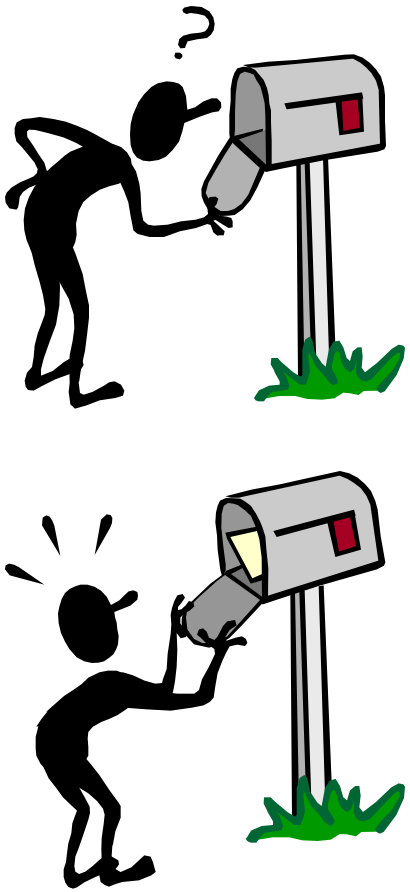
Addressing Modes and Other ISAs



- Where is the data?
- Addresses as data
- Names and Values
- Indirection

Assembly Exercise

- Let's write some assembly language programs

- Program #1: Write a function "isodd(int X)" which returns 1 if it's argument "X" is odd and 0 otherwise

```
main:   addiu   $a0,$0,37
        jal     isodd
        addiu   $a0,$0,42
        jal     isodd
halt:   beq     $0,$0,halt

isodd:  andi    $v0,$a0,1
        jr      $31
```

The addiu instruction is used to load constants (i.e. isodd(37)), can this be done in other ways?

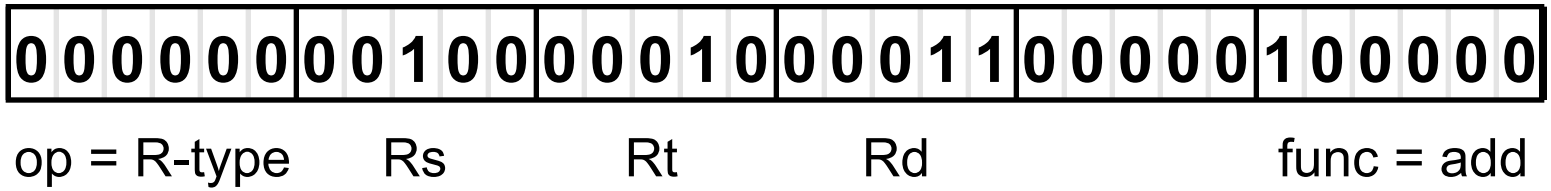
The function is implemented using only one instruction. How does "andi $Y,$X,1" determine that $X is odd?

Your Turn

- Program #2: A function "ones(int X)" that returns a count of the number of ones in its argument "X"

The MIPS ISA

32-bit (4-byte) ADD instruction:



Means, to MIPS, Reg[3] = Reg[4] + Reg[2]

But, most of us would prefer to write

```
add $3, $4, $2
```
(ASSEMBLER)

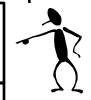
or, better yet,

```
a = b + c;
```
(C)

Journey of a One-Address Machine

- Once, 1974, there was an heroic effort to build a single-chip computer on a budget of 3500 devices, by a Silicon Valley startup called Intel, that could sell for < $200.
- It had one primary 8-bit register, A, for accumulator, and 6 other 8-bit registers, B, C, D, E, H, and L with more limited and special functions. Certain register pairs could be linked to act as 16-bit registers, BC, DE, and HL, mostly for addressing 65536 bytes of memory.
- It had one other register, SP, which was always 16-bits and was used to implement a stack.

The FLAG register tracks ancillary results of the previous accumulator operation. For example, was there a carry out of the MSB? is the result zero? is it negative?



A (accumulator)	Flags
B	C
D	E
H	L
SP	

Journey of a One-Address Machine

- Typical instructions

ADD s	A ← A + s	where d, s one of {A,B,C,D,E,H,L,Mem[HL]}
SUB s	A ← A – s	
ANA s	A ← A & s	
ORA s	A ← A | s	
XRA s	A ← A ^ s	
CMP s	A – s	Note: Sets flags only
INR d	d ← d + 1	
DCR d	d ← d – 1	

Notice how most instructions include only one operand d or s. This is possible, because the other operand and the destination are implicit. They are either both the accumulator, A, or the operand specifies both source and destination and the other operand is a constant. Instructions with one operand are called one-address machines.

- A really cool one:

MOV d, s	d ← s

A (accumulator)	Flags
B	C
D	E
H	L
SP	

Journey of a One-Address Machine

- Immediate Operands

ADI imm_8	A ← A + imm_8
SUI imm_8	A ← A – imm_8
ANI imm_8	A ← A & imm_8
ORI imm_8	A ← A | imm_8
XRI imm_8	A ← A ^ imm_8
CPI imm_8	A – imm_8
MVI d,imm_8	d ← imm_8

- No operands

RAL	a ← a << 1
RAR	a ← a >> 1
XCHG	HL ←→ DE

- Register Pair Operands

INX p	p ← p + 1
DCX p	p ← p – 1
DAD p	HL ← HL + p
LXI p,imm_{16}	p ← imm_{16}

where p is one of {BC, DE, HL, or SP}

A (accumulator)	Flags
B	C
D	E
H	L
SP	

Journey of a One-Address Machine

- Fancy Memory Reference (recall s and d can be Mem[HL])

LDA $addr_{16}$	A ← Mem[$addr_{16}$]	
STA $addr_{16}$	A → Mem[$addr_{16}$]	
LHLD $addr_{16}$	HL ← Mem[$addr_{16}$]	
SHLD $addr_{16}$	HL → Mem[$addr_{16}$]	
LDAX p	A ← Mem[p]	(p is one of {BC, DE})
STAX p	A → Mem[p]	
XTHL	HL ←→ Mem[SP]	

in the following p is one of

{A | flags, BC, DE, or HL}

PUSH p
Mem[SP-1] ← p_L;
Mem[SP-2] ← p_H;
SP ← SP – 2

POP p
p_H ← Mem[SP];
p_L ← Mem[SP+1];
SP ← SP + 2

A (accumulator)	Flags
B	C
D	E
H	L
SP	

Journey of a One-Address Machine

- Branch and control

JMP $addr_{16}$	PC ← Mem[$addr_{16}$]
JNZ $addr_{16}$	if Flags.Z = 0 PC ← Mem[$addr_{16}$]
JZ $addr_{16}$	if Flags.Z = 1 PC ← Mem[$addr_{16}$]
JNC $addr_{16}$	if Flags.C = 0 PC ← Mem[$addr_{16}$]
JC $addr_{16}$	if Flags.C = 1 PC ← Mem[$addr_{16}$]

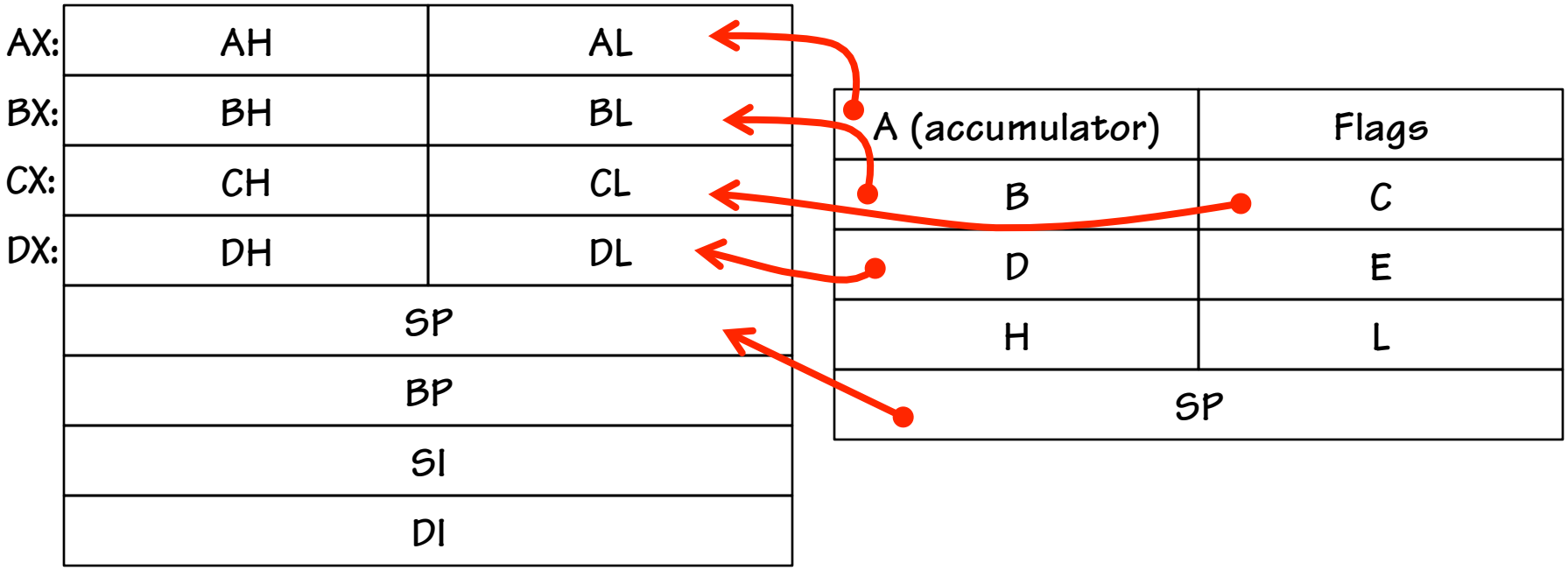
CALL $addr_{16}$
CNZ $addr_{16}$
CZ $addr_{16}$
CNC $addr_{16}$
CC $addr_{16}$

CALL instructions behave similar to their corresponding JMP instructions. However, they also do the following:
MEM[SP] ← PC; SP ← SP -2

A (accumulator)	Flags
B	C
D	E
H	L
SP	

Growing to a Two-Address Machine

- Intel's 8080 was a huge success, but it soon started to exhibit growing pains that limited its usefulness in comparison to *minicomputers* of the same era.
- Key Problems:
  - 8-bit registers were too small
  - The maximum of addressable 65536 bytes was too limiting
- So it grew into the 8086, 16-bit architecture (1978)



Growing to a Two-Address Machine

- This change came with more innovations
  - Assembly-language/mnemonic compatibility
  - Making the AX, BX, CX, and DX more general purpose (i.e. all could be used like accumulators)
- Similar, but different two-operand instructions:

I am intentionally ignoring lots of crazy instructions like "AAA- Add with an ASCII Adjust," for adding digits encoded in ASCII

AX:	AH	AL
BX:	BH	BL
CX:	CH	CL
DX:	DH	DL
	SP	
	BP	
	SI	
	DI	

ADD d, s d ← d + s
SUB d, s d ←d – s
AND d, s d ←d & s
OR d, s d ← d | s
XOR d, s d ← d ^ s
MOV d,s d ←s

s is {AX,BX,CX,DX,imm_{16},sext(Imm_8), $Mem[addr_{16}]$, Mem[SI], $Mem[SI+addr_{16}]$, Mem[BP+SI]}

d is {AX,BX,CX,DX, $Mem[addr_{16}]$, Mem[DI], $Mem[DI+addr_{16}]$, Mem[BP+DI]}

Growing to a Two-Address Machine

- There was one more major addition
  - 4 segment registers extend addresses to 20-bits (1048576 bytes)
  - Technically, addresses are to any of 65536 bytes offset from a 16-byte aligned segment

Mem[X] = Mem[((segment register)<<4) + X] where the segment register varies depending on the "type" of access.

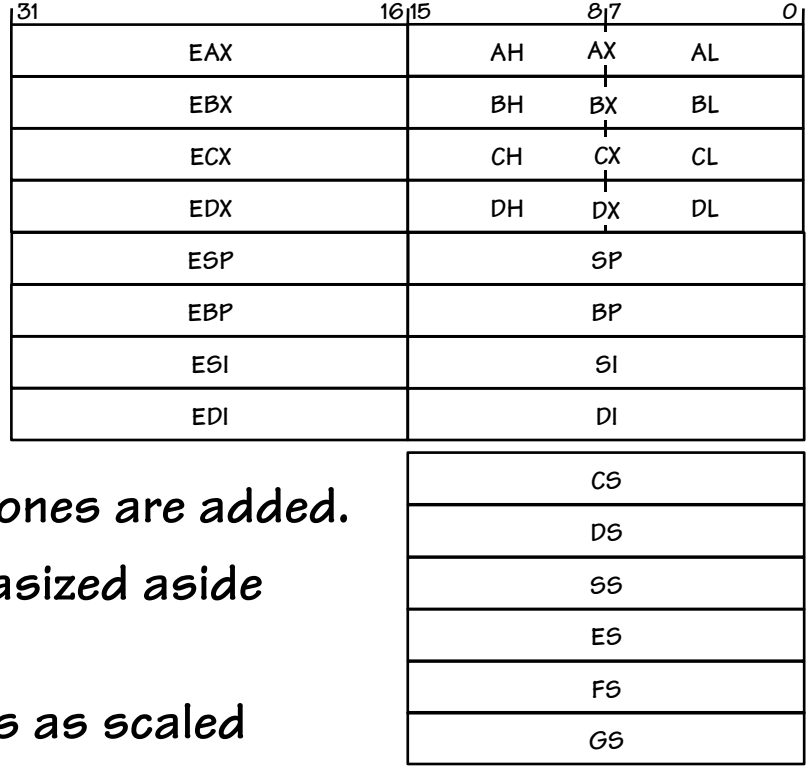
AX:	AH	AL
BX:	BH	BL
CX:	CH	CL
DX:	DH	DL
	SP	
	BP	
	SI	
	DI	

Memory accesses for instructions (involving the PC) use CS (code segment), for data use DS (data segment) or ES (extra segment), and for stack operations (involving SP) use SS (stack segment)

CS
DS
SS
ES

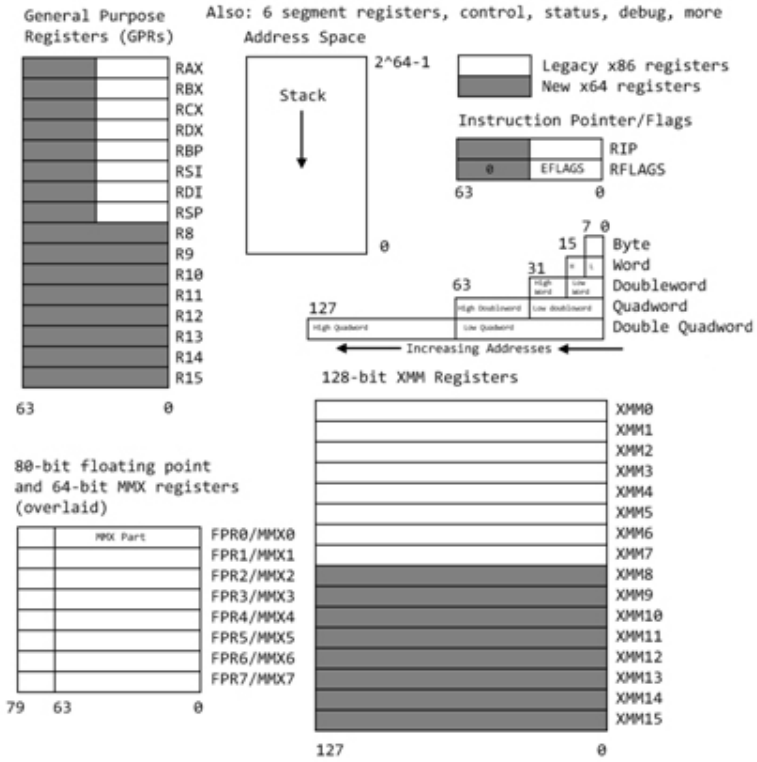
More Growing Pains (x86)

- Once again the 8088, 8086, 80186, and 80286 machines were immensely successful (IBM PC, Compaq, Clones), and, once again, they started to show limitations compared to a new class of machines called workstations.
- Intel introduced the i386 and i486 extensions
  - Registers expanded to 32-bits
  - Address space non-segmented, "flat" 4,294,967,296 bytes
- Segment registers
  - Not extended to 32-bits. Two new ones are added.
  - Segmented addressing is deemphasized aside from backward compatibility
  - Take on more general-purpose roles as scaled offsets (eg. Mem[SI+(DS<<4)])



Success is so Painful!

- The i386 and i486 are once again successful, even amongst simpler, faster, and cheaper RISC CPUs. Intel manages to adapt again
- In 1995 Intel introduces their first pipelined version of the x86 ISA (Pentium) adapting many RISC concepts and adding a few new ones.
- Nonetheless, x86 starts to feel growing pains again as 32-bit architectures run out of space for code and data



- Around 2005, Intel introduces a 64-bit Pentium D, Core 2, Core i3, i5, and i7 archetectures

Lessons Learned

- Instruction Set Architectures would rather Evolve than be Reinvented!
  - While developing new CPU hardware is expensive, it pales in comparison to developing software. Thus, maintaining backward-compatibility has been one of Intel's secrets of success.

- Beauty does not equal Truth!
  - Just because instructions are ugly does not mean that they can't accomplish the task. Conversely, if a task gets done, no one cares if it was done in an ugly or beautiful manner
- Birds in the hand are unlikely to become jet liners!
  - It is always better to apply your cleverness to making existing customers happier with what they've already done, than trying to convince them they should start over doing it a better way.

Revisiting 1,2, and 3 Operands

- Operands – the variables needed to perform an instruction's operation
- Two types in Intel's history
  - One address: ADD C # A ← A + C
  - Two address: ADD CX,DX # CX ← CX + DX
- Three types in the MIPS ISA:
  - Three Address (Registers only!):

    ```
    add $2, $3, $4      # operands are the "Contents" of a register
    ```

  - Immediate:

    ```
    addi $2,$2,1        # 2nd source operand is part of the instruction
    ```

  - Register-Indirect:

    ```
    lw  $2, 12($28)     # source operand is in memory
    sw $2, 12($28)      # destination operand is memory
    ```

- Simple, but is it enough?

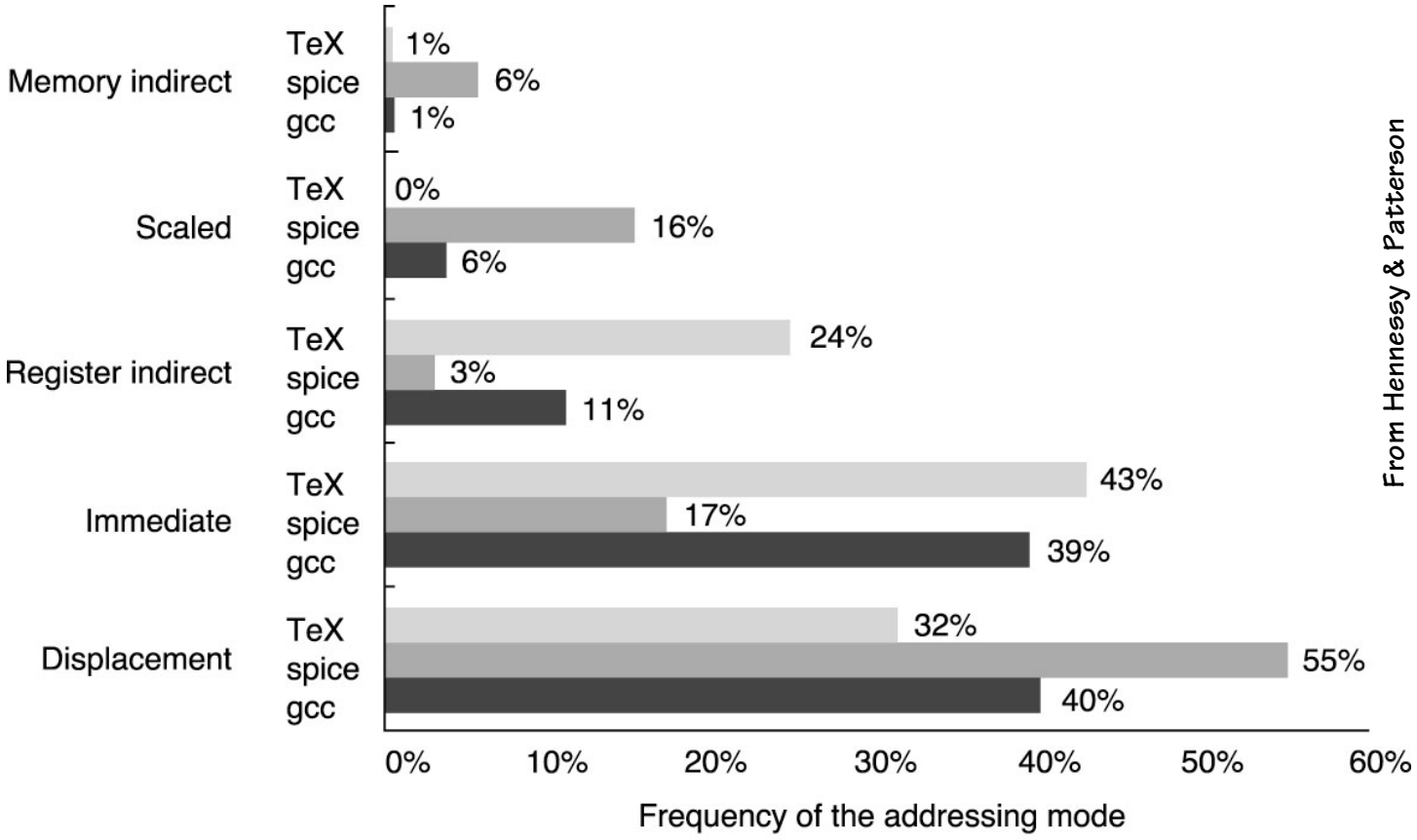
Common "Addressing Modes"

MIPS can do these with appropriate choices for Ra and const

- **Absolute (Direct):** lw $8, 1000($0)
  - Value = Mem[constant]
  - Use: accessing static data
- **Indirect:** lw $8 0($9)
  - Value = Mem[Reg[x]]
  - Use: pointer accesses
- **Displacement:** lw $8, 16($9)
  - Value = Mem[Reg[x] + constant]
  - Use: access to local variables
- **Indexed**:
  - Value = Mem[Reg[x] + Reg[y]]
  - Use: array accesses (base+index)
- **Memory indirect**:
  - Value = Mem[Mem[Reg[x]]]
  - Use: access thru pointer in mem
- **Autoincrement**:
  - Value = Mem[Reg[x]]; Reg[x]++
  - Use: sequential pointer accesses
- **Autodecrement**:
  - Value = Reg[X]--; Mem[Reg[x]]
  - Use: stack operations
- **Scaled**:
  - Value = Mem[Reg[x] + c + d*Reg[y]]
  - Use: array accesses (base+index)

Argh! Is the complexity worth the cost?
Need a cost/benefit analysis!

Memory Operands: Usage



Usage of different memory operand modes

Real-World Addressing

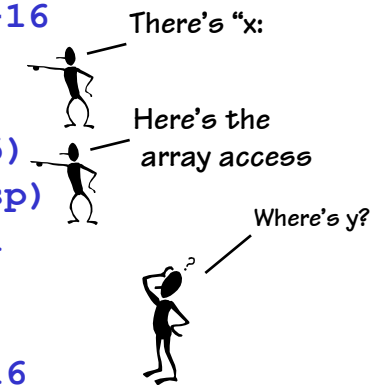
- What we want:
  - In general, the contents of a specific memory location
- How we get it? Let's look at high-level constructs!
- Examples:

"C"

```
int x = 5;
int data[10];
main() {
    int y;
    v = data[x];
    x = x + 1;
}
```

"MIPS Assembly"

```
main: addiu $sp,$sp,-16
      lw $24,x
      sll $15,$24,2
      lw $15,data($15)
      sw $15,-4+16($sp)
      addiu $24,$24,1
      sw $24,x
      move $2,$0
      addiu $sp,$sp,16
      jr $31
x:    .word 0x5
data: .space 10
```



- Caveats
  - In practice $gp is often used as a base address for all variables
  - Can only address the first and last 32K of memory this way
  - Sometimes generates a two instruction sequence:

```
lui  $1,xhighbits
lw   $2,xlowbits($1)
```

Next Time

- More about how "C"
  - How and where does it allocate variables?
  - How are common high-level constructs converted to assembly language?
    - if () { } else{ };
    - for (;;) { }
    - while () { }
    - do { } while ();
    - +=, ++, -=, &, &&
    - myfunc(arg1, arg2)