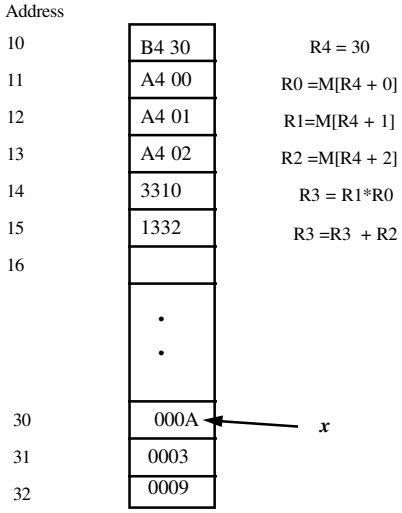
## Program fragment to evaluate 3x + 9 at x = 10

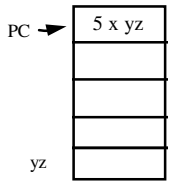





## Program Control and Looping

Uses two Format 3 instructions:

```
5   x   yz
```
stands for
```
JUMP  yz
```
which means
```
PC  =  yz
```
(note: x is ignored)




```
6  x yz
```
stands for
```
JUMP<0 Rx  yz
```
which means
```
if (Rx < 0)
  PC = yz
```

## Loop to compute the sum of the numbers from 1 to n

In C:

```
sum =0;
for( ; n>=0; n=n-1)
 {
   sum = sum + i;
  }
```

In TOY Code: (assume n is stored in R2 initially)

| Address | |
|---|---|
| 10 | R0 = 0 |
| 11 | R1 = 1 |
| 12 | JUMP$_{<0}$ R2 16 |
| 13 | R0 = R0 + R2 |
| 14 | R2 = R2 – R1 |
| 15 | JUMP 12 |
| 16 | |



## Implementing Arrays in TOY

C code:

```
{
   int  i =5, A[10];
   A[i] = A[i+1] + A[i+2];
}
```

TOY code:

| | |
|---|---|
| 10 | R1 = 5 |
| 11 | R2 = 30 |
| 12 | R3 = R2 + R1 |
| 13 | R4 = M[R3 +1] |
| 14 | R5 = M[R3 +2] |
| 15 | R6= R4 + R5 |
| 16 | M[R3 +0] = R6 |
| | . |
| | . |
| 30 | |
| 31 | |
| 32 | |

*Array A Begins here*

## Implementing Functions in TOY

Uses the following format 3 instructions:

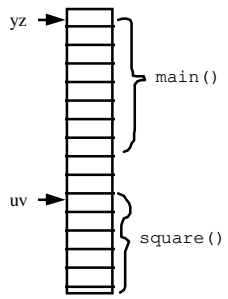
1. `7 x yz`
   means
   `Jump-Indirect Rx yz`
   which stands for
   `PC = Rx`
   (note: yz is ignored)

2. `8 x yz`
   means
   `Jump-and-Link Rx yz`
   which stands for
   `Rx = PC`
   `PC = yz`



## Implementing Functions (contd.)

```
int square(int)
void main()
{ int a=10, b=5;
  a =square(b);
  b =square(a);
}
int square(int b)
{
 return b*b;
}
```




Function Call:
- `main()` stores argument for `square()` in R1
- `main()` executes `JUMP-and-LINK 0 uv`
- `square()` stores R0 somewhere

Function Return:
- `square()` stores its returned value in R1
- `square()` restores stored value of R0
- `square()` executes JUMP-INDIRECT R0 to return control to main( )

***Many other subtleties !***

## Example (contd.)

```
10   R3 =10
11   R4 = 5
12   R2 = 0
13   R1 =R4 + R2
14   JUMP-AND-LINK R0 uv
15   R3 = R1
16   R1 = R3
17   JUMP-AND-LINK R0 30
18   R4 = R1


30   R1 = R1*R1
     JUMP-INDIRECT R0
```



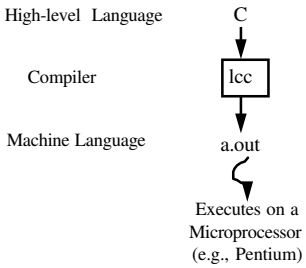
## TOY v/s Real Machines

Real machines:

- VLSI Chip on a board
- Timing issues (registers are faster than memory)
- Richer set of I/O functions
- Much more memory
  - billion of bytes, not 256
  - 32 (or 64) bits per word, not 16
- More kinds of built-in data formats
  e.g., floating point
- More kinds of instructions
  e.g., division, floating point ops

**But:** Given enough memory, TOY can simulate any one of these, even with its limited instruction set.

## Multi-level design




Advantages:

- Cheap and Fast !
  A microprocessor that can "interpret" C would be very expensive and slow.
- Allows Flexibility and Portability
  Want to switch from C to PASCAL?
  Want to switch to a different microprocessor?
  Just use a different compiler!



## Simulation

Any "general purpose" computer can simulate any other computer.

*Example:* Simulating TOY in C

- represent memory and registers as arrays

```
int  memory[256], regs[8];
```

- "loading programs and data" involves a `for` loop that uses `scanf()` statement
- interpreting instructions involves a `switch` statement with 16 cases

```
opcode = (inst >> 12) & 0x7;
```

/u/cs126/simulator contains the program

# How a program can handle positive integers containing 320 bits

Main Idea:

- Use arrays

  `int d[10];`

  The integer represented by `d[]` is

  $$d[0] + 2^{32} d[1] + 2^{64} d[2] + \ldots + 2^{288} d[9]$$

- Define functions that do arithmetic operations

```
int add(int a[], int b[], int c[])
/* adds integers represented by
   b[] and c[]; stores the sum
   in a[] */
{
  Just the grade-school method
 for addition
}
```