# TeenyAT

/ˈtē·nē·ət/

An Architecture to Virtualize

# Core Architectural Elements

- 16 bit words (each memory address points to a 16 bit value)
- 32K words of RAM from 0x0000 through 0x7FFF
- Address 0x8000: character output to terminal
- Address 0x8001: character input from terminal
- 8 registers available to all register-use instructions
  - pc (or r0) is the program counter, contains address of next instruction, initially 0x0000
  - r1 through r6 are general purpose registers
  - sp (or r7) is the stack pointer, contains address below top element, initially 0x8000 (empty)
- Each instruction is encoded in two 16 bit words
  - Fetch must increment the PC by 2 to account for this

# Instruction Set

Arithmetic (dreg,sreg)

- **add**
- **sub**
- **mult**
- **div**
- **mod**
- **neg** reg
- **inc** reg
- **dec** reg

Logic (dreg,sreg)

- **and**
- **or**
- **xor**
- **inv** reg
- **shl** dreg, imm
- **shr** dreg, imm

Data

- **set** dreg, imm
- **copy** dreg, sreg
- **load** dreg, addr
- **stor** addr, sreg
- **pload** dreg, preg
- **pstor** preg, sreg
- **push** sreg
- **pop** dreg

Control

- **call** addr
- **ret***
- **jmp** addr*
- **jl** reg1, reg2, addr
- **jle** reg1, reg2, addr
- **je** reg1, reg2, addr
- **jne** reg1, reg2, addr
- **jge** reg1, reg2, addr
- **jg** reg1, reg2, addr

* **Pseudoinstructions**: These instructions can be synthesized with a single different instruction

# Logic Instruction Details

- and rA, rB  bitwise ANDs contents of rA and rB, storing result in rA
- or rA, rB  bitwise ORs contents of rA and rB, storing result in rA
- xor rA, rB  bitwise XORs contents of rA and rB, storing result in rA
- inv rA  flip all bits of rA, storing result in rA
- shl rA, imm  bitwise left shift of rA by imm bits, storing result in rA
  - for imm < 0 or imm >= 16, rA will be 0x0000
- shr rA, imm  bitwise right shift of rA by imm bits, storing result in rA
  - for imm < 0 or imm >= 16, rA will be 0x0000

# Arithmetic Instruction Details

- add rA, rB — adds contents of rA and rB, storing result in rA
- sub rA, rB — subtracts contents of rA and rB, storing result in rA
- mult rA, rB — multiplies contents of rA and rB, storing result in rA
- div rA, rB — divides contents of rA and rB, storing quotient in rA
- mod rA, rB — divides contents of rA and rB, storing remainder in rA
- neg rA — negates contents of rA, storing remainder in rA
- inc rA — increments contents of rA, storing remainder in rA
- dec rA — decrements contents of rA, storing remainder in rA

# Data Instruction Details

- set rA, imm  sets contents of rA to imm
- copy rA, rB  sets contents of rA  to the same as rB
- load rA, addr sets contents rA to the contents of memory[addr]
- stor addr, rA  sets the contents of memory[addr] to rA
- pload rA, rB  sets contents rA to the contents of memory[rB]
- pstor rA, rB  sets the contents of memory[rA] to rB
- push rA  sets contents of rA to top of stack and decrements SP
- pop rA  sets contents of rA from the top of the stack and increments SP

# Control Instruction Details

- call addr  pushes next PC to stack and jumps to code at memory[addr]
- ret*  pops PC from stack to return from prior call
  - same as "pop pc"
- jmp addr*  sets PC to addr
  - same as either "set pc, addr" or "je r0, r0, addr"

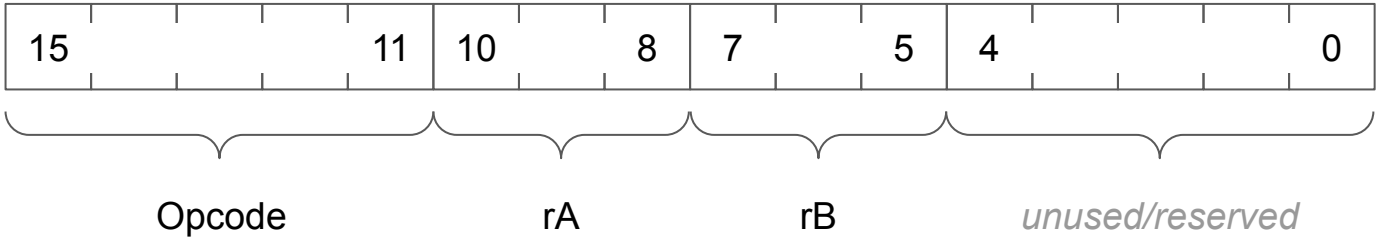
* **Pseudoinstructions**: These instructions can be synthesized with a single different instruction

# Control Instruction Details (continued)

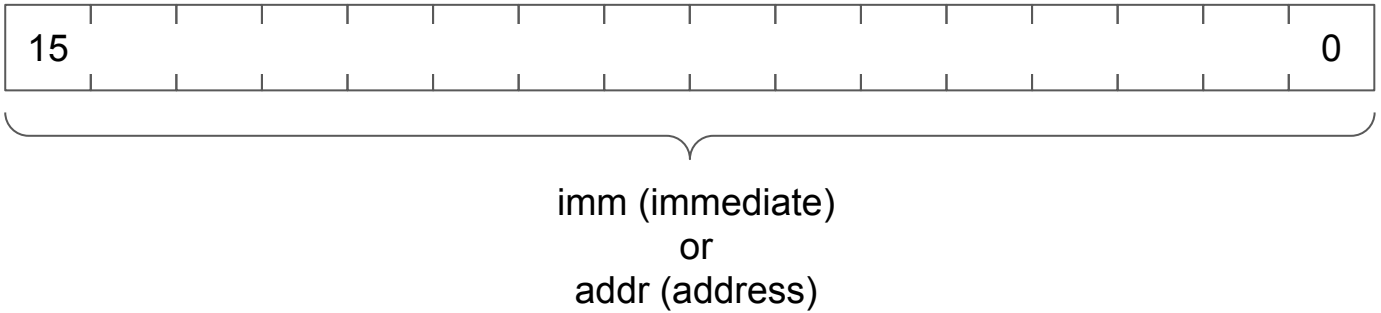
- jl rA, rB, addr  sets PC to addr if rA < rB
- jle rA, rB, addr  sets PC to addr if rA <= rB
- je rA, rB, addr  sets PC to addr if rA == rB
- jne rA, rB, addr  sets PC to addr if rA != rB
- jge rA, rB, addr  sets PC to addr if rA >= rB
- jg rA, rB, addr  sets PC to addr if rA > rB

# Instruction Encodings

**1st instruction word**




**2nd instruction word**



# Instruction Opcodes

| Instr. | Opcode |
|---|---|
| set | 0 |
| copy | 1 |
| load | 2 |
| stor | 3 |
| pload | 4 |
| pstor | 5 |
| push | 6 |
| pop | 7 |
| add | 8 |
| sub | 9 |

| Instr. | Opcode |
|---|---|
| mult | 10 |
| div | 11 |
| mod | 12 |
| neg | 13 |
| inc | 14 |
| dec | 15 |
| and | 16 |
| or | 17 |
| xor | 18 |
| inv | 19 |

| Instr. | Opcode |
|---|---|
| shl | 20 |
| shr | 21 |
| call | 22 |
| jl | 23 |
| jle | 24 |
| je | 25 |
| jne | 26 |
| jge | 27 |
| jg | 28 |
| | |

**Fun fact:** If all words of memory are initialized to 0x0000, then if-ever a program should try to run code beyond that loaded into memory, the fetched instruction will be 0x0000_0000, which decodes into "set pc, 0x0000"... so the program will start over automatically since the first line of code is at 0x0000.