texttt{RET}
    - Pops top of stack into PC

- No longer need our \texttt{RET1, RET2, \ldots} labels, etc.
Arguments and Return-Address (2)

- New strategy for subroutine calls:
  - Caller pushes subroutine arguments onto stack
  - Caller uses `CALL` to invoke subroutine
  - Subroutine uses stack to perform its computations
    - Access arguments, use stack for temporary storage
    - At end, restore stack to original state at time of call
  - Subroutine uses `RET` to return to the caller
ACCESSING ARGUMENTS

- How does subroutine access its arguments?
- Our discriminant function:

```
DISCR:
PUSH R1   # R7 = mul(B, B)
PUSH R1
CALL MUL
...
```

- For subroutine to access arguments, **definitely** need indirect memory access support!

- Multiply function arguments:
  - \([SP + 2]\) = first argument
  - \([SP + 1]\) = second argument
  - Remember: our stack grows downward
    - Values pushed earlier are at *higher* addresses
ACCESSING ARGUMENTS (2)

- How does subroutine access its arguments?
- Our discriminant function:
  ```
  DISCR:
  PUSH R1   # R7 = mul(B, B)
  PUSH R1
  CALL MUL
  ...
  ```

- Alternative to indirect memory access?
  - Subroutine pops off return-address to access args, then later restores stack for return to caller
  - “What could possibly go wrong?”
ACCESSING ARGUMENTS (3)

- `mul()` routine also modifies certain registers
  - e.g. R2 is used to compute (A & 1) temporary value
  - R0 and R1 are also modified
  - Need to push old values onto stack so we can restore these values later

Problem:
- Makes it much harder to reference our function arguments!
  - Now args are at [SP + 5] and [SP + 4]
  - If subroutine has to push other values onto stack as it executes, these offsets change again
ACCESSING ARGUMENTS (4)

- **Solution:** introduce a reference-point on the stack for accessing arguments
  - **Example:** a BP ("base pointer") register
    - Set BP = SP, **before** saving registers that are locally modified
    - Since we change BP, need to save it first before we store SP into it
  - **Now arguments can be accessed using BP as a reference-point**
    - Argument 1 is at location [BP + 3]
    - Argument 2 is at location [BP + 2]
    - Return address is at location [BP + 1]
    - Locally modified registers stored below BP on stack
ACCESSING ARGUMENTS (5)

- Our discriminant function:
  ```
  DISCR:
  PUSH R1  # R7 = mul(B, B)
  PUSH R1
  CALL MUL
  ...
  ```

- `mul()` routine, updated with new argument-passing mechanism:
  ```
  MUL:
  PUSH BP      # Save old BP
  MOV  SP, BP  # Copy SP to BP
  PUSH R0      # Save registers
  PUSH R1      # that we modify
  PUSH R2      # locally.
  MOV [BP + 3], R0  # Arg 1
  MOV [BP + 2], R1  # Arg 2
  ...
  ```
Stack Frames

- A stack frame is the portion of the stack allocated for a specific procedure call.
- Includes arguments, return address, and local state used by the subroutine.
- BP called the frame pointer:
  - Since SP can move, values are accessed via the frame pointer.
  - Since number/size of arguments is known, can tell where stack frame starts.
- Very common strategy for supporting procedures
  - IA32 has a BP register for storing frame pointer.
TRANSITION TO IA32

- Our model has gotten quite sophisticated
  - Memory access, including indirect memory access
  - Branching instructions, plus “branch-to-register”
  - Introduce a “stack” abstraction for saving registers, managing procedure arguments, return addresses
  - Introduce “stack frames” and frame pointers for easy access to procedure arguments and local variables

- Time to move to IA32 instruction set architecture
  - Like example, has only 8 general-purpose registers...
  - Provides rich support for all of these abstractions
    - Includes many special-purpose registers devoted to these abstractions
    - A very rich instruction set that makes it easy to use them
IA32 Overview

- IA32 is the instruction-set architecture for Intel Pentium-family processors
  - 32 bit and 64 bit processors
- Also known as x86 family
  - First processor was 8086 (released 1978)
  - 16 bit processor; 29K transistors
- Intel continued to develop this series
  - 80186, 80286 – 16 bit; various addressing modes
  - 80386, 80486 – 32 bit; 486 integrated floating-point
  - Pentium series – instruction-set upgrades, optimizations
  - Pentium 4 – first introduction of 64 bit support
    - AMD developed x86-64 extensions first; Intel adopted them
    - P4 also introduced “hyper-threading” architecture: allows interleaved execution of two threads on one processor
  - Core 2 (multicore), Core i7 (multicore + hyper-threading)
- Backward-compatibility preserved throughout series
IA32 Registers

- IA32 has 8 general-purpose registers, and a wide variety of specialized registers
- `eax`, `ebx`, `ecx`, `edx`
  - General 32-bit registers for computations
- `esp = stack pointer`
  - Used with PUSH, POP, CALL, RET, etc.
- `ebp = base pointer`
  - For stack frame pointer
- `esi`, `edi`
  - Used for string load, move, store operations
Most operations support args of varying widths
- eax is 32 bits
- ax is low 16 bits of eax
- ah, al are high/low 8 bits of ax

Code written for 8086 thru 286 only accesses ax, ah, al
- Still available if necessary, but not used very often

Also 64-bit registers:
- rax, rbx, rcx, rdx
- rsp, rbp, rsi, rdi
IA32 Registers (3)

- Two other important registers:
  - **eip** = instruction pointer
    - 32-bits
    - Cannot access this register directly!
    - Modify eip using branching instructions
    - rip = 64-bit version
  - **eflags** = flags register
    - Many different flags
    - e.g. carry flag, zero flag, sign flag, overflow flag, direction flag
    - Cannot access/manipulate directly
    - Many operations for loading/saving/manipulating the flags register
IA32 Registers (4)

- Many other interesting registers
  - See IA32 manuals if you are curious!
  - Won’t use these registers for assignments this term

- Registers for segmented memory models
  - cs, ds, es, fs, gs, ss – all 16 bit

- Registers for floating-point arithmetic
  - 32-bit, 64-bit, 80-bit floating point values

- Registers for SIMD and MMX instructions
  - Single Instruction, Multiple Data – instructions for processing vectors of data very rapidly
  - MMX – more SIMD instructions for hardware media processing acceleration
IA32 and Word Size

- Word size in computing systems *usually* refers to unit of data processor is designed to work with
  - Can vary widely depending on system/application
- For IA32, word size defined to be 2 bytes (16 bits)
  - Original word size of 8086/8088
  - Even on 32/64-bit processors, word size is still 2 bytes
- Doubleword (dword) = 4 bytes (32 bits)
  - C *int* and *long int* data types are usually doublewords for IA32, gcc
- Quadword (qword) = 8 bytes (64 bits)
  - C *long long int* is a quadword for IA32, gcc
IA32 Words and Byte-Ordering

- For multibyte values, order of bytes in value becomes important
- Example: store value 0x12345678 in memory
  - Big endian: most significant byte at lowest address
    | Address | 0x100 | 0x101 | 0x102 | 0x103 |
    |---------|-------|-------|-------|-------|
    | Value   | 0x12  | 0x34  | 0x56  | 0x78  |
  - Little endian: least significant byte at lowest address
    | Address | 0x100 | 0x101 | 0x102 | 0x103 |
    |---------|-------|-------|-------|-------|
    | Value   | 0x78  | 0x56  | 0x34  | 0x12  |
- IA32 uses little-endian byte ordering
  - Can make it confusing to look at memory dumps 😐
  - Address of multibyte value is address of lowest byte
WORDS AND POINTERS

- Word size has an important impact on system!
  - Directly affects how much memory the system can access
- For IA32 (and x86-64):
  - 32-bit processors can access up to $2^{32}$ bytes (4 GB)
  - 64-bit processors can access much more memory
    - Currently can address up to $2^{48}$ bytes (256 TB)
  - Other hardware may impose greater restrictions
    - e.g. motherboard supports 64-bit processor, only 16GB RAM
- Pointer:
  - The address or location of a value in main memory
  - Pointers also have a type, which specifies number of bytes that the value occupies (among other things)
IA32 Instructions

- Instructions follow this pattern:
  - *opcode* *operand, operand, ...*

- Examples:
  - `add %ax, $5`
  - `mov %ecx, %edx`
  - `push %ebp`

- Important note!
  - Above assembly-code syntax is called *AT&T syntax*
  - GNU assembler uses this syntax by default
  - Intel IA32 manuals, other assemblers use *Intel syntax*

- Some big differences between the two formats!
  - `mov %ecx, %edx` # AT&T: Copies ecx to edx
  - `mov edx, ecx` # Intel: Copies ecx to edx
**IA32 Instructions (2)**

- Some general categories of instructions:
  - Data movement instructions
  - Arithmetic and logical instructions
  - Flow-control instructions
  - (many others too, e.g. floating point, SIMD, etc.)

- Next time:
  - Dive into the details of IA32 instruction set
  - Examine how to integrate C and IA32 programs