2019

Systems

Notes

Murphy
UNIVERSITY OF GLASGOW

CONTENTS

CS1S Systems

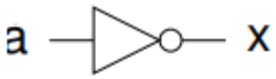
1 Circuits

1.1 Digital Circuits

- Digital circuits are built from a very small number of primitive machines
- Only a few are needed:
  - Logic Gates:
    - inv
    - and2
    - or2
    - xor2
  - Flip flop:
    - dff

1.2 Inverter

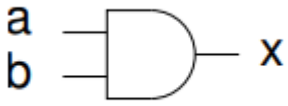
- The inverter is a basic component
- Takes an input bit **a** and produces an output bit **x**
- The output is the logical opposite of the input
- *x = **inv** a*



a	x
0	1
1	0

1.3 And2

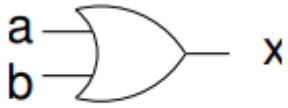
- The 2-input and gate is a basic component
- Takes inputs **a** and **b** and produces an output bit **x**
- The output is 1 if all inputs are 1
- *x = **and2** a b*



a	b	x
0	0	0
0	1	0
1	0	0
1	1	1

1.4 OR2

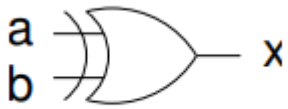
- The 2-input inclusive or gate is a basic component
- Takes inputs **a** and **b** and produces an output bit **x**
- The output is 1 if any input is 1
- *x = **or2** a b*



a	b	x
0	0	0
0	1	1
1	0	1
1	1	1

1.5 XOR2

- The 2-input exclusive or gate is a basic component
- Takes inputs **a** and **b** and produces an output bit **x**
- The output is 1 if either input is 1, but not both
- *x = **xor2** a b*



a	b	x
0	0	0
0	1	1
1	0	1
1	1	0

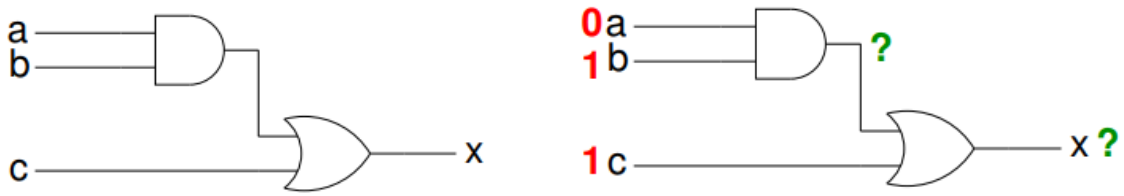
1.6 GATE DELAY

- If the inputs to a logic gate remain stable, the output will remain stable
- If an input changes at a point in time, it will take the logic gate a small amount of time to bring the output to the new value.
- Gate delays are on the order of *0.01ns* and may be faster or slower depending on the technology

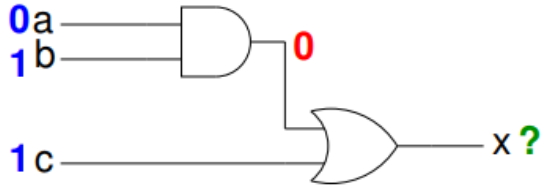
1.7 COMBINATIONAL CIRCUITS

- A combinational circuit consists of logic gates connected together
- The circuit has inputs and produces outputs
- It contains no feedback loops
- The outputs depend on the current input values (the circuit has no memory)

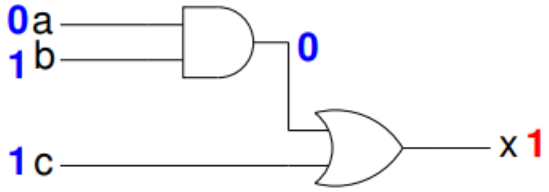
1.8 Circuit Simulation



1.8.1 After one gate delay



1.8.2 After two gate delays



1.9 Building Block Circuits

- To do anything useful a circuit usually needs to be big, but it's hard to understand or design it
- The solution is **abstraction**:
  - Put several components together to build a *black box circuit*
  - Each black box circuit will combine to make a bigger one, and so on…

1.10 Operators in Boolean Algebra

- These are the same as the definitions of the logic gates (except we don't have a Boolean operator for the XOR gate)

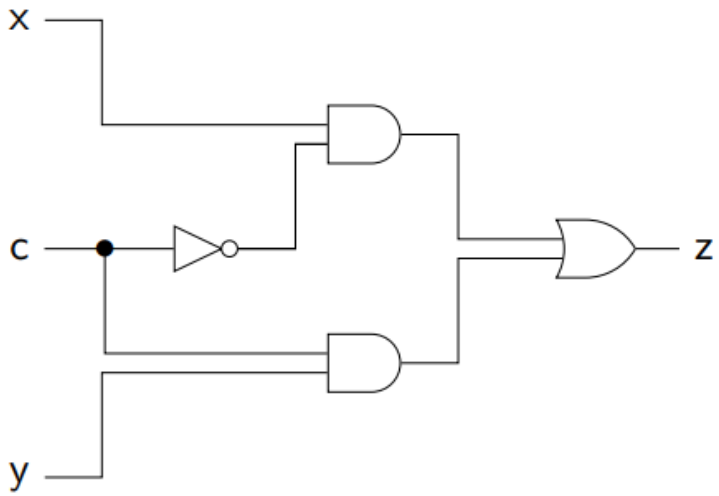
x	$\neg x$
0	1
1	0

x	y	$x \wedge y$	$x \vee y$
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

Algebra	Name	Circuit
$\neg x$	not	inv x
$x \vee y$	or	or2 x y
$x \wedge y$	and	and2 x y

1.11 MULTIPLEXER

- The **mux1** is the fundamental circuit used to make decisions
- A multiplexer is a hardware version of the **if-then-else** expression
- The idea is to choose between two value **(x, y)** based on another value **(c)**
    - Given three inputs: **c, x, y**
    - Produce one output
        - If **c** = 0 the output is **x**
        - If **c** = 1 the output is **y**
- In digital hardware circuits, there are no statements
    - So, there are no **if statements** to determine whether other statements should be executed
    - Instead, we have signals whose values depend on other signals
- ***z** = (if **c**=0 then **x** else **y**)*



c	x	y	out
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

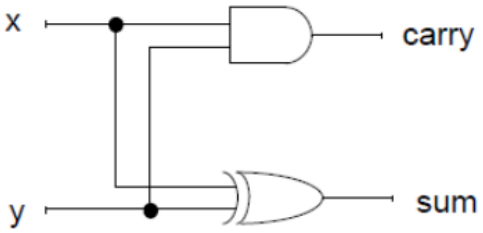
→

c	x	y	out
0	**0**	0	**0**
0	**0**	1	**0**
0	**1**	0	**1**
0	**1**	1	**1**
1	0	**0**	**0**
1	0	**1**	**1**
1	1	**0**	**0**
1	1	**1**	**1**

- Multiplexers are a central black box which we will use again and again
- One way to understand the circuit is to calculate its truth table by simulating it for each set of input values
- Most conditional operations in digital circuits are implemented at the lowest level by a multiplexer
- In a circuit we define signals using **if-then-else** expressions implemented with multiplexers

1.12 The Half Adder

- A half adder adds two bits. Since the result could be 0, 1, or 2, we need a two-bit representation of the sum, called the (carry, sum)
- The carry function is just **and2**, and the sum function is just **xor2**

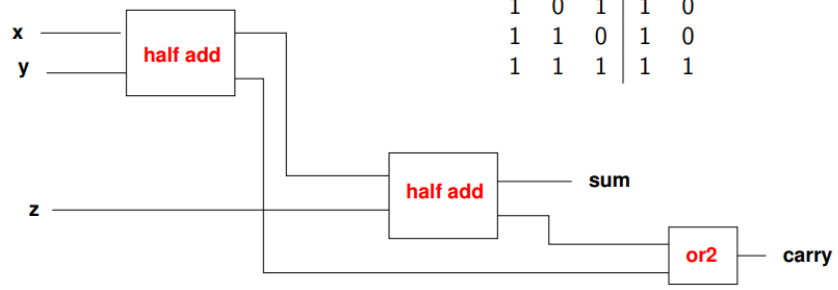


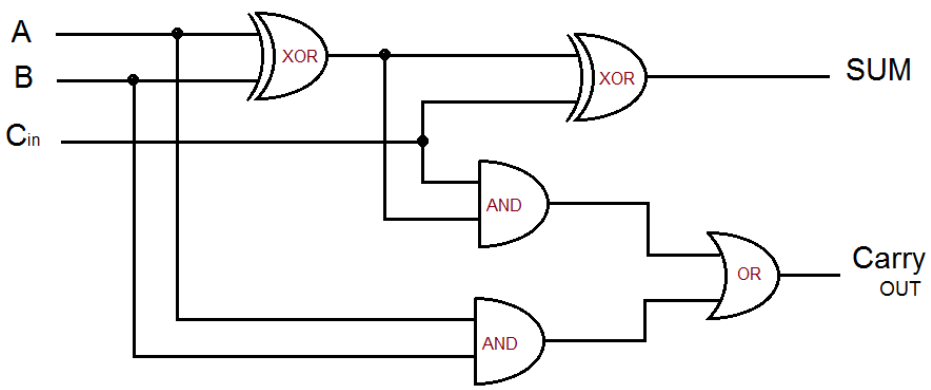
x	y	result	carry	sum
0	0	0	0	0
0	1	1	0	1
1	0	1	0	1
1	1	2	1	0

1.13 The Full Adder

- To add two binary numbers, we must add **three** bits for each column:
  - The two data bits (one from each word)
  - The carry input from the column to the left
- A full adder adds three bits **x, y, z** and outputs a carry **c** and sum **s**

x	y	z	c	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1





1.14 Boolean Algebra

- Operations with constants:

$$\begin{aligned} x \wedge 0 &= 0 \\ x \wedge 1 &= x \\ x \vee 0 &= x \\ x \vee 1 &= 1 \end{aligned}$$

- These help to work out the values of signals in a circuit

1.14.1 Idempotence

- In general, an operation is idempotent if doing it several times is the same as doing it once
- Here is the mathematical definition: $x \vee x = x$, $x \wedge x = x$

1.14.2 Commutativity

- You can swap around the order of inputs to an **and** gate or an **or** gate without changing the result: $x \vee y = y \vee x$, $x \wedge y = y \wedge x$

1.14.3 Associativity

- This gives a way to compute the logical **and**/**or** of many inputs, by using several 2-input **and**/**or** gates: $x \vee (y \vee z) = (x \vee y) \vee z$, $x \wedge (y \wedge z) = (x \wedge y) \wedge z$

- These laws are crucial in:
  - High performance circuit design
  - Programming massively parallel high-performance computers

1.15 Equational Reasoning

- Equational reasoning means "substituting equals for equals" using the laws of algebra

Problem: Use Boolean algebra to simplify $(0 \wedge P) \vee Q$

Each of these steps uses one of the laws:

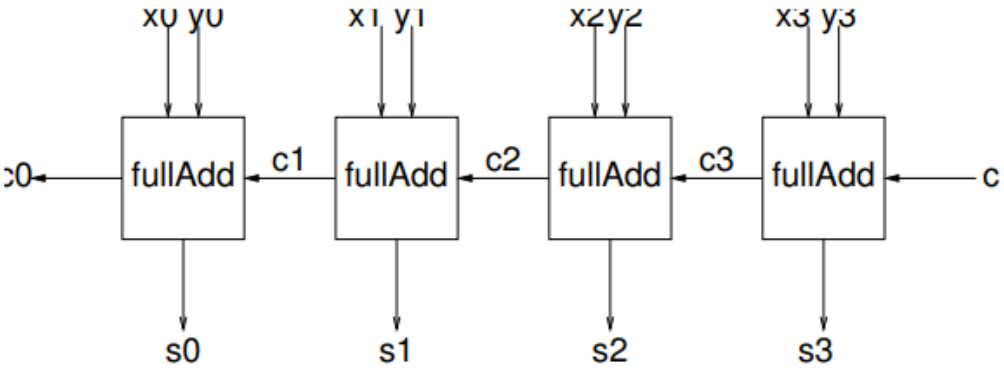
$$\begin{aligned} & (0 \wedge P) \vee Q \\ = \; & (P \wedge 0) \vee Q && \wedge \text{ is commutative} \\ = \; & 0 \vee Q && \text{constant operation on } \wedge \\ = \; & Q \vee 0 && \vee \text{ is commutative} \\ = \; & Q && \text{constant operation on } \vee \end{aligned}$$

1.16 Addition and Subtraction

- Addition, subtraction, and negation are all done by one circuit: **rippleAdd**, the "ripple carry adder"
- We will proceed in stages:
  - Adding two bits: **halfAdd**
  - Adding three bits: **fullAdd**
  - Adding two integers: **rippleAdd**
  - Subtracting an integer from another

1.17 4-Bit Ripple Carry Adder

- Inputs: word **x**, word **y,** carry input bit **c**
- Outputs: sum **s**, carry output
- Each bit position receives a bit from **x**, a bit from **y**, and a carry input from the position to the right
- A full adder circuit adds these three bits



1.18 Subtraction

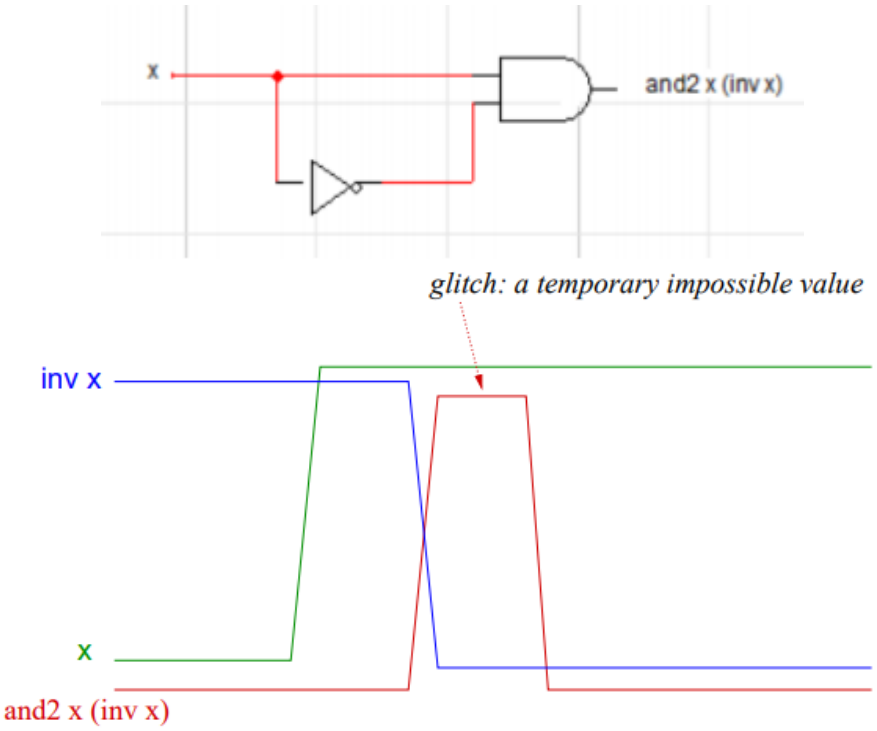
- Work with two's complement numbers
- Note that x - y = x + (-y)
- Recall that to negate a number, you invert the bits and add 1
- To invert the bits of **y**, just put an inverter on each bit of **y**
- To add 1, just set the carry input to the entire ripple carry adder to 1
- Use a ripple carry adder, with each bit of **y** inverted, and with carry input = 1, and the output will be **x** – **y**

1.19 Hazards

- Strange effects can occur when a circuit's inputs become stable at different times (which is common). According to Boolean algebra, we can transform this mathematically to: ***y*** = ***zero***
- However, if we draw a graph showing the signal values as a function of time, we see that when x changes from 0 to 1, the output of ***inv x*** changes from 1 to 0
- As a result, both inputs to the ***and2*** gate are both 1 for a little while, and the output will be 1 momentarily
- The hardware does not always obey the laws of Boolean algebra

1.20 GLITCH

- For a short time, the output of the circuit is incorrect, but if you wait a short time, it will reach the correct value
- For example, ***and2 x (inv x) = 1*** momentarily, which is incorrect



1.21 STATE

- State means something like "memory"; the values of all the memory locations and all the registers are a point in time
- A circuit consisting only of logic gates is called **combinational**
- A combinational circuit has no state: its output depends on its input (after gate delays)
- We use the **dff** and **clock** to:
  - Have a way to hold a state – a component that can remember a bit
  - Keep the circuit synchronized – the gate delays in different parts of a circuit are likely to be different, and this could lead to confusion

1.22 THE DELAY FLIP FLOP (DFF)

- A dff is a circuit that remembers one bit of data (the "state")
- There is a data input **x** carrying a value to be remembered, and a data output **y** conveying the state values
- There is also a clock input which ticks regularly

1.23 The Clock

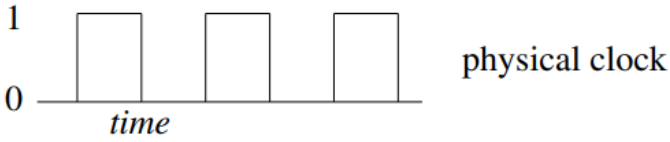
- The dff component copies its input value into its state at specific points in time, determined by a clock signal. The behaviour is:
  - To output the state continuously
  - To execute ***state*** *:=* ***input*** whenever a clock tick occurs
- There is just one clock signal, which is sent to all flip flops – so every flip flop updates its state at the same time
- The clock is generated externally by a circuit that produces a fast and steady sequence of clock ticks

1.23.1 Abstract Clock

- Abstractly, the clock is always 0, except at a regular sequence of instantaneous points in time, called clock ticks, when it is 1

1.23.2 Physical Clock

- The abstract clock cannot be built electronically, but the same effect can be obtained with a square wave clock; the dff components treat a rising edge as the clock tick



1.24 Synchronous Circuits

- A circuit is synchronous if:
  - Every flip flop is connected directly to a unique global clock
  - No logic functions are performed on the clock signal
  - The circuit is designed so that every clock tick reaches each flip flop simultaneously (a little bit of variation is okay, but not much)
  - Every feedback loop in the circuit passes through a flip flop (no feedback in pure combinational logic)
  - The inputs to the circuit are assumed to remain stable throughout an entire clock cycle

1.25 Clock Cycles

- Clock ticks are points in time. Clock cycles are intervals of time between two ticks
- At a clock tick, the flip flops get new states which remain stable through the cycle, and the inputs get new values which remain stable
- During the clock cycles, the combinational logic settles down, and eventually al the signals become valid
- Then the clock ticks. The clock must be run slowly enough to ensure that all signals become valid!

1.26 Clock Speed

- Number of ticks per second is measured in Herz (Hz)
- A typical computer has about 3 billion ticks per second: 3GHz
- The duration of a clock cycle is the time between ticks: about ⅓ ns (nanoseconds)

1.27 Flip Flops and Registers

- A flip flop remembers a bit for a single clock cycle
- A register remembers a bit until you tell it to load a new value, so its memory can last a long time

1.28 The 1-bit Register

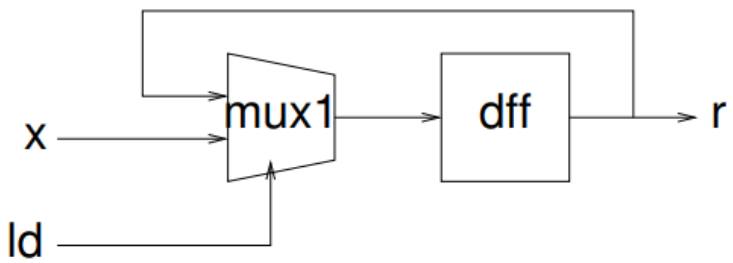
- Remembers a bit
- Receives a control input ***ld*** ("load") and a data input ***x***, and outputs a 1-bit state
- Initial value is zero
- At each clock tick, the register loads the data value ***x*** and just retains its previous state
- Throughout the clock cycle, the register outputs its state value, which will not change (until, perhaps, the next tick)
- The registers in a computer are constructed from many copies of the **reg1** circuit

1.28.1 Designing the 1-bit register

- A **dff** is needed to hold the state
- Since the **dff** will load a new state value at every clock tick, we need to use combinational logic to determine what that value should be
  - If **ld == 1** then the input to the dff should be the data input **x**
  - Otherwise, the input to the dff should just be its old state value
- Therefore **`dff_input = (if ld=0 then old_state else x)`**
- The conditional is implemented with a multiplexer
- **`dff_input = mux1 ld old_state x`**

1.29 Reg1

- Inputs:
  - A control input **`ld`** (stands for load)
  - A data input bit **x**
- Output:
  - State of the register r



1.30 Simulation Tables

- A simulation table provides a systematic way to calculate and display the execution of a synchronous digital circuit
- Every time there is a clock tick, a new row is added to the bottom of the table
- Every signal of interest has a column, giving its values through time

1.30.1 Simulating the Register

- The simulation table will show the inputs, the state, and the internal signals for each clock cycle
- Assume the initial state of the flip flop is "?". The circuit should never use this value
- At the start of each clock cycle, the values of the input signals are established by the "outside world", and remain stable for the entire cycle

1.30.2 Receive Inputs

- Before the first tick, the initial state of the flip flop is "?"
- This is also the value of signal **r**
- Inputs are given: **ld = 1** and **x = 1**
- Calculate
  - `dff_input = mux1 ld r x`
  - `= mux 1 ? 1`
  - `= 1`
- At the clock tick, we will use **dff_input** but not **r**. Therefore, the undefined initial value of the flip flop will be thrown away, and won't cause problems

cycle	inputs		state	internal
	ld	*x*	*r*	*dff_input*
0	1	1	?	1

1.30.3 Update Flip Flops

- Tick ends cycle 0 and begins cycle 1
- The new horizontal line symbolises the tick
- State of the flip flop is replaced by value of its input signal **dff_input**

cycle	inputs		state	internal
	ld	*x*	*r*	*dff_input*
0	1	1	?	1
1			1	

- Repeat these steps until you have finished executing

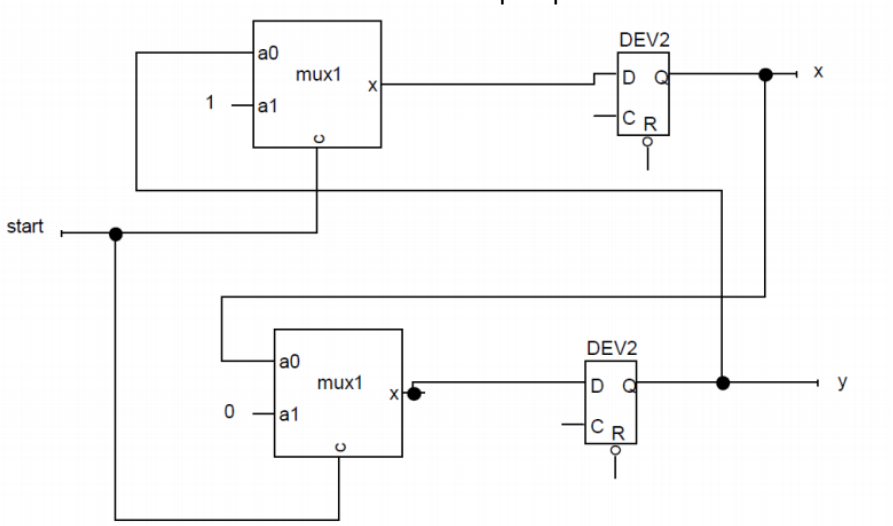
cycle	inputs		state	internal
	ld	*x*	*r*	*dff_input*
0	1	1	?	1
1	1	0	1	0
2	0	1	0	0
3	1	1	0	1
4			1	

1.31 Simultaneous Update of State

- The state of a circuit consists of the states of all the flip flops
- In a programming language, you often think of executing a sequence of statements, where a statement will modify one variable and leave all the others unchanged
- In a digital circuit, all the flip flops change their state *simultaneously* at the clock tick
- This is the big difference between hardware and software

1.32 Parallel Assignment

- Here is a circuit with feedback and two flip flops:



- At each clock tick it swaps the values of **x** and **y**

1.32.1 Modelling State with Parallel Assignment

- The values of **x** and **y** are updates simultaneously at the clock tick
- You can think of the flip flop states as variables
- But the circuit does not correspond to a sequential program
- It acts like parallel assignments, which are supported in some programming languages
  - All the right-hand sides are evaluated in parallel
  - Then all the variables are updated in parallel

```
begin parallel
  {  x = dff (mux1 start y one)
     y = dff (mux1 start x zero)
end parallel
```

1.32.2 Effect of Parallel Assignment

- To run the circuit, set the control input **start** to 1 for a clock cycle, then leave it 0 while the circuit executes
- If the control signal **start** is 1, then **x** initialises to 1 and **y** initialises to 0
- If **start** is 0, the flip flops keep exchanging their values

Cycle	start	x	y
0	1	?	?
1	0	1	0
2	0	0	1
3	0	1	0
4	0	0	1
...	...	...	...

1.33 Instructions

- The expression in the right-hand side of an assignment statement can be arbitrarily large and complex
- For a digital circuit, we need:
  - Simple statements (e.g just one arithmetic operation)
  - Statements with a fixed form
  - A small number of types of statements
- These are called instructions
- Our circuit will use two instructions, of the form:
  - **R2 := 6**
  - **R3 := R1 + R0**

1.34 Register File

- A register file is an array of registers: **R0, R1, R2, R3** etc
- Each variable is held in a register
- We refer to a variable by its *address*
- An address is a binary number 0, 1, 2, 3, …
- The register file circuit does:
  - It contains the array of registers
  - Each register holds a word (binary number)
  - It enables you to specify an address and read out that register
  - It enables you to specify another address and load a data value into that register

1.34.1 Behaviour of Register File

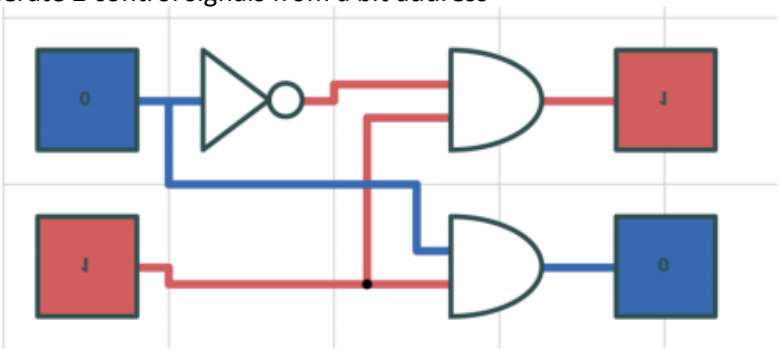
- At clock tick: **if ld = 1 then Reg[d] := x**
- During clock cycle: **a = Reg[sa]** and **b = Reg[sb]**
- The operation is determined by the control signals ***ld, d, sa, sb***
- The data input is **x**, and there are data outputs **a, b**

1.34.2 Loading into a Register File: Demultiplexer

- Often, every register in the register file retains its previous state
- Sometimes we will modify one register in the register file
- We need a way to take a destination register number (**R0 and R1**) and tell just that register to perform a load
- This is done with a *demultiplexer*
- **R0** and **R1** each need an individual load control
- The register selected by **d** gets the regfile load input
- The register not selected by **d** gets a load control of 0

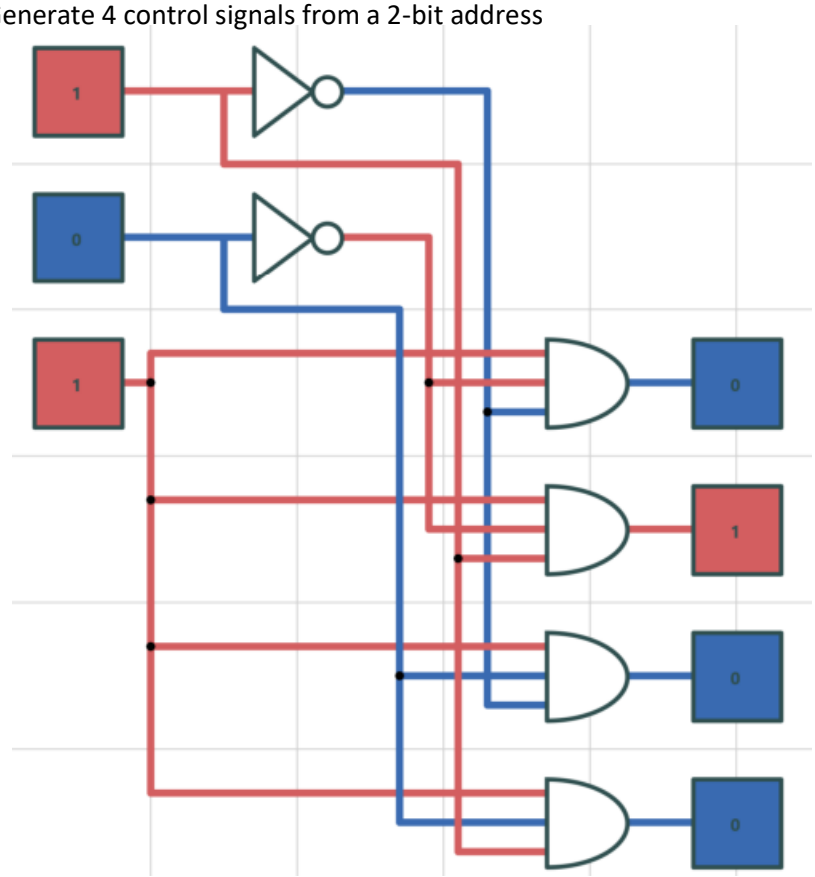
1.35 Demux1

- Generate 2 control signals from a bit address



1.36 DEMUX2

- Generate 4 control signals from a 2-bit address



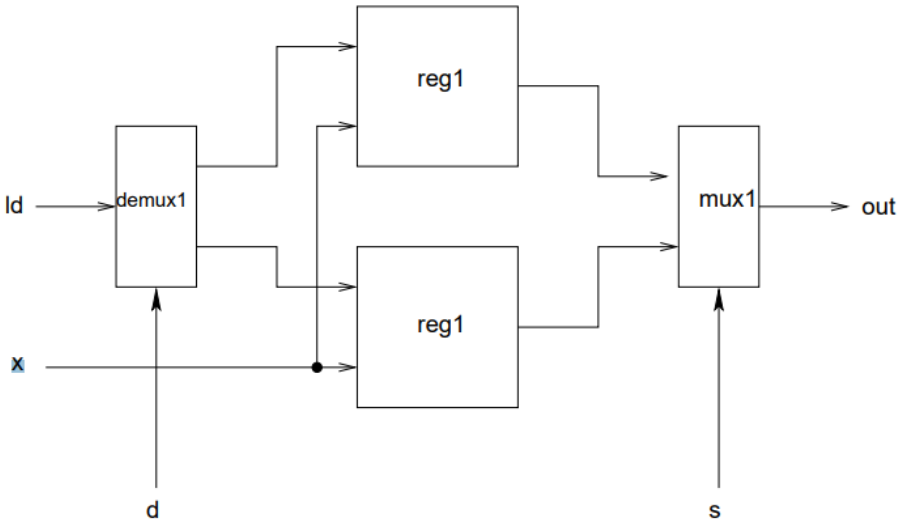
1.37 TYPES OF MULTIPLEXERS

- A bit-level multiplexer **mux1** uses a **control bit** to choose between **two data bits**
- A world-level multiplexer **wmux1** uses a **control bit** to choose between **two data words**
- A multiplexer for 4-bit words consists of **4 copies of the basic multiplexer**

1.38 IMPLEMENTATION OF REGISTER FILE

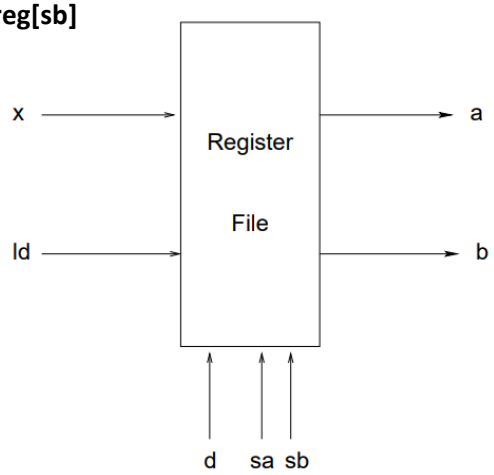
1.38.1 2 Registers, 1 Readout

- The output is the value of **reg[s]**
- At a clock tick, if **ld** then **reg[d] := x**



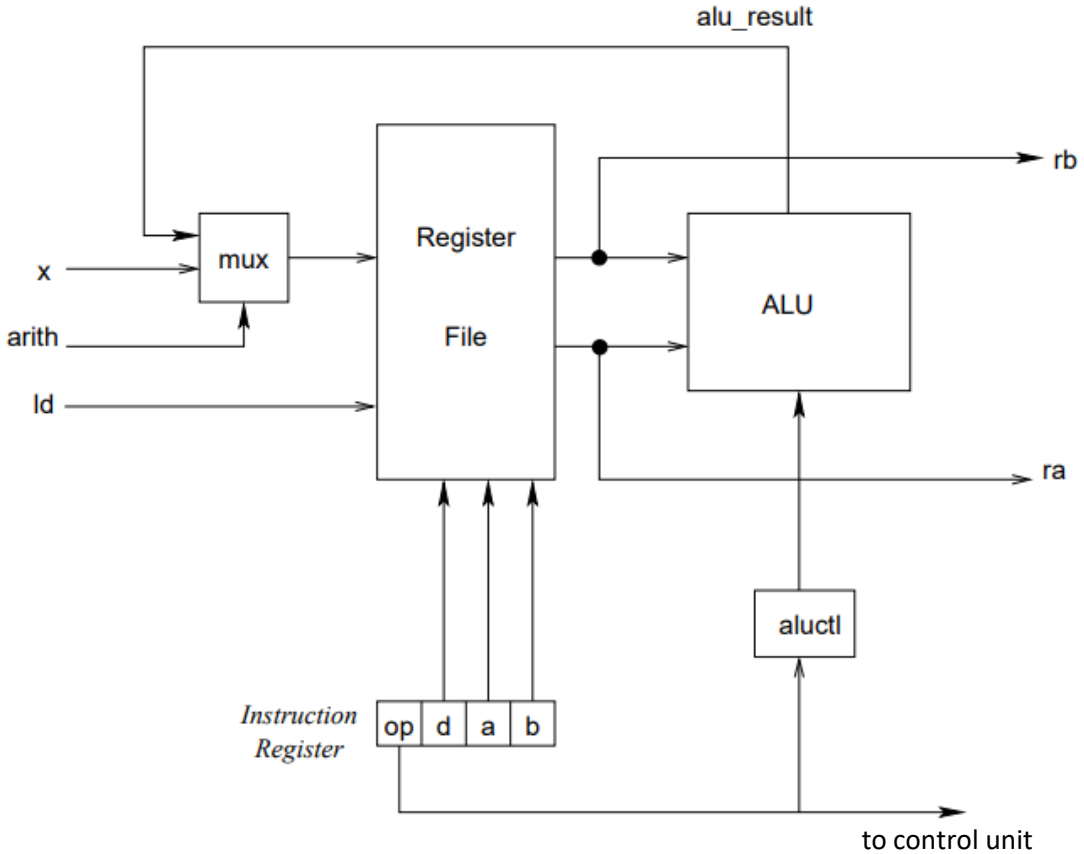
1.38.2 Register File with 2 Readouts

- Our aim is to do calculations like **p + q** or **x – y**
- The variables will be in the registers
- So we need to read out two registers, not just one
- Therefore, we extend the register file so we give it two sources addresses **sa, sb** and it will read out both
  - **a = reg[sa]**
  - **b = reg[sb]**



1.39 Register Transfer Machine

- The register file produces two outputs and the adder requires two inputs
- The adder produces one data word output, and the register file takes one data word input
- We can connect them in a feedback loop
- The effect will be:
  - **If ld then reg[d] := reg[sa] + reg[sb]**
- Add an external data input, and use a multiplexer to decide whether the external input or the ALU result should be sent to the register file



1.40 Behaviour of RTM

- Data input **x**
- Control inputs **ld**, **arith**
- Address inputs **d, sa, sb** (specify registers)
- At clock tick:

```
if ld
   then if arith
          then reg[d] := reg[sa] + reg[sb]
          else reg[d] := x
```

- All other registers remain unchanged (the variables you aren't assigning to don't change)

1.41 Controlling a Circuit

- The "core" of a circuit is a register file and an adder
- To make it perform useful computations, we need to control it
- This is done with **control signals**
- A control signal is a bit, just like any other signal, but it helps conceptually to make a distinction between **data signals** and **control signals**

1.42 Running a Program on RTM

- For now, let's just consider a fixed sequence of simple assignment statements
- Each variable must be a register: **R0, R1, R2, R3**
- Each statement must be written in one of these forms:
  - **reg = const**
  - **reg = reg + reg**
  - **reg = reg – reg**
- Example:
  - **R1 = 3**
  - **R2 = 1**
  - **R3 = R1 + R2**
  - **R1 = R2 + R2**

1.42.1 How to Execute R1 := 5

- Tell it to use the **indata** input (the hex keypad) as the data input to register file: **ctl_add = 0**
- Tell it to put the data into **R1: ctl_d1, ctl_d0 = 01**
- Tell it to perform a load (actually, the circuit does a load every clock cycle, but in a real computer we would have to set **ctl_ld = 1**)
- Note: the values of **ctl_sa1, ctl_sa0, ctl_sb0** don't matter. The registers they specify will be read out and you can look at them on the hex displays, but these values won't actually be used
- Perform a clock tick: pulse clock (click it to 1, then click it to 0)

1.42.2 How to Execute R2 := R0 + R3

- Tell it to read out R0 on the **a** output: **ctl_sa1, ctl_sa0 = 00**
- Tell it to read out R3 on the **b** output: **ctl_sb1, ctl_sb0 = 11**
- Tell it to use the adder output as the data input to register file: **ctl_add = 1**
- Tell it to put the data into **R2: ctl_d1, ctl_d0 = 10**
- Tell it to perform a load (actually, the circuit does a load every clock cycle, but in a real computer we would have to set **ctl_ld = 1**)
- Note: the value of **indata** is ignored, so you can set it to any value you like — makes no difference. This is called a “don’t care” value
- Perform a clock tick: pulse clock (click it to 1, then click it to 0)

2 COMPUTER ARCHITECTURE

2.1 MACHINE LANGUAGE

- A fixed digital circuit can execute one fixed **machine language**
- Example:
  - Intel Core or Pentium (there is a family of Intel processors "x86" and their machine languages are related, but not identical)
  - ARM
  - MIPS
  - Sparc
  - … and many more
- The details of different machine languages are quite different, but we will focus on the principles common to all of them

2.2 INSTRUCTIONS

- A machine language provides **instructions**
- Like statements in a programming language, with some differences:
  - Statements can be complex: **x := 2 * (a + b/c)**
  - Instructions are simple: **R2 := R1 + R3**
  - Each instruction just performs one operation

2.3 STRUCTURE OF A COMPUTER

- The **register file** is a set of 16 "registers"; this is the set of registers in the RTM circuit. They are often named **R0, R1, R2, …, R15**
- A register is a circuit that can remember a 16-bit word
- The **ALU** (arithmetic and logic unit) is a circuit that can do arithmetic, such as addition, subtraction, comparison, and some other operations
- The **memory** can hold a large number of words. It's similar to the register file, but significantly slower and much larger
- The **input/output** can transfer data from the outside world to/from the memory

2.4 REGISTERS

- There are 16 registers
- Each register holds a 16-bit word (we'll write the words using hexadecimal)
- Recall the **register transfer machine (RTM) circuit**
- It contains four register, each holding 4 bits
- Sigma16 is just the same, but with 16 registers and each holds 16 bits
- Each 16-bit register is 16 copies of the **reg1** circuit
- In machine language, we are programming directly with the hardware in the computer, so we use registers instead of variables

2.5 ADD INSTRUCTION

- Examples:
  - **add R5,R2,R3** ; means R5 := R2 + R3
  - **add R12,R1,R7** ; means R12 := R1 + R7
- General form:
  - **add dest,op1,op2** where dest, op1, op2 are **registers**
  - meaning: **dest := op1 + op2**
- Everything after a semicolon is a comment

2.6 REGISTERS CAN HOLD VARIABLES

- An **add** instruction (or **sub, mul, div**) is like an assignment statement
- **add R2,R8,R2** means **R2 := R8 + R2**
  1. Evaluate the right-hand side **R8 + R2**
  2. The operands **(R8, R2)** are not changed
  3. Overwrite the left-hand side (destination) (**R2**) with the result
  4. The old value of the destination is destroyed
- It is not a mathematical equation, it is a command to do an operation and put the result into a register, overwriting the previous contents
- The **:=** operator means *assign*, and does not mean *equals*

2.7 MORE ARITHMETIC INSTRUCTIONS

- `add R4,R11,R0 ; R4 := R11 + R0`
- `sub R8,R2,R5 ; R8 := R2 - R5`
- `mul R10,R1,R2 ; R10 := R1 * R2`
- `div R7,R2,R12 ; R7 := R2 / R12`
- Every arithmetic operation takes its operands from registers, and loads the result into a register

2.8 SPECIAL REGISTERS

- You should not use **R0** or **R15** to hold ordinary variables
- **R0 always contains 0**
  - Any time you need the number 0, it's available in **R0**
  - You cannot change the value of **R0**
  - **add R0,R2,R3** ; does nothing – **R0** will not change
  - it is **legal** to use **R0** as the destination, but it will still be 0 after you do it
- **R15 holds status information**
  - Some instructions place additional information in **R15** (is the result negative? Was there an overflow?)
  - Therefore, the information in **R15** is temporary
  - **R15** is not a safe place to keep long-term data

2.9 MEMORY

- Memory is similar to the register file: it is a large collection of words
- A variable name (x, sum, count) refers to a word in memory
- Some differences between memory and register file:
  - The memory is much larger: **65,536** locations (the register file has only 16)
  - The memory cannot do arithmetic
- So, our strategy in programming:
  - Keep data permanently in memory
  - When you need to do arithmetic, copy a variable from memory to a register
  - When finished, copy the result from a register back to memory

2.10 REGISTERS AND MEMORY

- The **register file**
  - 16 registers
  - Can do arithmetic, but too small to hold all your variables
  - Each register holds a 16-bit word
  - Names are **R0, R1, R2, …, R15**
  - Use registers to hold data temporarily
- The **memory**
  - 65,536 memory locations
  - Each memory location holds a 16-bit word
  - Each memory location has an address 0, 1, 2, …, 65,535
  - The machine cannot do arithmetic on a memory location
  - Use memory locations to store program variables permanently, Also, use memory locations to store the program

2.10.1 Copying a word between memory and register

- There are two instructions for accessing the memory
- **load** copies a variable from the memory to a register
  - **load R2,x[R0]** copies the variable **x** from memory to register **R2**
  - **R2 := x**
  - **R2** is changed; **x** is unchanged
- **store** copies a variable from a register to memory
  - **store R3,y[R0]** copies the word in register **R3** to the variable **y** in memory
  - **y := R3**
  - **y** is changed; **R3** is unchanged

2.10.2 Assignment statements in machine language

- **x := a+b+c**

```
load   R1,a[R0]      ; R1 := a
load   R2,b[R0]      ; R2 := b
add    R3,R1,R2      ; R3 := a+b
load   R4,c[R0]      ; R4 := c
add    R5,R3,R4      ; R5 := (a+b) + c
store  R5,x[R0]      ; x := a+b+c
```

- Use **load** to copy variable from memory to registers
- Do arithmetic with **add, sub, mul, div**
- Use **store** to copy result back to memory

2.11 Constants: Lea

- The **RTM** has an instruction that loads a constant into a register
- Use the **lea** instruction
- **lea R2, 57[R0]** loads the constant 57 into R2: **R2 := 57**
- General form: **lea reg, const[R0]**

2.12 Stopping the Program

- The last instruction should be:
  - **trap R0, R0, R0** ; halt
- This tells the computer to halt; it stops execution of the program

2.13 Defining Variables

- To define variables **x, y, z** and give them initial values

```
x    data   34    ; x is a variable with initial value 34
y    data    9    ; y is initially 9
z    data    0    ; z is initially 0
abc  data  $02c6  ; specify initial value as hex
```

- The data statements should come **after** all the instructions in the program

3 Control Structures

3.1 Compiling

- The computer cannot execute programs in a high-level language
- Therefore, we must **translate** a program into assembly language
- Translating from a high-level programming language to assembly language is called compiling
- This is done by software called a compiler: it reads in a program in e.g. C++ and translates it to assembly language
- There are many benefits of using compilers
  - We can have many compilers, one for each language, so a computer can run programs in *many* languages
  - The compilers can make programming easier: good error messages, etc.
  - Languages can be designed to fit well for different purposes
- For each type of high-level language construct, we will translate to assembly language following a standard pattern

3.2 Control Structures

- A program contains:
- Statements that perform calculations
  - Assignment statements
- Statements that determine what order the calculations occur in
  - Conditionals: **if-then-else**
  - Loops: **while, repeat, for**
  - Structuring computation**: functions, procedures, coroutines, recursion**

3.3 High-Level Control Structures

- Notation:
  - **S_1, S_2, S_3,** etc means "any statement" (e.g an assignment statement)
  - **bexp** means any Boolean expression
- **Block**
  - We can treat several consecutive statements as just a single statement:
  - **{ S_1; S_2; S_3; }**
- **if-then**
  - if **bexp** then **S**;
- **if-then-else**
  - if **bexp** then **S_1** else **S_2**
- **while-loop**
  - while **bexp** do **S**
- and many more…

3.4 Low-Level Constructs

- Assignment statements: **x := a * 2**
- Goto: **goto computeTotal**
- Conditional: **if x < y then goto loop**
- First we translate high-level constructs into these low level statements
- Then translate the low-level statements into assembly language

3.5 The Goto Statement

- Many programming languages have a **goto** statement
- Any statement may have a **label** (for example "loop")
- Normally, execution proceeds from one statement to the next, on and on
- A **goto L** transfers control to the statement with label **L**

```
        S;
loop:   S;
        S;
        S;
        goto loop;
```

3.5.1 Using the goto statement

- goto leads to unreadable programs and unreliable software
- The modern view:
  - In a high-level language, you should **not** use goto
  - For low-level programming – like assembly language – the goto servers as the foundation for implementing the higher-level control statements
- We will use to forms:
  - **goto L**
  - **if b then goto L**

3.5.2 The conditional goto statement

- if **bexp** then **goto label**
- **bexp** is a Boolean expression: **x < y, j = k, abc > def**
- If the **bexp** is **True**, the statement goes to the label
- Otherwise, we just move on to the next statement
- The only thing you can put after **then** is a goto statement

3.6 JUMPING

- The foundation of control structures is **jump** instructions
- **Jumping** is the machine language equivalent of **goto**
- An instruction may have a **label**
- The **label** is a name, starting with a letter
- The unconditional instruction **jump loop[R0]** means **goto loop**

3.7 COMPARISON INSTRUCTION

- **cmplt R2, R5, R8**
- Means "compare for less than"
- The operands are compared: **R5 < R8**
- This gives a Boolean, 0 (for False) or 1 (for True)
- That Boolean result is loaded into the destination **R2**
- There are three of these instructions:
  - **cmplt** – compare for **less than**
  - **cmpeq** – compare for **equal**
  - **cmpgt** – compare for **greater than**

3.8 CONDITIONAL JUMPS

- There are two instructions: you can jump if a Boolean is False or True
- **jumpf** – jump if **false**
  - **jumpf R4,loop[R0]**
  - means if **R4** contains **false**, then **goto loop**
  - 0 means **false**, so this means if **R4=0** then **goto loop**
- **jumpt** – jump if **true**
  - **jumpt R5,check[R0]**
  - means if **R5** contains **true**, then **goto check**
  - Any number other than 0 means **true**, so this means if **R5 != 0** then **goto check**

3.9 Compilation Patterns

- Each programming construct can be translated according to a standard pattern
- It's useful to translate in two steps:
  - First, translate complex statements to simple high-level statements (**goto label**, if **b** then **goto label**)
  - The "goto form" of the algorithm corresponds closely to machine instructions
  - Then it's straightforward to complete the translation to assembly language
    - Assignment statements — loads, then arithmetic, then store
    - **goto label** — **jump label[R0]**
    - if **b** then **goto label** — **jumpt R5,label[R0]** where **R5** contains **b**
    - if not **b** then **goto label** — **jumpf R5,label[R0]** where **R5** contains **b**
- This approach clarifies how the algorithm works

3.10 Compiling an Assignment Statement

- Load the operands; do calculations; store results

```
; x := a + b*c;
   load  R1,a[R0]    ; R1 = a
   load  R2,b[R0]    ; R2 = b
   load  R3,c[R0]    ; R3 = c
   mul   R4,R2,R3    ; R4 = b*c
   add   R4,R1,R4    ; R4 = a + (b*c)
   store R4,x[R0]    ; x := a+(b*c)
```

3.10.1 If bexp then S

```
if x<y
  then {statement 1;}
statement 2;
```

- Translates into

```
   R7 := (x < y)
   jumpf R7,skip[R0]
   instructions for statement 1
skip
   instructions for statement 2
```

3.10.2 If bexp then S1 else S2

```
if x<y
  then { S1 }
  else { S2 }
S3
```

Compiled into:

```
   R5 := (x<y)
   jumpf R5,else[R0]
; then part of the statement
   instructions for S1
   jump   done[R0]
; else part of the statement
else
   instructions for S2
done
   instructions for statement S3
```

3.10.3 While b do S

```
while i<n do
  { S1 }
S2
```

Compiled into:

```
loop
   R6 := (i<n)
   jumpf  R6,done[R0]
   ... instructions for the loop body S1 ...
   jump   loop[R0]
done
  instructions for S2
```

3.10.4 Infinite loops

```
while (true)
  {statements}
```

Compiled into:

```
loop
   ... instructions for the loop body ...
   jump   loop[R0]
```

3.11 Nested Statements

- For each kind of high-level statement, there is a pattern for translating it to
  - Low-level code (goto)
  - Assembly language
- In larger programs, there will be nested statements

```
if b1
  then { S1;
         if b2 then {S2} else {S3};
         S4;
       }
  else { S5;
         while b3 do {S6};
       }
S7
```

3.11.1 Compiling nested statements

- A block is a sequence of instructions where
  - To execute it, always start with the first statement
  - When it finishes, it always reaches the last statement
- Every statement should be compiled into a block of code
- This block may contain internal structure – it may contain several smaller blocks – but to execute it you should always begin at the beginning and it should always finish at the end
- The patterns work for nested statements
- You need to use new labels (can't have a label called "skip" in several places)

3.12 Programming Technique

- There are two ways to handle variables:
- The **statement-by-statement** style:
  - Each statement is compiled independently
  - **load, arithmetic, store**
  - Straightforward, but inefficient
  - Use this style if you feel confused
- The **register-variable** style
  - Keep variables in registers across a group of statements
  - Don't need as many loads and stores
  - More efficient
  - You have to keep track of whether variables are in memory or a register
  - Use comments to show register usage
  - Real compilers use this style
  - Use this style if you like the shorter code it produces

We'll translate the following program fragment to assembly language, using each style:

```
x = 50;
y = 2*z;
x = x+1+z;
```

3.12.1 Statement-by-statement

```
; x = 50;
     lea     R1,$0032    ; R1 = 50
     store   R1,x[R0]    ; x = 50

; y = 2*z;
     lea     R1,$0002    ; R1 = 2
     load    R2,z[R0]    ; R2 = z
     mul     R3,R1,R2    ; R3 = 2*z
     store   R3,y[R0]    ; y = 2*z

; x = x+1+z;
     load    R1,x[R0]    ; R1 = x
     lea     R2,1[R0]    ; R2 = 1
     load    R3,z[R0]    ; R3 = z
     add     R4,R1,R2    ; R4 = x+1
     add     R4,R4,R3    ; R4 = x+1+z
     store   R4,x[R0]    ; x = x+1+z
```

3.12.2 Register-variable

```
; Usage of registers
;   R1 = x
;   R2 = y
;   R3 = z

; x = 50;
     lea     R1,$0032    ; x = 50
     load    R3,z[R0]    ; R3 = z
     lea     R4,$0002    ; R4 = 2
; y = 2*z;
     mul     R2,R4,R3    ; y = 2*z
; x = x+1+z;
     lea     R4,$0001    ; R4 = 1
     add     R1,R1,R4    ; x = x+1
     add     R1,R1,R3    ; x = x+z
     store   R1,x[R0]    ; move x to memory
     store   R2,y[R0]    ; move y to memory
```

4 THE STORED PROGRAM COMPUTER

4.1 STORED PROGRAM COMPUTER

- Variables, data structures, arrays, lists and the machine language program itself is stored inside the computer's main memory
- An alternative approach is to have a separate memory to hold the program, but experience has shown that to be inferior for general purpose computers

4.2 INSTRUCTION FORMATS

- Sigma16 has three kinds of instructions:
  - **RRR** instructions use the **registers**
  - **RX** instructions use the **memory**
  - **EXP** instructions use **registers and constants**
- Each kind of instruction is called an **instruction format**
- All the instructions with the same format are similar
- Each instruction format has a standard representation in the memory
- The machine language program is in the memory, so we need to represent each instruction as a word
- An instruction format is a systematic way to represent an instruction using a string of bits on one or more words
- Every instruction is either **RRR, RX,** or **EXP**
  - An **RRR** instruction is represented in **one word** (16 bits)
  - An **RX** or **EXP** instruction is represented in **two words**
- We just need to learn three ways to represent an instruction
- For now, we just need **RRR** and **RX** (**EXP** is needed only for some more advanced instructions)

4.3 FIELDS OF AN INSTRUCTION WORD

- An instruction word has **16 bits**
- There are four fields, each **4 bits**
- We write the value in a field using **hexadecimal**
- The fields have standard names:
  - **op** – holds the operation code
  - **d** – usually holds the destination register
  - **a** – usually holds the first source operand register
  - **b** – usually holds the second source operand register

4.4 RRR INSTRUCTIONS

- Every **RRR** instruction consists of
  - An operation (e.g add)
  - Three register operands: a destination and two operands
  - The instruction performs the operation on the operands and puts the result in the destination

4.4.1 Representing RRR

- We need to specify which **RRR** instruction it is. Is it add? Sub? Mul? Another?
- This is done with an **operation code** – a number that says what the operation is
- There are about a dozen **RRR** instructions, so a 4-bit operation code suffices
- We also need to specify three registers: **dest** and two **source** operands
- There are 16 registers, so a particular one can be specified by 4 bits
- Total requirements: 4 fields, each 4 bits – total 16 bits
- An **RRR** instruction fills one word exactly

4.4.2 RRR form

- All **RRR** instructions have the same form, just the operation differs
  - **add R2,R2,R5** ; R2 = R2 + R5
  - **sub R3,R1,R3** ; R3 = R1 – R3
  - **mul R8,R6,R7** ; R8 = R6 * R7
- In **add R2,R5,R9** we call **R5** the first operands, **R9** the second operand, and **R2** the destination
- It's ok to use the same register as an operand and destination

4.4.3 Some RRR operation codes

mnemonic	operation code
add	0
sub	1
mul	2
div	3
⋮	⋮
trap	d

4.4.4 RRR examples

- **add R13,R6,R9**
- Since each filed of the instruction is 4 bits, written as a hex digit
- The opcode is 0
- Destination register is 13 (hex d)
- Source operand registers are 6 and 9 (hex 6 and 9)
- So the instruction is: **0d69**

4.5 RX INSTRUCTIONS

- Every **RX** instruction contains two operands:
  - A register
  - A memory location
- We have seen several so far:
  - **lea R15,19[R0]** ; R5 = 19
  - **load R1,x[R0]** **;** R1 = x
  - **store R3,z[R0]** ; z = R3
  - **jump finished [R0]** ; goto finished
- A typical **RX** instruction: **load R1,x[R0]**
- The first operand (**R1**) is called the **destination register**, just like for **RRR** instructions
- The second operand **x[R0]** specifies a **memory address**
- Each variable is kept in memory at a specific location
- The memory operand has two parts:
  - The variable **x** is a name for the address where **x** is kept – called the **displacement**
  - The **R0** part is just a register, called the **index register**

4.5.1 Format of RX instruction

- **load R1,x[R0]**
- There are two words in the machine language code
- The first word has 4 fields: **op, d, a, b**
  - **op** contains **f** for every **RX** instruction
  - **d** contains the register operand (in the example, 1)
  - **a** contains the index register (in the example, 0)
  - **b** contains a code indicating which **RX** instruction this is (1 means **load**)
- The second word contains the displacement (address) (in the example, the address of **x**)
- Suppose **x** has memory address **0008**. Then the machine code for **load R1,x[R0]** is:
  - **f101 0008**

4.5.2 Operation codes for RX instructions

- Recall, for **RRR** the **op** field contains a number saying which **RRR** instruction it is
- For **RX**, the **op** field always contains **f**
- So how does the machine know which **RX** instruction it is?
- There is a secondary code in the **b** field

mnemonic	RX operation code (in b field)
lea	0
load	1
store	2
⋮	⋮

4.6 Assembly Language

- People write assembly language
  - The program is in text
  - It's easier to read
  - You don't need to remember all the codes
  - Memory addresses are much easier to handle
- The machine executes machine language
  - The program is words containing 16-bit numbers: **042c**
  - It's possible for a digital circuit (the computer) to execute
  - No names for instructions or variables: everything is a number
- Each statement corresponds to one instruction
- You can use names (**add, div**) rather than numeric codes (0,3)
- You can use variable names (**x, y, sum**) rather than memory addresses (**02c3, 18d2**)
- You write a program in assembly language
- The assembler translates it into machine language
- What's the relationship between compilers and assemblers?
  - Compilers translate between languages that are very different
  - Assemblers translate between very similar languages

4.7 The Assembler

- A person writes a machine-level program in assembly language
- A software application called the assembler reads it in, and translates it to machine language
- What does the assembler do?
  - When it sees an instruction mnemonic like **add** or **div**, it replaces it with the operation code (0, 3, etc…)
  - The assembler helps with variable names – the machine languages needs addresses and the assembler calculates them

4.7.1 Sequence of RRR instructions

- Assembly language:
  - **add R3,R5,R1**
  - **sub R4,R2,R3**
  - **mul R1,R9,R10**
- Machine language:
  - **0351**
  - **1423**
  - **219a**

4.8 How the Assembler Allocates Memory

- The assembler maintains a variable called the **location counter**. This is the address where it will place the next piece of code
- Initially, the location counter is 0
- The assembler reads through each line of code
    - If there is a label, it remembers that the value of the label is the current value of the location counter. This goes into the **symbol table**
    - The assembler decides how many words of memory this line of assembly code will require (add needs one word, load needs two), and adds this to the location counter
- Then the assembler reads through the assembly language program a second time
- Now it generates the words of object code for each statement. If there is a reference to a label that appears farther on (e.g **load x** or **jump loop**) it looks up the value of the label in the symbol table

4.9 Program Structure

- A complete program needs
    - Good comments
    - The actual program – a sequence of instructions
    - An instruction to stop the program: **trap R0, R0, R0**
    - Declarations of the variables: **data** statements
- Why do we put the instructions first, and define the variables at the end?
    - The assembler can find the definitions because it reads the program twice: the first pass finds all the labels, the second pass generates the machine language code
    - The computer will start executing at memory address 0, so there should be an instruction there, not data

4.9.1 Example add program

```
; Program Add
; A minimal program that adds two integer variables

; Execution starts at location 0, where the first instruction will be
; placed when the program is executed.

      load   R1,x[R0]    ; R1 := x
      load   R2,y[R0]    ; R2 := y
      add    R3,R1,R2    ; R3 := x + y
      store  R3,z[R0]    ; z := x + y
      trap   R0,R0,R0    ; terminate

; Static variables are placed in memory after the program

x     data  23
y     data  14
z     data  99
```

4.9.2 Snapshot of memory for example add program

address	contents	what the contents mean
0000	f101	first word of load R1,x[R0]
0001	0008	second word: address of x
0002	f201	first word of load R2,y[R0]
0003	0009	second word: address of y
0004	0312	add R3,R1,R2
0005	f302	first word of store R3,z[R0]
0006	000a	second word: address of z
0007	d000	trap R0,R0,R0
0008	0017	x = 23
0009	000e	y = 14
000a	0063	z = 99

4.10 Boot: Reading in the Program

- The program is placed in memory starting at location 0
- The program should finish by executing the instruction **trap R0, R0, R0**
- Normally, **trap R0, R0, R0** should be the last instruction of the program
- After the **trap R0, R0, R0** come the data statements, which tell the assembler the names of the variables and their initial values
- These conventions were typical for early computers

4.11 Control Registers

- Some of the registers in the computer are accessible to the programmer: **R0 – R15**
- There are several more registers that the machine uses to keep track of what it's doing
- These are called **control registers**
- They are (mostly) invisible to the program

4.12 PC and IR Control Registers

- When you hand execute a program, you need to know
  - Where you are
  - What you're doing
- The **PC** register (program counter) contains the **address of the next instruction** to be executed
- The **IR** (instruction register) contains **the instruction being executed right now**
- If an **RX** instruction is being executed, the **ADR** (address register) contains the **memory address of the second operand**.

4.13 Writing Constants

- In assembly language, you can write constants in either decimal or hexadecimal
  - Decimal: **50**
  - Hexadecimal: **$0032**

4.14 Comments

- In Sigma16, a semicolon **;** indicates that the rest of the line is a comment
- You can have a full line comment: just put **;** at the beginning
- You should use good comments in all programs, regardless of language
- But they are even more important in machine language, because the code needs more explanation
- At the beginning of the program, use comments to give the name of the program and to say what it does
- Use a comment on every instruction to explain what it's doing

5 Arrays

5.1 Address Arithmetic

- Every piece of data in the computer (in registers, or memory) is a **word**
- A word can represent several different kinds of data
  - So far, we've just been using **integers**
  - Represented with **two's complement**: -2^{15}, …, -1, 0, 1, 2, …,2^{15}-1
- Now we'll start doing computations with addresses too
- Addresses are **unsigned numbers**: 0, 1, 2, …, 65535

5.1.1 What can you do with address arithmetic?

- Powerful data structures
  - Arrays
  - Pointers and records
  - Linked lists, queues, dequeues, stacks, trees, graphs, hash tables, …
- Powerful control structures
  - Input/output
  - Procedures and functions
  - Recursion
  - Case dispatch
  - Coroutines, classes, methods

5.2 Data Structures

- An ordinary variable holds one value (e.g an integer)
- A **data structure** can hold many individual elements
- A data structure is a container
- The simplest data structure is an **array**

5.3 ARRAYS

- In mathematics, an array (vector) is a sequence of indexed values
- Arrays are ubiquitous: used in all kinds of applications
- In programming languages, we refer to $\mathbf{x_i}$ as **x[i]**

5.3.1 Representing an array

- An array is represented in a computer by placing the elements in consecutive memory locations
- The array **x** starts in memory at some location: here, it's **01a5**
- The address of the array **x** is the address of its first element **x[0]**
- The elements follow in consecutive locations

value		x[0]	x[1]	x[2]	x[3]	x[4]	x[5]	x[6]	
address	...	01a5	01a6	01a7	01a8	01a9	01aa	01ab	...

- The address of **x[i]** is **x+i**

5.3.2 Allocating an array

- An array is in memory along with other data – after the trap that terminates the program
- You can allocate the elements and give them initial value with data statements
- Use the name of the array as a label on the first element (the one with index 0)
- Don't put labels on the other elements

```
...
    trap   R0,R0,R0  ; terminate

; Variables and arrays

abc   data    25       ; some variable
n     data     6       ; size of array x

x     data    13       ; x[0]
      data   189       ; x[1]
      data   870       ; x[2]
      data    42       ; x[3]
      data     0       ; x[4]
      data  1749       ; x[5]

def   data     0       ; some other variable
```

5.3.3 Big arrays

- In real scientific computing, it's common to have large arrays with thousands, or even millions, of elements
- In large scale software, arrays are allocated **dynamically** with help from the operating system
  - The user program calculates how large an array it wants, and stores that in a variable (e.g n=40000)
  - It uses a **trap** to request (from operating system) a block of memory big enough to hold the array
  - The operating system returns the address of this block to the user program
- We won't be doing this: we will just allocate small arrays using **data** statements

5.3.4 Accessing an element of an array

- Suppose we have array **x** with elements **x[0], x[1], …, x[n-1]**
- Elements are stored in consecutive memory locations
- Use the label **x** to refer to the array; **x** is also the location of **x[0]**
- To do any calculations on **x[i]**, we must load it into a register, or store a new value into it
- If you try **load R4,x[R0]** the effect will be **R4 := x[0]**
- We need a way to access **x[i]** where **i** is a variable

5.4 Effective Address

- An **RX** instruction specifies addresses in two parts, for example **result[R0]** or **x[R4]** or **$00a5[R2]**
  - The **displacement** is a 16-bit constant (you can write the number, or use a name and the assembler will put in the address for you)
  - The **index register** is written in brackets
- The machine adds the displacement to the value in the index register – this is called the **effective address**
- The instruction is performed using the effective address

5.4.1 Using the effective address

- The addressing mechanism is flexible
- You can access ordinary variables:
  - **load R2,sum[R0]**
  - **R0** always contains 0, so the effective address is just the address of **sum**
- You can access an array element:
  - If **R8** contains an index **i**, then
  - **load R2,x[R8]**
  - will load **x[i]** into **R2**
- There's more: effective addresses are used to implement pointers, functions, procedures, methods, classes, instances, jump tables, case dispatch, coroutines, records, interrupt vectors, lists, heaps, trees, forests, graphs, hash tables, activation records, stacks, queues, dequeues, …

5.4.2 Using effective address for an array

- Suppose we want to execute **x[i] := x[i] + 50**

```
lea   R1,50[R0]    ; R1 := 50
load  R5,i[R0]     ; R5 := i
load  R6,x[R5]     ; R6 := x[i]
add   R6,R6,R1     ; R6 := x[i] + 50
store R6,x[R5]     ; x[i] := x[i] + 50
```

5.5 Array Traversal

- A typical operation on an array is to **traverse** it
- That means to perform a calculation on each element
- Here's a loop that doubles each element of **x**:

```
i := 0;
while i < n do
  { x[i] := x[i] * 2;
    i := i + 1;
  }
```

5.6 For Loops

- A for loop is designed specifically for array traversal
- It handles the loop index automatically
- It sets the index to each array element index and executes the body
- The intuition is "*do the body for every element of the array*"

```
for i := exp1 to exp2 do
  { statements }
```

5.7 Array Traversal with While and For Loops

- Here is the program that doubles each element of **x**, written with both constructs

```
i := 0;
while i < n do
  { x[i] := x[i] * 2;
    i := i + 1;
  }
```

```
for i := 0 to n-1 do
    { x[i] := x[i] * 2; }
```

5.8 Translating the For Loop to Low Level

```
for i := exp1 to exp2 do
   { statement1;
     statement2;
   }
```

s

Translate to low level with this pattern:

```
       i := exp1;
loop:  if i > exp2 then goto loopdone;
       statement1;
       statement2;
       i := i + 1;
       goto loop;
loopdone:
```

5.9 Tip: Copying One Register to Another

- Here's a useful tip
- Sometimes you want to copy a value from one register to another
  - **R3 := R12**
- There's a standard way to do it:
  - **add R3,R12,R0** ; R3 := R12
- The idea is that **R12 + 0 = R12**
- It's actually more efficient to do it this way than providing a separate instruction just to copy the register

5.10 Using Load and Store

- A common error is to confuse **load** and **store**
- The main points to remember:
  - We need to keep variables in memory (most of the time) because memory is big – there aren't enough registers to hold all your variables
  - The computer hardware can do arithmetic on data in registers, but it cannot do arithmetic on data in memory
  - Therefore, to do arithmetic on variables, you must
    - Copy the variables from memory to registers (**load**)
    - Do the arithmetic in the registers (**add, sub, …**)
    - Copy the result from registers back to memory (**store**)

6 Records and Pointers

6.1 Compilation Patterns

- We have looked at several high-level programming constructs
  - *If b then S*
  - *If b then S else S*
  - *while b do S*
  - *for var := exp to exp do S*
- There is a standard way to translate each to low level form:
- Assignment, goto L, if b then goto L
- The low-level statements correspond closely to instructions
- Following these patterns helps you understand precisely what high-level language constructs mean – this is one of the aims of the course
- This is essentially how real compilers work, and this is another aim of the course
- This saves time because
  - It's quicker to catch errors at the highest level (e.g. translating the if-then-else to goto) rather than the lowest level (instructions)
  - It makes the program more readable, and therefore faster to check and to debug
- This leads to good comments that make the program more readable
- This approach scales up to large programs
- Experienced programmers recognise the patterns, so I you use them your code is easier to read, debug and maintain

6.1.1 How can you tell if you're using the pattern?

- Each pattern contains
  - Changeable parts: Boolean expressions, integer expressions, statements
  - Fixed parts: goto, if-then-goto
  - The labels have to be different every time, but the structure of the fixed parts never changes
- Example: translating a while loop
  - There should be one comparison, one conditional jump at the start of the loop
  - There should be one unconditional jump at the end of the loop

High level code:

```
while bexp do
    S
```

The pattern for translation to low level:

```
label1
    if bexp = False then goto label2
    S
    goto label1
label2
```

6.2 Comments

- Initial comments to identify the program, author, date
- Early comments to say what the program does
- High-level algorithm (in comments)
- Translation to low-level algorithm (in comments)
- Translation to assembly language (with comments)
  - Copy the low-level algorithm comments, and paste, so you have two copies
  - The first copy remains as the low-level algorithm
  - In the second copy, insert the assembly language code
  - Every low-level statement should appear as a comment in the assembly code
- The program development methodology entails writing the comments *first*
- Avoid the temptation of writing code first, hacking it until it appears to work, and then adding comments
- The comments, the high and low-level algorithms, help you get it correct

6.3 Why is Goto Controversial?

- If you develop code randomly, with **goto** jumping all over the place, the program is hard to understand, unlikely to work, and difficult to debug
- This has given the **goto** statement a bad reputation
- But **goto** is essential for a compiler because it's essentially the jump instruction
- The compilation patterns provide a safe and systematic way to introduce the **goto** into the program
- But if you ignore the patterns, you lose these advantages
- Unstructured **goto** leads to complicated code

6.4 RECORDS

- A **record** contains several **fields**. Access a field with the dot (.) operator

```
;   x, y :
;     record
;       { fieldA : int;
;         fieldB : int;
;         fieldC : int;
;       }
;
;   x.fieldA := x.fieldB + x.fieldC;
;   y.fieldA := y.fieldB + y.fieldC;
```

- Some programming languages call it a **tuple** or **struct**

6.4.1 Defining records

```
; Data definitions

; The record x
x
x_fieldA   data   3    ; offset 0 from x  &x_fieldA = &x
x_fieldB   data   4    ; offset 1 from x  &x_fieldB = &x + 1
x_fieldC   data   5    ; offset 2 from x  &x_fieldC = &x + 2

; The record y
y
y_fieldA   data  20    ; offset 0 from y  &y_fieldA = &y
y_fieldB   data  21    ; offset 1 from y  &y_fieldB = &y + 1
y_fieldC   data  22    ; offset 2 from y  &y_fieldC = &y + 2

; In record x,  fieldA := fieldB + fieldC

; x.fieldA := x.fieldB + x.fieldC
    load   R1,x_fieldB[R0]
    load   R2,x_fieldC[R0]
    add    R1,R1,R2
    store  R1,x_fieldA[R0]
```

6.5 POINTERS

- So far, we've been finding a piece of data by giving it a label
- An alternative way to find the data is to make a **pointer** to it
- **A pointer is an address**
- **&x** means the address of **x**: a pointer to **x**. You can apply the & operator to a variable, but not to a complex expression
  - **&x** is okay
  - **&(3*x)** is not okay
- ***p** means the value that **p** points to. You can apply the * operator to any pointer

6.6 EXPRESSIONS USING POINTERS

- The & operator gives the address of its operand
  - **p := x** puts the value of **x** into **p**
  - **p := &x** puts the address of **x** into **p**
  - The address of **x** is called a pointer to **x**, we say "**p** points at **x**"
- The * operator follows a pointer and gives whatever it points to
  - ***p** is an expression whose value is whatever **p** points at
  - **y := p** stores **p** into **y**, so **y** is also now a pointer to **x**
  - **y := *p** follows the pointer **p**, gets the value (which is **x**) and stores that in **y**
- The & operator requires only one instruction: **lea**
- The * operator requires only one instruction: **load**

6.6.1 Flexibility of load and lea

- We have seen two ways to use **lea**:
  - To load a constant into a register: **lea R1,42[R0]** ; R1 := 42
  - To create a pointer: **lea R2,x[R0]** ; R2 := &x
- There are several ways to use **load**:
  - To load a variable into a register: **load R3,x[R0]** ; R3 := x
  - To access an array element: **load R4,a[R5]** ; R4 := a[R5]
  - To follow a pointer: **load R6,0[R7]** ; R6 := *R7
- Following a pointer to the address of **x** gives **x**
- For example, the value of ***(&x)** is just **x**

```
lea     R4,x[R0]     ; R4 := &x
load    R5,0[R4]     ; R5 := *(&x) = x

load    R6,x[R0]     ; R6 := x
```

6.7 Requests to the Operating System

- Many operations cannot be performed directly by a user program because
  - **User could violate system security**
  - Some operations are difficult to program
  - The code would need to change when OS is updated
- The program requests the operating system to perform them
- An OS request is performed by executing a **trap** instruction, such as **trap R1,R2,R3**
- A **trap** is a jump to the operating system (and you don't have to give the address to jump to)
- We use pointers to tell the operating system what to do

6.8 Typical OS Requests

- The type of request is a number, place in **R1**, and operands (if any) are in **R2, R3**
- The specific codes used to make a request are defined by the operating system, not by the hardware
- This is a major reason why compiled programs run only on one operating system
- Typical requests:
  - Terminate execution of the program
  - Read from a file
  - Write to a file
  - Allocate a block of memory

6.9 Termination

- A program cannot stop the machine; it requests the operating system to terminate it
- The operating system then removes the program from its tables of running programs, and reclaims any resources dedicated to the program
- In Sigma16, you request termination by **trap R0,R0,R0**

6.10 Write Operation in Sigma16

- To write a string of characters
- **trap R1,R2,R3**
  - **R1** – 2 is the code that indicates a write request
  - **R2** – address of the first word of string to write
  - **R3** – length of string (the last word should be newline character)

6.11 Character Strings

- A string like "the cat in the hat" is represented as an array of characters
- Each element of the array contains one character
- If you are writing a string to output, the last character of the string should be a **newline character**

6.12 Writing a String

- To write a string named **out**, we use:
  - **lea** to load a constant
  - **lea** to load the address of an array
  - **load** to get a variable

```
; write out (size = k)
     lea    R1,2[R0]          ; trap code: write
     lea    R2,animal[R0]     ; address of string to print
     load   R3,k[R0]          ; string size = k
     trap   R1,R2,R3          ; write out (size = k)

     trap   R0,R0,R0          ; terminate

k    data   4    ; length of animal
; animal = string "cat"
animal
     data   99   ; character code for 'c'
     data   97   ; character code for 'a'
     data  116   ; character code for 't'
     data   10   ; character code for newline
```

7 Procedures and the Call Stack

7.1 Procedures: Reusable Code

- Often, there is a sequence of instructions that comes up again and again
  - For example **sqrt** (square root)
  - It takes a lot of instructions to calculate a square root
  - An application program may need a square root in many different places
- Write the code **one time** – the block of code is called a procedure (or subroutine, function)
- Put the instructions off by themselves somewhere, not in the main flow of instructions
- Give the block of code a label (e.g work) that describes what it does
- Every time you need to perform this computation, call it: **goto work**
- When it finishes, the procedure needs to return: go back to the instruction after the one that jumped to it

7.2 Calling and Returning

- Here is a main program that calls a procedure **dowork** several times. It takes the value in **R1** and doubles it, and the main program would use the result but we ignore that

```
here    load     R1,x[R0]     ; R1 = x
        "goto"   dowork[R0]   ; call the procedure "dowork"
        load     R1,y[R0]     ; R1 = y
        "goto"   dowork[R0]   ; call dowork
        sub      R5,R6,R7     ; R5 – R6-R7


dowork  add      R1,R1,R1     ; R1 = R1+R1
        "return"              ; return — go back to the caller
```

7.3 Jump-and-link Instruction: JAL

- When the main program calls the subroutine, it needs to **remember where the call came from**
- This is the purpose of the **jal** instruction – jump and link
- **jal R5,dowork[R0]**
  - A pointer to the next instruction after the jal – the return address – is loaded into the destination register (e.g **R5**)
  - Then the machine jumps to the effective address

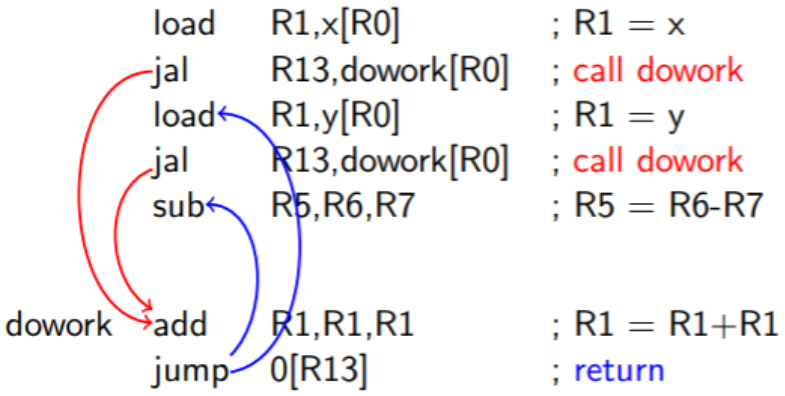
7.3.1 Jumping

- All jump instructions (**jump, jal, jumplt,** etc) refer to effective addresses
- **jump loop[R0]**
  - goto loop
- **jump 0[R14]**
  - goto instruction whose address is in **R14**
- **jump const[R2]**
  - goto instruction whose address is const+**R2**

7.4 Implementing Call and Return

- To call a procedure dowork**: jal R13,dowork[R0]**
  - The address of the instruction after the **jal** is placed in **R13**
  - The program jumps to the effective address, and the procedure starts executing
- To return when the procedure has finished: **jump 0[R13]**
  - The effective address is 0 + the address of the instruction after the **jal**
  - The program jumps there and the main program resumes

7.4.1 Calling with jal and returning with jump



7.5 Parameter Passage

- There are several different conventions for passing arguments to the function, and passing the result back
- What is important is that the called and the procedure agree on how information is passed between them
- If there is a small number of arguments, the caller may put them in register before calling the procedure
- If there are many arguments, the caller builds an array or vector (sequence of adjacent memory locations), puts the arguments into the vector, and passes the address of the vector in a register (typically **R1**)
- A simple convention: **the argument and result are passed in R1**

7.6 Functions

- A **function** is a procedure that
  - Receives a parameter (a word of data) from the caller
  - Calculates a result
  - Passes the result back to the caller when it returns
- A **pure function** is a function that doesn't do anything else – it doesn't change any global variables, or do any input/output

7.6.1 Passing arguments and result in R1

```
; Main program
        load  R1,x[R0]        ; arg = x
        jal   R13,work[R0]    ; result = work (x)
        ...
        load  R1,y[R0]        ; arg = y
        jal   R13,work[R0]    ; result = work (y)
        ...

; Function work (x) = 1 + 7*x
work    lea   R2,7[R0]        ; R7 = 2
        lea   R3,1[R0]        ; R3 = 1
        mul   R1,R1,R2        ; result = arg * 7
        add   R1,R3,R1        ; result = 1 + 7*arg
        jump  0[R13]          ; return
```

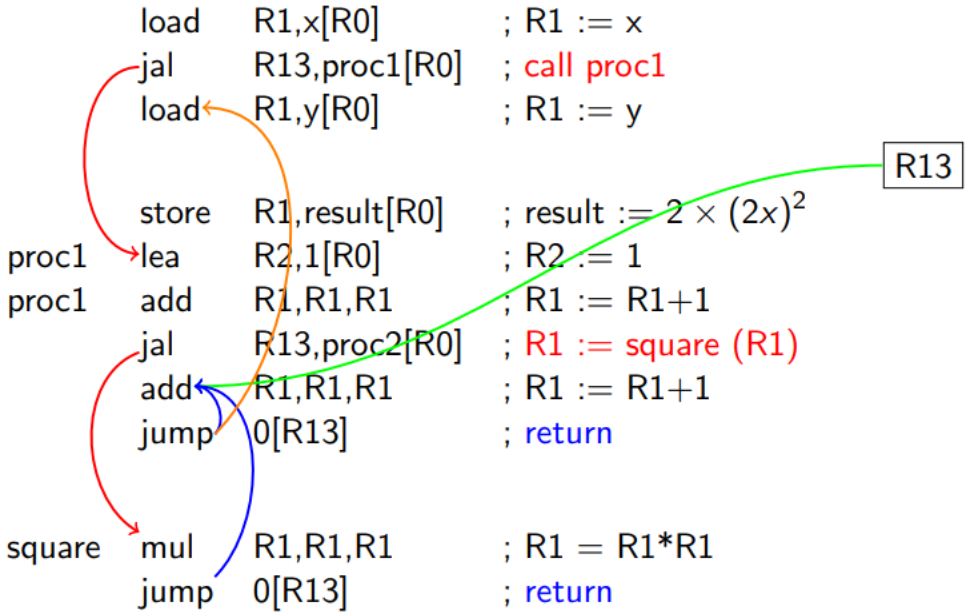
7.6.2 What if a procedure calls another procedure?

- The simplest kind of procedure
  - Call it with **jal R13,procname[R0]**
  - It returns by executing **jump 0[R13]**

7.7 Limitations of Basic Call

- If the procedure modifies any registers, it may destroy data belonging to the caller
- If the procedure calls another procedure, it can't use **R13** again. Each procedure would need a dedicated register for its return address, limiting the program to a small number of procedures
- The basic call mechanism doesn't allow a procedure to call itself (this is called **recursion**)

7.7.1 R13 overwritten: proc1 returns to the wrong place



7.8 Saving State

- Calling a procedure creates new information
  - The return address
  - Whatever values the procedure loads into the registers
- But this new information could overwrite essential information belonging to the caller
- We need to **save the caller's state** so the procedure won't destroy it

7.9 Incorrect Way to Save State

- Suppose we just have a variable **saveRetAdr**
- Store **R13** into it in the procedure, load that when we return
- Now it's ok for proc1 to call proc2
- But if proc2 calls proc3, we are back to the same problem: it doesn't work!
- The solution: a **stack**

7.10 Saving Registers

- Most procedures need to user several registers
- It's nearly impossible to do anything without using some registers
- The first thing a procedure should do is to save the registers it will use by copying them into memory (with **store** instructions)
- The last thing it should do before returning is to restore the registers by copying their values back from memory (with **load** instructions)

7.10.1 Where can registers be saved?

- It won't work to copy data from some of the registers to other registers
- It's essential to save the data into memory
- Two approaches:
  - Allocate fixed variables in memory to save the registers into – simple but doesn't allow recursion
  - Maintain a **stack** in memory, and **push** the data into the stack – this is the best approach and is used by most programming languages

7.10.2 Who saves the state?

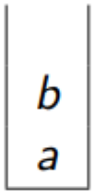
- Caller saves (used occasionally)
  - Before calling a procedure, the caller saves the registers, so all its essential data is in memory
  - After the procedure returns, the caller does whatever loads are needed
- Calle saves (usually the preferred method)
  - The caller keeps data in registers, and assumes that the procedure won't disturb it
  - The first thing the procedure does is to save the registers it needs into memory
  - Just before returning, the procedure restores the registers by loading the data from memory

7.11 Stack of Return Addresses

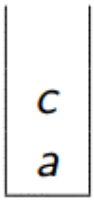
- To allow a large number of procedures, we can't dedicate a specific register to each one for its return address
- There we:
  - Always use the same register for the return address in a **jal** instruction (we will use **R13**)
  - The first thing a procedure does is to store its return address into memory
  - The last thing the procedure does is to load its return address and jump to it
  - The return addresses are pushed onto a **stack**, rather than being stored at a fixed address

7.12 STACKS

- A **stack** is a container
- Initially it is empty
- You can **push** a value onto the stack; this is now sitting on the top of the stack
- You can **pop** the stack; this removes the most recently pushed value and returns it
- A stack allows access only to the top value; you cannot access anything below the top
- We can save procedure return addresses on a stack because return always needs the most recently saved return address




```
2. Push return address b
1. Push return address a
```




```
2. Push return address c
1. Pop return address b
```

7.13 THE CALL STACK

- Central technique for
  - Preserving data during a procedure call
  - Holding most of your variables
- It goes by several names
  - Call stack
  - Execution stack
  - The stack
- Most programming languages use it
- Computers are designed to support it
- Often referrer to (stackoverflow site, etc)

7.14 STACK FRAMES

- There is a **call stack** or **execution stack** that maintains complete information about all procedure calls and returns
- Every activation of a procedure pushes a **stack frame**
- When the procedure returns, its stack frame is **popped** (removed) from the stack
- **R14** contains the address of the current (top) stack frame
- The stack frame contains
  - A pointer to the previous stack frame (this is required to make **pop** work)
  - The return address (saved value of **R13**)
  - The saved registers (so the procedure can use the registers without destroying information)
  - Local variables (so the procedure can have some memory of its own to use)

7.15 Implementing The Call Stack

- Dedicated **R14** to the **stack pointer**
- This is a programming convention, not a hardware feature
- When the program is started, **R14** will be set to point to an empty stack
- When a procedure is called, the saved state will be pushed onto the stack:
  - **Store a word at 0[R14] and add 1**
- When a procedure returns, it pops the stack and restores the state:
  - **Subtract 1, load from 0[R14]**
- The program should never modify **R14** apart from the **push** and **pop**

8 Variables

8.1 What is a Computer Program?

- The lines of source code are input to the assembler (compiler) which generates the initial value of the machine language
- When the program is booted, the initial machine language is stored in memory
- The computer executes the machine language instructions in memory; the original assembly language code (labels and all) no longer exists

8.2 What is a Variable?

- Variables are distinct from variable names: many variables may have the same name
- A variable has a scope in a program: a region where it corresponds to a particular box
- Variables do not correspond to data statements: they are created and destroyed dynamically as a program runs
- Initialising a variable is not the same as assigning a value to it

8.3 Access to Variables

- Depending on the programming language, the are several different ways that variables can be allocated
- For each of these, there is a corresponding way to access the variable in memory
- Three key issues:
  - The **lifetime** of a variable: when it is created, when it is destroyed
  - The **scope** of a variable: which parts of the source program are able to access the variable
  - The **location** of the variable: what its address in memory is
- The compiler generates the correct object code to access each variable

8.4 Three Classes of Variable

- **Static** variables (sometimes called **global** variables) – visible through the entire program
- **Local** variables (sometimes called **automatic** variables) – visible only in a local procedure
- **Dynamic** variables (sometimes called **heap** variables) – used in object oriented and functional languages

8.5 Static Variables

- The lifetime of a static variable is the entire execution of a program
  - When the program is launched, its static variables are created
  - They continue to exist, and to retain their values, until the program is terminated
- In C, you can declare a variable to be static. In Pascal, all global variables (i.e. all variables that aren't defined locally) are static
- So far, we have been using static variables

8.5.1 Combining static variables with code

- The simple way we have been defining variables makes them static
- These variables exist for the entire program execution. There is one variable **x**, and one variable **n**

```
        load  R1,x[R0]     ; R1 := x
    ...
        trap  R0,R0,R0     ; terminate

; Static variables

x   data    0
n   data  100
```

8.5.2 Disadvantages of combining variables and code

- The executable code cannot be shared
  - Suppose two users want to run the program
  - Each needs to have a copy of the entire object, which contains both the instructions and the data
  - That means the instructions are duplicated in memory
  - This is inefficient use of memory
- To avoid duplication of instruction, we need to separate the data from the code
- Modern operating systems organise information into **segments**
  - A **code segment** is read-only, and can be shared
  - A **data segment** is read/write, and cannot be shared

8.6 Local Variables

- Local variables are defined in a function, procedure, method, or in a begin…end block, or a {…} block
- A local variable has one name, but there may be many instances of it if the function is recursive
- Therefore, local variables cannot be stored in the static data segment
- They are kept in stack frames
- The compiler (or assembler) works out the address of each local variable relative to the address of the stack frame
- The variables are accessed using the stack frame register

8.6.1 Accessing local variables

- **load R1,x[R14]** ; access local variable x; R14 points to frame
- The compiler (or the programmer) works out the exact format of the stack frame
- Each local variable has a dedicated spot in the stack frame, and its address (relative to the frame) is used in the **load** instruction

8.7 Dynamic Variables

- A **dynamic variable** is created explicitly (e.g using *new* in Java)
- It is not limited to use in just one function
- The lifetime of a dynamic variables does not need to follow the order that stack frames are pushed or popped
- So dynamic variables can't be kept in the static data segment, and they can't be kept on the stack

8.8 The Heap

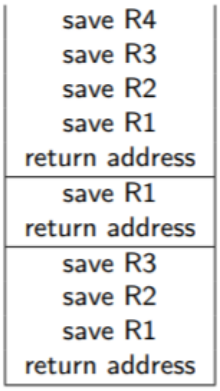
- Languages that support dynamic variables (List, Scheme, Haskell, Java) have a region of memory called the **heap**
- The heap typically contains a very large number of very small objects
- The heap contains a free space list, a data structure that points to all the free words of memory
- The heap is maintained by the language "runtime system", not by the operating system
- When you do a new, a (small) amount of memory is allocated from the heap and a pointer (address) to the object is returned
- When the object is no longer required, the memory used to hold it is linked back in to free space

8.9 The Call Stack

- Each procedure call pushes information on the stack
- The information needed by the procedure is in the stack frame (also called activation record)
- Each procedure return pops information off the stack
- A register is permanently used as the **stack pointer**
    - For each computer architecture, there is a standard register chosen to be the stack pointer
    - In Sigm16, R14 is the stack pointer
    - When you call, you push a new stack frame and increase **R14**
    - As a procedure runs, it accesses its data via **R14**
    - When you return, you set **R14** to the stack frame below

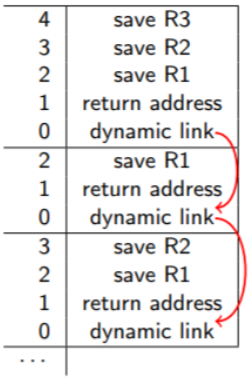
8.10 Saved Registers

- Save the registers the procedure needs to use on the stack and restore them before returning. This way, the procedure won't crash the caller



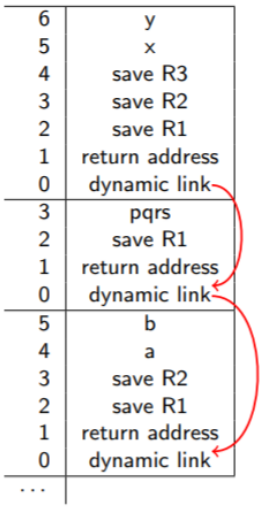
8.11 Dynamic Links

- Problem: since each activation record can have a different size, how do we pop the top one off the stack?
- Simplest solution: each activation record contains a pointer (called dynamic link) to the one below

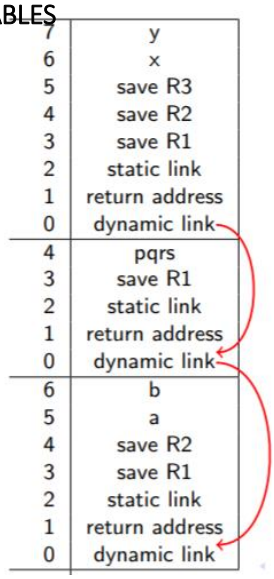


8.12 Local Variables

- The procedure keeps its local variables on the stack



8.13 Static Links for Scoped variables



8.14 Accessing a Word in the Stack Frame

- Work out a "map" showing the format of a stack frame
- Describe this in comments (it's similar to the register usage comments we've been using)
- Suppose the local variable **avocado** is kept at position 7 in the stack frame
- To access the variable:
    - **load R1,7[R14]** ; R1 := avacado
    - **store R1,7[R14]** ; R1 := avacado
    - Also, we can define the symbol "avacodo" to be 7, and write:
    - **load R1,avacado[R14]** ; R1 := avacado
    - **store R1,avacado[R14]** ; R1 := avacado

8.15 Example from Factorial Program

```
; Structure of stack frame for fact function
;    6[R14]   origin of next frame
;    5[R14]   save R4
;    4[R14]   save R3
;    3[R14]   save R2
;    2[R14]   save R1 (parameter n)
;    1[R14]   return address
;    0[R14]   pointer to previous stack frame
```

8.16 Recursive Factorial (factorial.asm)

- In the Sigma16 examples, there is a program called **factorial**
- This program illustrates the full stack frame technique
- It uses recursion – a function that calls itself
- *The best way to compute a factorial is with a simple loop, not with recursion*
- Recursion is an important technique, and it's better to study it with a simple example

8.16.1 About the factorial program

- Comments are used to identify the program, describe the algorithm, and document the data structures
- Blank lines and full-line comments organise the program into small sections
- The caller just uses **jal** to call the function
- The function is responsible for building the stack frame, saving and restoring registers
- The technique of using the stack for functions is general, and can be used for large scale programs

9 Nodes

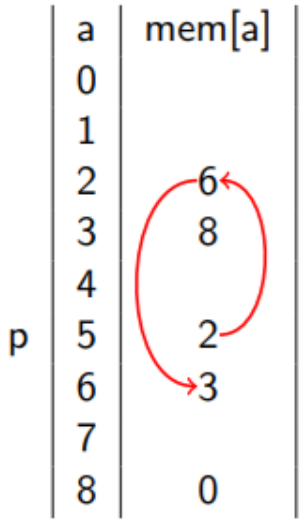
9.1 Nodes

- A linked list consists of a linear chain of nodes
- A node is a **record with two fields**
  - **value** is a word containing useful information, the content of the node may be an integer, character, or even a pointer to something else
  - **next** is a word containing a pointer to the next node in the list
- The last node in the list has a special **nil** in the next field
- **nil** is represented by 0 (so you can't have a pointer to memory location 0, but normally that's where the program will be, so you wouldn't want that anyway)

9.1.1 Accessing the fields of a node

- Suppose **p** is a pointer to a node
- **load R1,p[R0]** ; R1 := p
- **load R2,0[R1]** ; R2 := (*p).value
- **load R3,1[R1]** ; R3 := (*p).next

9.2 Representing a Linked List



9.3 Basic Operations on Lists

- Three key operations:
  - Is a list **p** empty?
  - What's the value in a node?
  - What's the next node?
- The following code assumes that all the pointer variables (**p, q**) are in memory, so they must be loaded and stored
- In practice, we often keep the pointers in registers so you don't need all those loads and stores

9.3.1 Is list p empty?

- Nil is 0, so the list that **p** points at is empty *iff (if and only if)* **p**=0
- Generally, it is unsafe to perform an action on a list **p** unless **p** actually points to a node, so this test is commonly needed

```
        load   R1,p[R0]
        cmpeq  R2,R1,R0
        jumpt  R2,pIsEmpty[R0]
; No, p is not empty
 ...
 ...
pIsEmpty
; Yes, p is empty
```

9.3.2 Get value in node that p points at: x := (*p).value

- **x := (*p).value**
- This is safe to do only if **p** is not empty
- The value field of a node is at offset 0 in the node record
- **load R1,p[R0]** ; R1 := p
- **load R2,0[R1]** ; R2 := (*p).value
- **store R2,x[R0]** ; x := (*p).value

9.3.3 Get pointer to next node in a list: q := (*p).next

- **q := (*p).next**
- This is safe to do only if **p** is not empty
- The next field of a node is at offset 1 in the node record
- **load R1,p[R0]** ; R1 := p
- **load R2,1[R1]** ; R2 := (*p).next
- **store R2,q[R0]** ; q := (*p).next

9.3.4 Traversing a list p

- A while loop is the best looping construct for traversing a list

```
ListSum (p)
  { sum := 0;
    while p /= nil do
      { x := (*p).value;
        sum := sum + x;
        p := (*p).next;
      }
```

9.3.5 Search a list p for a value x

- This is a good example of the proper use of a while loop
- The condition checks for end of data, and also for early completion
- There is no break statement or goto
- The plop works even if the original list **p** is nil

```
ListSearch (p, x)
  { found := False;
    while p /= nil && not found do
      { found := x = (*p).value;
        p := (*p).next;
      }
    return found;
  }
```

9.4 CONS

- Suppose **p = [23, 81, 62]**
- **q := cons(56, p)**
- After computing **q**, we have
  - **q = [56, 23, 81, 62]** – q is the same as p but with 56 attached to the front
  - **p = [23, 81, 62]** – p is unchanged
- **cons** creates a new list **q** with the **56** at the front of the new list, and leaves the old list **p** unchanged

9.5 Implementing Cons

- No change is made to **p**, or to the node **p** points to
- A new node is allocated and set to point **p**
- A pointer to the new node is returned
- A function like **cons** – which produces a new result but does not modify its arguments – is called a **pure function**

```
cons (x, p)
  { q := newnode ();
    (*q).value := x;
    (*q).next := p;
    return q;
  }
```

9.5.1 Getting a new node from avail list

- **avail** stands for "available" and means the heap

```
if avail = nil
  then { error "fatal error: out of heap" }
  else { newnode := avail;
         avail := (*avail).next;
         return newnode;
       }
```

9.5.2 Inserting a node with x where p points

- Notice that we can insert **x** after the node that **p** points to
- But we cannot insert **x** before that node
- It's common, in list algorithms, to have two pointers moving along through the list, one lagging an element behind the other, to make insertion possible

```
r := newnode ();
(*r).value := x;
(*r).next := (*p).next;
(*p).next := r;
```

9.6 List Header

- Suppose we have a list **p** and a value **x**
- We want to insert **x** into the list **p** at an arbitrary point
- Another pointer **q** points to the insertion position
- A slightly awkward problem: the code to insert **x** at the front of the list is slightly different from the code to insert **x** after some element **(*q)**
  - If somewhere in the middle, we can insert **x** after the node that **q** points to
  - The insertion algorithm will change **(*q).next**
  - But if we need to insert **x** at the beginning of the list, we cannot do that; instead the pointer **p** needs to be changed
- Solution: don't use an ordinary variable for **p**; make a **header node** whose next field points to the list

9.7 Deleting a Node

- Need a pointer **p** into the list; the node after **p** will be deleted
- Just change **(*p).next** to skip over the next node, and point to the one after
- The node being deleted should be returned to the **avail** list, so it can be reused

9.7.1 Code for deleting a node

- If **p** points to a node, delete the node after that, assuming it exists
- We can't delete the node **p** points to, we can only delete the following node, which **q** points at
- If you know that **p** cannot be nil, the first test can be omitted
- We do need to check whether **q = nil**; if it is, there's no node to delete
- It doesn't matter whether **(*q).next** is nil

9.8 Space Leaks

- IF you return a deleted node to the **avail** list, it can be reused
- If you don't, this node becomes inaccessible: it doesn't hold useful data, yet it can't be allocated
- **This is a bug in the program**
- Over time, as a program runs, more and more nodes may become inaccessible: a **space leak**

9.9 Memory Management

- It's a bug if you delete a node that contains useful data
- It's a bug if you don't delete a node that doesn't contain useful data
- With complicated structures, this can be difficult
- A common solution is **garbage collection**
    - The program doesn't explicitly return nodes to the **avail** list
    - Periodically, the **GC (garbage collector)** traverses all data structures and marks the nodes it finds
    - Then the **GC** adds all unmarked nodes to the **avail** list

9.10 Sharing and Side Effects

- Suppose **p = [6, 2, 19, 37, 41]**
- Traverse a few elements, and set **q** to point to the 19 node
- Now **q = [19, 37, 41]** and **p** is unchanged
- Then delete the second element of **q**. The result is
    - **q = [19, 41]**
    - **p = [6, 2, 19, 41]**
- This is called a **side effect**
- Sometimes you want this to happen, sometimes not, so it's important to be careful about it

9.11 Comparing Lists and Arrays

- Lists and arrays are two different kinds of data structure that contain a sequence of data values
- How do you decide which to use?
- Consider the properties of lists and arrays, and the needs of your program

9.12 Accessing Elements

- Direct access to an element
  - **Array:** gives direct access (random access) to element with arbitrary index **i**
  - **List:** gives direct access only to an element you have a pointer to; random access in inefficient
- Traversal
  - **Array:** initialise **i** to 0; repeatedly set **i := i+1**; terminate when **i >= n** (that's the purpose of a **for** loop)
  - **List:** initialise **p** to point to the list; repeatedly set **p := (*p).next**; terminate when **p = nil**

9.13 Usage of Memory

- Memory needed per element
  - **Array:** need just the memory required for the element itself (typically a word)
  - **List:** need a node for each element, which also requires space for the next pointer (typically a word)
  - So typically, an array with **n** elements needs **n** words, while a list requires 2 x **n** words
- Flexibility
  - An array has fixed size and needs to be allocated fully
  - A list has variable size and needs only enough memory to hold its nodes

9.14 More Data Structures

- We can put several pointer fields in each node, and produce an enormous variety of data structures, tailored for the needs of an application program
- Just a few examples
  - Double linked list: each node contains two pointers, one to the previous node and one to the next. Allows traversal both directions
  - Circular list: there is not last node where **next=nil**; instead, every node points to the next node, and the list loops back to itself. There is no first or last node

9.15 Abstract Data Type

- A stack is an **abstract data type**
  - The idea: define the type by the **operations** it supports, not by the code that implements it
  - This is useful because there may be different implementations of an ADT, and which implementation is best may depend on the application using it
- The stack ADT is defined by the **operations** it supports: **push, pop**
- There are several completely different ways to implement a stack
  - Array
  - Linked lists

9.15.1 Linked list implementation of stack

- A linked list gives easy access to the front of the list, and a stack gives access to the top of the stack
- Represent empty stack as nil
- **push x** is implemented by **stack := cons(x, stack)**
- **pop x** is implemented by **stack := (*stack).next**

9.15.2 Array representation of stack

- We can implement a stack using an array
- There is a variable **stLim** which gives the size of the array – this is the limit on the maximum number of elements that can be pushed
- There is a variable **stTop** that gives the current number of elements in the stack

9.16 Relationship Between Arrays and Stacks

- Array
  - A container that holds many elements
  - Each element has an index (which is an integer)
  - You can access any element **x[i]**
  - You can access the elements in any order
- Stack
  - A container that holds many elements
  - You can only access the top element, and you don't need to know its index
  - You can (and must) access the elements in **last in – first out** order

9.16.1 Pushing x onto a stack

```
; push the x onto the stack
; stack[stTop] := R1; stTop := stTop + 1

push   load  R1,x[R0]       ; R1 := x
       load  R2,stTop[R0]   ; R2 := stTop
       store R1,stack[R2]   ; stack[stTop] := x
       lea   R3,1[R0]       ; R3 := constant 1
       add   R2,R2,R3       ; R2 := stTop + 1
       store R2,stTop[R0]   ; stTop := stTop + 1
```

9.16.2 Pop a stack, returning x

```
; pop the stack, store top element into x
; stTop := stTop - 1; x := stack[stTop]

pop     load  R2,stTop[R0]    ; R2 := stTop
        lea   R3,1[R0]        ; R3 := constant 1
        sub   R2,R1,R3        ; R2 := stTop - 1
        load  R1,stack[R2]    ; R1 := stack[stTop-1]
        store R1,x[R0]        ; x := stack[stTop-1]
        store R2,stTop[R0]    ; stTop := stTop - 1
```

9.16.3 Issues with simplest implementation

- It doesn't check for errors
  - If **push** is called when stack is full, data will be written outside the array
  - If **pop** is called when the stack is empty, a garbage result will be returned
- Either of these errors may cause the program to get wrong answers or to crash

9.17 Robust Software

- Fragile software will respond to a minor problem by going haywire: might crash or produce wrong answers
- **Robust software** checks for all errors and does something appropriate; a minor problem doesn't turn into a major one

9.18 Error Checking and Error Handling

- Software should not assume everything is okay – it should check for errors
  - **push(x)** when the stack is full
  - **x := pop()** when the stack is empty
- If an error is detected, the error must be handled
- There are many approaches
  - Produce a message and terminate the program
  - Return an error code to the calling program and let it decide what to do
  - Throw an exception, which will interrupt the calling program, and invoke its error handler
- For simplicity, we will terminate the program if an error occurs

9.18.1 Error checking: push

- If the stack is full, there is no space to store the new element, so push fails

```
; push (v)
; if stTop >= stLim
;     then
;        terminate because the stack is full: cannot push
;     else
;        stack[stTop] := v
;        stTop := stTop + 1
;        return ()
```

9.18.2 Error checking: pop

- If the stack is empty, there is no element to return, so pop fails

```
; v = pop ()
;   if stTop == 0
;     then
;       terminate because the stack is empty: cannot pop
;     else
;       stTop := stTop - 1
;       v := stack[stTop]
;       return (v)
```

10 ARRAYS AND POINTERS

10.1 COMPOUND BOOLEAN EXPRESSIONS

```
i<n and x[i]>0      i<n && x[i]>0      i<n & x[i]>0
i<n or  j<n         i<n || j<n         i<n | x[i]>0
```

10.1.1 Short Circuit Expressions

- Suppose **x** is an array with **n** elements
- Consider **i<n && x[i]>0**
- If the first expression **i<n** is false, then the whole expression is false
- In that case, there is no need to evaluate the second expression **x[i]>0**
- We can "short circuit" the evaluation
- Big advantage: if **i<n** is false, then **x[i]** does not exist and evaluating it could cause an error
- So it is essential not to evaluate the second expression if the first one is false

10.1.2 Implementing a compound Boolean expression

```
while i<n && x[i]>0 do S

; if not (i<n && x[i]>0) then goto loopDone
    load   R1,i[R0]          ; R1 := i
    load   R2,n[R0]          ; R2 := n
    cmplt  R3,R1,R2          ; R3 := i<n
    jumpf  R3,loopDone[R0]   ; if not (i<n) then goto loopDon
    load   R3,x[R1]          ; R3 := x[i]   safe because i<n
    cmpgt  R4,R3,R0          ; R4 := x[i]>0
    jumpf  R4,loopDone[R0]   ; if not (x[i]>0) then goto loop
```

10.2 Condition Code

- We have seen one style for comparison and conditional jump
  - **cmpl R3,R8,R4**
  - **jumpt R3,someplace[R0]**
- There is also another way you can do it
  - **cmp R8,R4** ; no destination register
  - **jumplt someplace[R0]** ; jump if less than
- The **cmp** instruction sets a result (less than, equal, etc) in **R15** which is called the *condition code*
- There are conditional jumps for all the results: **jumpeq, jumplt, jumple,** etc
- An advantage is that you don't need to use a register for the Boolean result

10.3 Repeat-until Loop

- Similar to a while loop, except you decided whether to continue at the end of the loop
- The while loop is used far more often, but if you need to go through the loop at least one time, the repeat-until is useful

```
repeat
  {S1; S2; S3}
until i>n;
```

This is equivalent to

```
S1; S2; S3;
while not (i>n) do
  {S1; S2; S3}
```

10.4 Converting a Number to a String

- We need to do arithmetic to convert a binary number to decimal, and to a string of decimal digits
- It needs to divide the number by 10 to get the quotient and the remainder
- **div R1,R2,R3**
  - Divides **R2/R3**
  - The quotient goes into **R1** (destination)
  - The remained goes into **R15** (always R15, you cannot change this)
- The algorithm repeatedly divides the number by 10; the remained is used to get a digit character

10.5 Arrays and Pointers

- You can also access an array element by using pointers instead of indexes
- Create a pointer **p** to the beginning of the array **x**, so **p** is pointing to **x[0]**
  - **lea R1,x[R0]**
- To access the current element, follow **p**:
  - **load R2,0[R1]**
- To move on to the next element of the array, increment **p**
  - **lea R1,1[R1]**
- Notice that we are doing **arithmetic on pointers**

10.5.1 Sum of an array using pointers: high level

```
sum := 0;
p := &x;
q := &xEnd;
while p<q do
  { sum := sum + *p;
    p := p + 1; }
```

10.5.2 Sum of an array using pointers: assembly language

- In assembly language, we can use **lea** to increment the pointer
- Suppose **p** is in **R1**, then
  - **lea R1,1[R1]** ; p := p + 1
- We are incrementing **p** by the size of an array element

```
;   R1 = p = pointer to current element of array x
;   R2 = q = pointer to end of array x
;   R3 = sum of elements of array x

    lea   R1,x[R0]      ; p := &x
    lea   R2,xEnd[R0]   ; q := %xEnd
    add   R3,R0,R0      ; sum := 0
sumLoop
    cmplt R4,R1,R2      ; R4 := p<q
    jumpf sumLoopDone   ; if not p<q then goto sumLoopDone
    load  R4,0[R1]      ; R4 := *p  (this is current element of x)
    add   R3,R3,R4      ; sum := sum + *p
    lea   R1,1[R1]      ; p := p+1  (point to next element of x)
    jump  sumLoop[R0]   ; goto sumLoop
sumLoopEnd

x   data  23   ; first element of x
    data  42   ; next element of x
    data 19    ; last element of x
xEnd
```

10.5.3 Comparing the two approaches

- Accessing elements of an array suing index
  - Get **x[i]** with **load R5,x[R1]** where **R1=i**
  - Move to next element of array by **i := i+1**
  - Determine end of loop with **i < xSize**
  - Use a **for** loop
- Accessing elements of an array using pointer
  - Initialise p with **lea R1,x[R0]**
  - Get **x[i]** with **load R5,0[R1]** where **R1=p**
  - Move to next element of array by **p := p+1**
  - Determine end of loop with **p < q** (**q** points to end of array)
  - Use a **while** loop
- Both techniques are important
- If you have an array of records, it's easier to use a pointer

10.6 Records and Pointers

- Suppose we have an array of these records and want to
  - Set **fieldA := fieldB + fieldC** in every record in the array
  - Calculate the sum of the **fieldA** in every record

```
program Records
 { x, y :
     record
       { fieldA : int;
         fieldB : int;
         fieldC : int; }
```

10.6.1 Traverse array of records with indexing

```
sum := 0;
for i := 0 to nrecords do
  { RecordArray[i].fieldA :=
       RecordArray[i].fieldB + RecordArray[i].fieldC;
    sum := RecordArray[i].fieldA; }
```

- This is okay, but a little awkward

10.6.2 Traverse array of records with pointers: high level

- In professional programming, this is often preferred because accessing the elements of the records is easier (it's easier to access an "element of an element" via pointer)

```
sum := 0;
p := &RecordArray;
q := &RecordArrayEnd;
while p < q do
  { *p.fieldA := *p.fieldB + *p.fieldC;
    sum := sum + *p.fieldA;
    p := p + RecordSize; }
```

10.6.3 Traverse array of records with pointers: assembly

```
;    R1 = sum
;    R2 = p (pointer to current element)
;    R3 = q (pointer to end of array)
;    R4 = RecordSize

    lea    R1,0[R0]                ; sum := 0
    lea    R2,RecordArray[R0]      ; p := &RecordArray;
    lea    R3,RecordArrayEnd[R0]   ; q := &RecordArray;
    load   R4,RecordSize[R0]       ; R4 := RecordSize
RecordLoopDone
    cmplt  R5,R2,R3                ; R5 := p<q
    jumpf  R5,RecordLoopDone[R0]   ; if (p<q) = False then goto RecordLoop
    load   R5,1[R2]                ; R5 := *p.fieldB
    load   R6,2[R2]                ; R6 := *p.fieldC
    add    R7,R5,R6                ; R7 := *p.fieldB + *p.fieldC
    store  R7,0[R2]                ; *p.fieldA := *p.fieldB + *p.fieldC
    add    R1,R1,R7                ; sum := sum + *p.fieldA
    add    R2,R2,R4                ; p := p + RecordSize
    jump   RecordLoop[R0]          ; goto RecordLoop
```

10.7 Stack Overflow

- The mechanism for calling a procedure and returning is fairly complicated
- Rather than introducing all the details at once, we have looked at several versions, introducing the concepts one at a time
- Now we introduce the next level:
  - Simplifying calling a procedure
  - The procedure checks for stack overflow
- We need two more registers dedicated to procedures
  - **R12** holds stack top (the highest address in current stack frame)
  - **R11** holds stack limit (the stack is not allowed to grow beyond this address)

10.8 Register Usage

- See the **PrintIntegers** program for examples

REG	USAGE
R0	Holds constant 0
R1	Used for parameters and return values
R2	Used for parameters and return values
R3	Used for parameters and return values
R4	
R5	
R6	
R7	
R8	
R9	
R10	
R11	Stack limit
R12	Stack top
R13	Return address
R14	Stack pointer
R15	Transient condition code

10.9 Initialise the Stack

```
; Structure of stack frame for main program, frame size = 1
;       0[R14]      dynamic link is nil
; Initialise the stack
        lea   R14,CallStack[R0] ; initialise stack pointer
        store       R0,0[R14]         ; main program dynamic link = nil
        lea   R12,1[R14]        ; initialise stack top
        load  R1,StackSize[R0]  ; R1 := stack size
        add   R11,R14,R1        ; StackLimit := &CallStack + StackSize
```

10.10 Calling a Procedure

- To call a procedure **PROC**:
  - Place any parameters you're passing to **PROC** in **R1,R2,R3**
  - **jal R13,PROC[R0]**

10.10.1 Structure of procedure stack frame

- This is procedure **PrintInt** in lab exercise

```
; Arguments
;   R1 = x         = two's complement number to print
;   R2 = FieldSize = number of characters for print field
;        require FieldSize < FieldSizeLimit

; Structure of stack frame, frame size = 6
;    5[R14]  save R4
;    4[R14]  save R3
;    3[R14]  save R2 = argument fieldsize
;    2[R14]  save R1 = argument x
;    1[R14]  return address
;    0[R14]  dynamic link points to previous stack frame
```

10.10.2 Called procedure creates its stack frame

```
PrintInt
; Create stack frame
    store  R14,0[R12]          ; save dynamic link
    add    R14,R12,R0          ; stack pointer := stack top
    lea    R12,6[R14]          ; stack top := stack ptr + frame size
    cmp    R12,R11             ; stack top ~ stack limit
    jumpgt StackOverflow[R0]   ; if top>limit then goto stack
overflow
    store  R13,1[R14]          ; save return address
    store  R1,2[R14]           ; save R1
    store  R2,3[R14]           ; save R2
    store  R3,4[R14]           ; save R3
    store  R4,5[R14]           ; save R4
```

10.10.3 Procedure finishes and returns

```
; return
      load  R1,2[R14]   ; restore R1
      load  R2,3[R14]   ; restore R2
      load  R3,4[R14]   ; restore R3
      load  R13,1[R14]  ; restore return address
      load  R14,0[R14]  ; pop stack frame
      jump  0[R13]      ; return
```

10.11 Stack Overflow Example

```
StackOverflow
      lea    R1,2[R0]
      lea    R2,StackOverflowMessage[R0]
      lea    R3,15[R0]    ; string length
      trap   R1,R2,R3     ; print "Stack overflow\n"
      trap   R0,R0,R0     ; halt
StackOverflowMessage
      data   83     ; 'S'
      data   116    ; 't'
      data   97     ; 'a'
      data   99     ; 'c'
      data   107    ; 'k'
```

11 NESTED CONDITIONALS

11.1 BLOCKS

- A block or a compound statement is a single statement that contains several statements
- The purpose is to let you have a group of statements in a loop, or controlled by a conditional
- The syntax is the detailed punctuation used to indicate a block, and this varies in different languages

11.1.1 Enter at beginning, exit at end

- A common programming style is to require
  - Each block enters only at the beginning of the block
  - Each block exits only at the end of the block
- This style is helpful in some programming languages, but in some languages, it makes the code less readable
- In assembly language, and for compilation patterns, it is necessary to follow this style
- In high level languages this style is sometimes helpful, but not always

11.1.2 Single entrance/exit for compilation patterns

- It is straightforward to translate high level control structs using the compilation patterns
- These require that the blocks of code always start at the beginning and finish at the end
- It's bad to jump into the middle of a block, or exit out of the middle because
  - The compilation patterns won't work correctly
  - You'll have to duplicate a lot of code
  - Example: returning from a procedure requires storing the registers, resetting the stack pointer and loading the return address
  - That code should not be duplicated in several places in a procedure

11.2 SYSTEMATIC APPROACH TO PROGRAMMING

- Start by understanding what your program should do
- Express the algorithm using high level language notation (and it's ok to mix in some English too)
- Translate the high level to the low level
  - Assignment statements: **x := expression**
  - I/O statements: Write string
  - **Goto L**
  - **If Boolean then goto L**
- Translate the low level to assembly language
- Retain the high and low-level code as comments
- Do hand execution at every level

11.2.1 Why use this systematic approach to programming?

- It enables you to write correct code at the outset, and minimise debugging
- If there is a bug, it helps you to catch it early (e.g in translation to goto form)
- If there's a bug in an instruction, the comments enable you to find it quickly
  - A common error is to use a wrong register number: **add R9,R3,R4**
  - Poor comments don't help: **add R9,R3,R4** ; R9 := R3+R4
  - Good comments help a lot: **add R9,R3,R4** ; x := alpha + (a[i]*b[i])
  - Look at the register usage comments (oops, x is in **R8**, not **R9**, now I know how to fix it and I don't have to read the entire program)
- Professional software needs to be maintained; the comments make the software easier to read and more valuable

11.3 NESTED IF-THEN-ELSE

- Conditional statements can be nested deeply

```
if b1
  then S1
       if b2
         then S3
         else S4
       S5
  else S6
       if b3
         then S7
       S8
```

11.3.1 Special case of nested if-then-else

- Often, the nesting isn't random, but has this specific structure:

```
if b1
  then S1
  else if b2
         then S2
         else if b3
                then S3
                else if b4
                       then S4
                       else if b5
                              then S5
                              else S6
```

11.3.2 Another way to write it

- To avoid running off the right side of the window, it's usually indented like this:

```
if b1
  then S1
else if b2
  then S2
else if b3
  then S3
else if b4
  then S4
else if b5
  then S5
else S6
```

11.3.3 Some programming languages have elseif or elif

- It avoids ambiguity
- It signals to the compiler and to the programmer that this specific construct is being used
- It allows good indentation layout without violating the basic principle of indentation
- Some languages have this, some don't

```
if b1 then S1
  elif b2 then S2
  elif b3 then S3
  elif b4 then S4
  eleif b5 then S5
  else S6
```

11.3.4 A common application: numeric code

- Nested if-then-else but the Boolean conditions are not arbitrary: they are checking the value of a code:

```
if code = 0
  then S1
else if code = 1
  then S2
else if code = 2
  then S3
else if code = 3
  then S4
else if code = 4
  then S5
```

11.4 The Case Statement

```
case n of
   0 -> Stmt
   1 -> Stmt
   2 -> Stmt
   3 -> Stmt
   4 -> Stmt
   5 -> Stmt
   else -> Stmt // handle error
```

- This means: execute the statement corresponding to the value of **n**
- Many programming languages have this; the syntax varies a lot but that isn't what's important

11.4.1 Example: numeric code specifies a command

```
; The input data is an array of records, each specifying an operation
;    Command : record
;       code : Int     ; specify which operation to perform
;       i    : Int     ; index into array of lists
;       x    : Int     ; value of list element

; The meaning of a command depends on the code:
;    0  terminate the program
;    1  insert x into set[i]
;    2  delete x from set[i]
;    3  return 1 if set[i] contains x, otherwise 0
;    4  print the elements of set[i]
```

11.4.2 Selecting the command with a case statement

```
; Initialize
;    BuildHeap ()

; Execute the commands in the input data
;    finished := 0
;    while InputPtr <= InputEnd && not finished
;       CurrentCode := (*InputPtr).code
;       p := set[*InputPtr]     ; linked list
;       x := (*InputPtr).x      ; value to insert, delete, search

;       case CurrentCode of
;          0 : <CmdTerminate>
;          1 : <CmdInsert>
;          2 : <CmdDelete>
;          3 : <CmdSearch>
;          4 : <CmdPrint>
;          else : <>
;       InputPtr := InputPtr + sizeof(Command)
; Terminate the program
```

11.5 Finding a Numeric Code

- It's tedious and inefficient to go through the possible values of a numeric code in sequence
  - If you're looking up Dr Zhivago in the phone book, do you look at Arnold Aardvark, and Anne Anderson, and so on?
  - You go straight to the end of the book
- We want to find the statement corresponding to a numeric code directly, without checking all the other values

11.5.1 A problem with efficiency

- The problem
  - There are many applications of case statements
  - They are executed frequently
  - The number of cases can be large (not just 4 or 5; can be hundreds)
  - The implementation of the if-then-else requires a separate compare and jump for each condition
- The solution
  - A technique called **jump tables**

11.6 Jump Tables

- For each target statement (**S1, S2, S3,** etc) in the conditional, introduce a jump to it: **jump S1[R0], jump S2[R0], etc**
- Make an array **jt** of these jump instructions

```
jt[0] = jump S0[R0]
jt[1] = jump S1[R0]
jt[2] = jump S2[R0]
jt[3] = jump S3[R0]
jt[4] = jump S4[R0]
```

- Given the code, Jump to **jt[code]**
- Each element of the array is an instruction that requires two words
- So jump to **&jt + 2 * code**

11.6.1 Jump table in assembly language

```
; Jump to operation specified by code
    add    R4,R4,R4             ; code := 2*code
    lea    R5,CmdJumpTable[R0]  ; R5 := pointer to jump table
    add    R4,R5,R4             ; address of jump to operation
    jump   0[R4]                ; jump to jump to operation
CmdJumpTable
    jump   CmdTerminate[R0]     ; code 0: terminate the program
    jump   CmdInsert[R0]        ; code 1: insert
    jump   CmdDelete[R0]        ; code 2: delete
    jump   CmdSearch[R0]        ; code 3: search
    jump   CmdPrint[R0]         ; code 4: print

CmdDone
    load   R5,InputPtr[R0]
    lea    R6,3[R0]
    add    R5,R5,R6
    store  R5,InputPtr[R0]
    jump   CommandLoop[R0]
```

11.6.2 We have to be careful

- What if code is negative, or larger than the number of cases?
- The jump to the jump table could go anywhere
- It might not even go to an instruction
- But whatever is in memory at the place it goes to, will be interpreted as an instruction that will be executed
- The program will go haywire
- Debugging it will be hard: the only way to catch the error is to read the code and/or to single step
- Solution: before jumping into the jump table, check to see if code is invalid (too big or too small)

11.6.3 Checking whether the code is invalid

```
CommandLoop
    load   R5,InputPtr[R0]      ; R1 := InputPtr
    load   R4,0[R5]             ; R4 := *InputPtr.code
; Check for invalid code
    cmp    R4,R0                ; compare (*InputPtr).code, 0
    jumplt CmdDone[R0]          ; skip invalid code (negative)
    lea    R6,4[R0]             ; maximum valid code
    cmp    R4,R6                ; compare code with max valid code
    jumpgt CmdDone[R0]          ; skip invalid code (too large)

...

CmdDone
    load   R5,InputPtr[R0]
    lea    R6,3[R0]
    add    R5,R5,R6
    store  R5,InputPtr[R0]
    jump   CommandLoop[R0]
```

12 TREES

12.1 ORDERED LISTS

- There is an array of lists, initially empty. There are **nlists** of them

```
list[0] = [ ]
list[1] = [ ]
...
list[nlists-1] = [ ]
```

- At all times as the program runes, the lists are ordered: their elements are increasing

```
list[0] = [4, 9, 23, 51 ]
list[1] = [7, 102, 238 ]
...
list[nlists-1] = [2, 87, 89, 93, 103, 195 ]
```

12.1.1 Commands

- The program executes commands:
- Terminate – the program finishes
- Insert into list **i** the value **x** – modify **list[i]** so it contains **x**, while maintaining the ascending order
- Delete from list **i** the value **x** – modify **list[i]** so **x** is removed, but don't do anything if **x** isn't in the list
- Search list **i** for **x** – print Yes if **x** is in the list, No otherwise
- Print **i** – the numbers in **list[i]** are printed

12.1.2 Example

- Insert into **list[3]** the value 23 - **[23]**
- Insert into **list[3]** the value 6 - **[6,23]**
- Insert into **list[3]** the value 67 - **[6,23,67]**
- Insert into **list[3]** the value 19 - **[6,19,23,67]**
- Print **list[3]** **6 19 32 67**

12.1.3 Why are ordered lists useful?

- This is one way to arrange a database: think of the elements as person's names, or matriculation numbers
- Sometimes you want to process all the data in a container in a specified order
- If the data is ordered, it's faster to find a particular item (on average you only have to check half of the items)
- An ordered list can be used to represent a set

12.1.4 Where do commands come from?

- In a real application, we would read the commands from input
- But in this program, each command is represented as a record
- The entire input is a static array of records defined with data statements
- This is easier because
    - If you read from an input device, it's necessary to convert the input character string to numbers
    - In testing a program, it's convenient to have input data that is fixed and repeatable
    - Don't want to type in the same input every time you run the program

12.2 Representing a Command

- Each command is a record with three fields
    - A code indicating which kind of command
    - A number **i** indicating which list we're operating on
    - A value **x** which might be inserted etc
- Each record must have these three fields
- Some commands don't use them all (e.g Print just needs **i** not **x**)
- The main program uses a **case** statement to handle each commands, and implements this with a **jump table**

12.3 Reading a Program Before Writing

- You should read and understand the program before modifying it
    - Reading a program is an important skill you will need throughout your career
    - The program is filled with examples so it is excellent revision material
    - You need to understand a program before you'll be able to make changes to it
- One of the aims of the exercise is to get experience with reading a longer program – don't skip this!

12.4 Tips on Testing and Debugging

- Debugging has two phases:
    - Diagnosis: finding out what went wrong and why
    - Correction: fixing the error
- The most important point: don't just make random changes to the code and hope for the best – instead, find out what the error is and fix it cleanly

12.5 Reading and Testing a Program

- A good way to understand a section of assembly language instructions is to step through it, one instruction at a time
    - Check that the instruction did what you expected it to od
    - Check that the instruction is consistent with its comment
    - Try to relate the instruction with the bigger picture: what is it doing in the context of the program?

12.5.1 Coverage

- You don't need to step through a set of instructions a huge number of times
- If there's a loop, step through two or three iterations
- If possible, arrange test data so the loop will terminate after just a few iterations
- But try to step through as much of the program as possible
- This is called **coverage**: try to cover all of the program with your testing

12.6 BREAKPOINTS

- It's a good idea to step through a program one instruction at a time, so you understand clearly what each instruction is doing
- However, in a longer program this isn't always feasible
  - The OrderedLists program has to build the heap when it starts; this may take several thousand instructions before it even really gets going
- Solution: **breakpoints**
  - Find the address of an instruction where you want to start single stepping
  - Enter this address as a breakpoint
  - Click Run to execute the program at full speed; when it reaches the breakpoint it will stop
  - Then you can single step to examine what the instructions are doing

12.6.1 How to set a breakpoint

- On the Processor pane, click Breakpoint. It will say "Breakpoint is off"
- Enter the breakpoint command and click Set Breakpoint
- **BPeq BPpc (BPhex "01a6")**
- It will say "Breakpoint is on". Click Close
- On Processor, click Run. It will stop when the **pc** register gets the value you specified

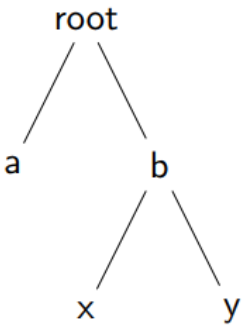
12.7 TREE

- A node doesn't have to have two fields named **value** and **next** – it's normal to define a specific node type for an application program
- Nodes with **value** and **next** can be connected in a **linked list**
- Nodes can also have several fields containing data, not just one **value** field
- And a node can have several pointer fields
- Common case: a **binary tree** has two pointers in each node, named **left** and **right**
- Each of these can either contain nil, or point to another node

```
Node : record
  value    ; the actual data in the node
  left     ; left subtree is a pointer to a Node
  right    ;  right subtree is a pointer to a Node
```

- Similar to a node for a linked list, but with two pointers
- There can also be several fields for data, not just one **value** field
- And we could have more than just two pointers

12.8 Binary Tree



- In computer science, we draw trees upside down (no idea why)
- Suppose **p** is a pointer to the tree
  - **(*p).left** is the pointer to the left subtree
  - **(*p).right** is the pointer to the right subtree

12.9 Applications of Trees

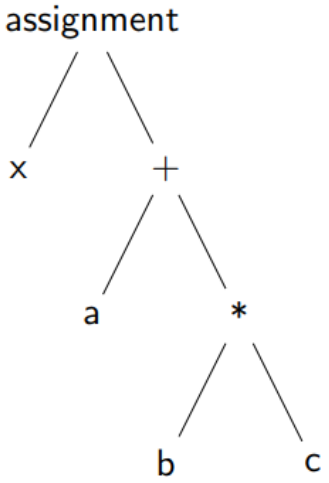
- Trees are used everywhere in programming
  - To hold structured data
  - To make programs much faster

12.10 Holding Structured Data

- A compiler reads in program text, which is just a character string: a sequence of characters
- It needs to represent the deep structure underlying that sequence of characters
- This is done by building a tree (the part of a compiler that takes a character string and produces a tree is called the **parser**)

12.11 Parsing

```
x := a + b * c
```



12.12 Another Application of Jump Tables

- In complicated applications, trees normally have *several different types of node*
- Examples: operations with 1 operand; operations with 2 operands; control constructs with a boolean expression and two statements, etc
- So there are several different kinds of record
- Each record has a code in the first word
- The value of the code determines how many more words there are in the record, and what they mean
- When a program has a pointer to a node, it needs to examine the code and take different actions depending on what the code is
- This is done with a jump table

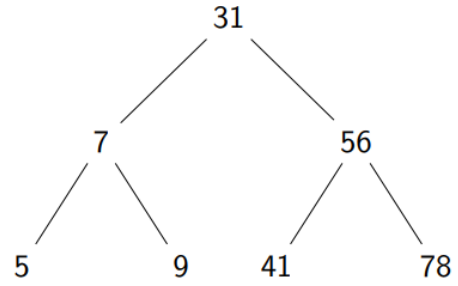
12.13 Searching

- Suppose we have a large number of records (e.g a database)
- We want to search the database for an entry where a field has a certain value (e.g search for a record where the MatricNumber field is 123456)
- If you just have these records in an array, or a linked list, you have to search them one by one
- On average, you have to look at half the entries in the database to find the one you want
- If you double the size of the database, you double the average time to look up and entry
- Terminology: this is called linear time or O(n) complexity

12.14 Better Approach

- Linear search is silly if you can place the records in order
- You're trying to find the telephone number of John Smith in the phone book
- Open the book to the middle, notice that S is in the second half
- Open the book to the middle of the second half
- Each time you look at an entry in the book, you discard half of the remaining possibilities

12.15 Binary Search Tree



- At every level: if a node contains **x**, then
  - Every node in the left subtree is less than **x**
  - Every node in the right subtree is greater than **x**
- You can search the tree by starting at the root, and at every step you know whether to go left or right

12.16 Algorithmic Complexity

- Complexity is concerned with how the execution time grows as the size of the input grows
- This is expressed as a function of the input size **n**
- Normally we don't care about the exact function, and we use O-notation. Instead of a function like f(n) = 4.823 x **n**, we just write **f(n) = O(n)**
    - **O(1)** – if the input grows, the execution time remains unchanged. This is unrealistic: the program cannot even look at the input
    - **O(n)** – if the input is **5** times bigger, the execution time is **5** times bigger. This is the best you can hope for
    - **O(n^2)** – if the input is **5** times bigger, the time is **25** times bigger

12.16.1 Algorithm is more important than small optimisation

- Some programmers spend lots of effort trying to save one or two instructions in a piece of a program
    - But it doesn't matter much whether a program takes 2.00032 seconds or 2.00031 seconds
- It's much more important to use a suitable **algorithm**
    - On small data it doesn't make much difference
    - On large (realistic) data, a better algorithm makes a huge difference

12.17 Complexity for Search

- Ordered lists
    - The Ordered Lists program has an operation to search a list for a value **x**
    - On average, you need to look through half of the data to find out whether **x** is present
    - So the ordered list makes the search about twice as fast
    - But in either case, this is **O(n)** – if you double the data size, the average time is doubled
- Binary search tree
    - The number of comparisons needed is roughly the same height of the tree
    - If the tree is balanced, the time complexity is **O(log *n*)**

12.17.1 How much faster?

- With a linear data structures (array, linked list)
    - Each time you compare a database entry with your key, you eliminate one possibility
    - The time is proportional to the size of the database
    - It's called **linear time** – time = **O(n)**
    - For 2 million records, you need a million comparisons
- With a binary search tree
    - Each time you compare a database entry with your key, you eliminate (on average) half of the possibilities
    - The time is proportional to the logarithm of the size of the database
    - It's called **log time** – time = **O(log *n*)**
    - For 2 million records, you need 21 comparisons
    - There's a saying "**log comes from trees**"

12.17.2 A common pitfall

- When you're writing a program, it's natural to test it with small data
- Even if the algorithm has bad complexity, the testing may be fast
- But then, when you run the program on real data, the execution time is intolerable
- That means going back and starting over again
- So it's a good idea to be aware of the complexity of your algorithm from the beginning

12.17.3 How bad can complexity be?

- Order of magnitude estimate of time for input of size **n**

n	$\log n$	$n \log n$	n^2	2^n
1	1	1	1	2
10	3	30	100	1,000
100	7	700	10,000	1267650600228229401496703205376
1,000	10	10,000	1,000,000	> age of universe

- Lots of real problems have data size larger than 1,000
- Lots of algorithms have exponential complexity: 2^n

13 INTERRUPTS

13.1 INTERRUPTS

- The hardware provides interrupts which are used to implement processes
- An interrupt is an **automatic jump**
- It goes either to the operating system or to an error handler
- But it is not the result of a jump instruction – it happens automatically when an external event occurs
- The program that was running never jumped to the OS – the processor just stops executing its instructions, and starts executing the OS instead
- It's like talking to a group of people, and suddenly you get interrupted

13.1.1 What causes an interrupt?

- An error in a user program: e.g overflow (result of arithmetic is too large to fit in a register)
- A trap: this is an explicit jump to the operating system, but the program doesn't specify the address
- An external event: a disk drive needs attention right now, or the timer goes off

13.1.2 What happens when an interrupt occurs?

- The computer is a digital circuit
- Without interrupts, it repeats forever
  - Fetch the instruction at the address in the **pc** register
  - Execute the instruction
- With interrupts, it repeats this forever:
  - Check to see if there is an interrupt request
  - If there is, **savepc := pc**, **pc := address of code in operating system**
  - Fetch the instruction at the address in the **pc** register
  - Execute the instruction
- Since the **pc** has been modified, the next instruction will not be part of the program that was interrupted – it will be the operating system

13.1.3 Saving state

- Remember, an interrupt is a jump to the OS
- This requires setting the address of OS in the **pc** register
- But if we simply assign a value to **pc**, the computer has forgotten where the interrupted program was
- Therefore, the hardware must save state: **savepc** := **pc**
- The OS has a special instruction that enables it to get the value of **savepc**

13.1.4 How interrupts are used

- Interrupts can be used to **catch errors** in a program, e.g arithmetic overflow
  - If an error occurs we want the program to jump to an error handler
- Trap is similar to an interrupt, and is used to **request service** from the OS
  - Use trap to ask the OS to stop running the program
- They can be used to provide **quick service** to an input/output device
  - A disk drive may generate an interrupt when the spinning platter reaches a certain point, and it needs service right away – within a tight deadline
- Interrupts are used to implement **concurrent processes**
  - The OS gives each process a **time slice** in round-robin order, so each process makes progress

13.2 Interrupts and Programming Languages

- Most programming languages don't provide the ability to work directly with interrupts
- But many programming languages provide **exceptions**
  - Without an exception handler, a division by 0 might terminate the program
  - In the program, you can set an exception handler: a procedure to execute if a division by 0 occurs
  - The compiler might implement this in several different ways:
    - It could put in explicit comparison and conditional jumps to check each division
    - Or it could set up an interrupt handlers (this requires negotiation with the OS)

13.3 Catching Errors

- As a program runs, it may accidentally produce an error
- Two examples:
  - An arithmetic instruction produces a result that's too large to fit in a register
  - A divide instruction attempted to divide by 0
- It's better to detect the error and do something about it
- This makes software robust
- If the program just keeps going, it's likely to produce wrong results and it won't tell
- Two approaches for catching errors (use one or the other):
  - Explicit error checking
  - Interrupts

13.4 Explicit Error Checking

- Most computers have a condition code register with a bit indicating each kind of error
- Sigma16 uses **R15**, and a bit in **R15** indicates whether overflow occurred
- Every time you do an arithmetic instruction, that bit is set to 0 if it was okay, and 1 if there was an overflow
- You can check for this with a conditional jump, and then take appropriate action
- Of course, you have to decide what the appropriate action is

```
add      R2,R5,R4      ; x := a + b
jumpovfl TooBig[R0]    ; if overflow then goto TooBig
```

13.4.1 Problems with explicit error checking

- You have to put in the **jumpovfl** after every arithmetic instruction
- This makes the program considerable longer
- It's also inefficient: those conditional jumps take time
- It is "fragile": if you forget the **jumpovfl** even once in a big program, that program can malfunction

13.4.2 A better approach: interrupts

- Most computers (including Sigma16) can also perform an interrupt if an overflow (or other error) occurs
- The digital circuit checks for overflow (or other error) after every arithmetic operation
- If the error occurred, the circuit performs an interrupt
- The OS then decides what to do
- User program can tell the OS in advance "in case of overflow, don't kill me, but jump to this address: TooBig"
- There is a special trap code for making this request
- In some programming languages, this is called "setting an exception handler" or "catching exceptions"
- There is a special control register with a bit that specifies whether overflow should trigger an interrupt

13.4.3 Why are interrupts better than explicit checking?

- Interrupts guarantee that every operation is checked
- It is faster: the circuit can do this checking with essentially zero overhead
- It is easier: the programmer doesn't have to worry about it
- The program is shorter: don't need a jump after every arithmetic instruction

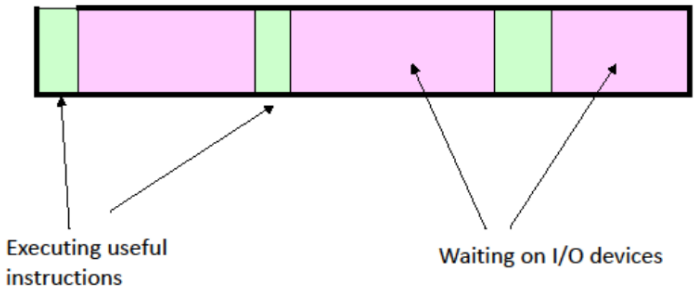
13.5 Interrupts and Processes

- One of the central features provided by an operating system is concurrent processes
  - A process is a running program
  - Think of a program as a document: it's just sitting there
  - A process is all the action that happens when a program is executed: it has its variables, the variables change over time, input/output happens, …
  - Several different processes may be running on the same program (e.g multiple tabs on a web browser)
  - Each process has its own variables
- Processes are implemented using interrupts
- The idea: the OS gives a user program a time slice
- The user is interrupted, and the OS can then run a different program

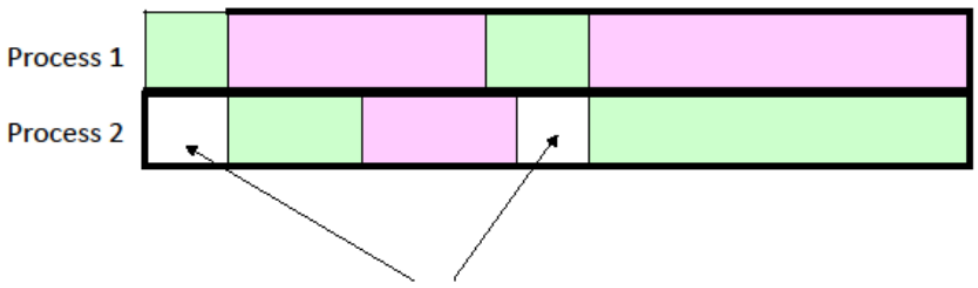
13.5.1 Waiting for I/O = wasted time

- Motivation for processes comes from I/O
- The problem:
  - Instructions execute quickly – typically about 0.3ns per second
  - I/O is much slower, especially if mechanical devices are involved
  - An I/O operation runs slower than an instruction by a factor ranging from 10^4 to 10^8
  - For comparison, a supersonic jet fighter is only 10^3 times faster than a turtle
- If a program does not **compute ⋯ print ⋯ computer** it is likely to spend a lot of time waiting for the I/O

13.5.2 A process must sometimes wait



13.5.3 Don't wait – switch to another process



***Ready**—not actually executing, but ready to go when the processor becomes available*

13.5.4 Don't wait- switch to another program

- When a program needs to perform I/O, it
  - Requests the operating system to do the I/O
  - The OS starts the I/O but doesn't wait for it to finish
  - The OS then allows a different program to run for a while
  - Eventually, when the I/O operation finishes, the OS allows the original program to resume
- This leads to an operating system running a large number of separate programs
- Each running program is called a process

13.6 Concurrent Processes

- A process is a running program
- At an instant of time, the computer is physically executing just one instruction (which belongs to one process)
- From time to time (around 100 or more times per second), the system will transfer control from one process to another one – this is called a **process break**
- Time scale:
  - At the scale of a nanosecond the computer is executing just one instruction belonging to a process; all other processes are doing nothing
  - At the scale of human perception (10^{-2} seconds) it appears that all the processes are making smooth process
- A motion picture is just a sequence of still photographs but displaying them rapidly gives the impression of continuous motion

13.7 Operating System Kernel

- A process does not transfer control to another process
- How could it? When you write a program, you don't know what other programs will be running when this one is
- A process break means
  - Running process jumps to the operating system kernel
  - The kernel is the innermost, core, central part of the OS
  - The kernel has a table of all the processes
  - (On Windows: check *Processes* tab in Task Manager)
  - The kernel chooses another process to run and jumps to it

13.8 Events that can trigger an Interrupt

- There is a timer that "bings" periodically – each time it goes off it generates an interrupt
- When an I/O device has competed a read or write, it generates an interrupt

13.9 Interrupts and Pre-emptive Scheduling

- When the operating system jumps to a user process, it sets a **timer** which will "go off" after a set amount of time (e.g 1ms)
- When does a running process jump to the operating system?
    - When the timer goes off
    - When the process makes an I/O request
    - This guarantees that the process won't run forever and freeze the system even if it goes into an infinite loop

13.10 The Scheduler

- The core of an operating system is the scheduler
- It maintains a list of all processes
- When an interrupt occurs:
    - The process that was running stops executing instructions: it has been interrupted
    - The OS takes any necessary action (e.g service the I/O device)
    - Then the OS jumps to the scheduler
    - The scheduler chooses a different process to run
    - It sets the timer and jumps to that process

13.11 Mouse

- The mouse isn't connected to the cursor on the screen
- When you move the mouse, it generates an interrupt
- The OS reads the mouse movements then calculates where the cursor should be and redraws it
- This happens many times per second, giving the illusion of a smooth movement

13.11.1 How interrupts are implemented

- Interrupts cannot be implemented in software
- The processor repeatedly goes through a sequence of steps to execute instructions
- This is the control algorithm and it's performed by a digital circuit in the processor (the control circuit)
- Interrupts are implemented by the control circuit

13.12 Control

- We have seen the **RTM**, which executes operations like **reg[d] := reg[a] + reg[b]**
- This is the core of a processor
- We have seen the control registers: **pc, ir, adr**
- The processor uses these to keep track of what it is doing

13.13 The Control Algorithm

- The behaviour of the entire processor is defined by a control algorithm
- We can describe this using a special notation (which looks like a simple programming language, but it is not a program)
- We can implement the control algorithm using flip flops and logic gates

13.14 Registers

- **pc** (program counter) contains address of the next instruction
- **ir** (instruction register) contains the current instruction (or first word of an RX instruction)
- **adr** (address register) holds the effective address for RX instructions
- **reg[a]** (register file) contains 16 registers for use by user program

13.15 Notation

- **pc, ir, adr** – contents of these 16-bit registers
- **ir_op, ir_d, ir_a, ir_d** – 4-bit fields in the **ir**
- **reg[x]** – the register in the register file with address **x**
- **mem[x]** – the memory location with address **x**

13.16 Infinite Loop

- In hardware, we need infinite loops
- The computer should never stop executing instructions

13.17 Case Dispatch

- We often have an operation code – a binary number, with **k** bits (e.g 4 bits)
- There are 2^k alternative actions to take, depending on the value of the code

```
case opcode
  0: action
  1: action
  ...
  15: action
```

14 Programming Languages

14.1 Syntax, Semantics, Compilation

- Syntax is the **form** of a program
- Semantics is the **meaning** of the program
- Compilation (or interpretation) is **how the language is implemented** so it can be run on a computer

14.2 Syntax

- The rules are clear cut
- If in doubt, just look it up
- Example: various languages have different names for the same thing: *Bool, Boolean, Logical.* These differences are superficial

14.3 Syntax Errors

- Compilers insist that the syntax is right
  - In English, if you spell a word wrong or have a missing comma you'll (probably) still be understood
- Can't a compiler be equally forgiving?
  - There were experiments with compilers that guess what the programmer meant
  - It was a disaster: the compiler nearly always guessed correctly, but occasionally it would guess wrong
    - How can you debug a program when the compiler didn't translate what you wrote, but something different?
    - You have to debug code that is not in the file and you cannot see it
    - You want the compiler to insist that the syntax is correct

14.3.1 Example of syntax: operator precedence

- Expressions can contain many operations, but the computer can do only one operation at a time
- We can make the operations explicit by using parentheses around each operation
- You don't have to write the parentheses, but the compiler needs to know where they go
- **a + b + c** is parsed as **(a + b) + c**
- **a + b * c** is parsed as **a + (b * c)**

14.4 Ambiguity

- A language is ambiguous if a sentence in the language can have two different meanings
- English is full of ambiguity
- Programming languages are designed to avoid ambiguity, most of the time
- If ambiguity is possible, the compiler needs to know how to resolve it, and so does the programmer

14.4.1 Ambiguity in if-then-else

- This is ambiguous: **which if does the "else S2" belong to?**

```
if b1 then if b2 then S1 else S2
```

- There are two interpretations, and they lead to different results

```
if b1 then { if b2 then S1 else S2 }
  b1 = true,  b2 = true     S1
  b1 = true,  b2 = false    S2
  b1 = false, b2 = true
  b1 = false, b2 = false

if b1 then { if b2 then S1 } else S2
  b1 = true,  b2 = true     S1
  b1 = true,  b2 = false
  b1 = false, b2 = true     S2
  b1 = false, b2 = false    S2
```

14.4.2 How does Python prevent ambiguity?

- The structure of the program is determined by indentation
  - This means it is essential to indent the program properly
  - It makes the program structure highly visible to the programmer
- Some languages use braces to indicate structure (e.g C, Java)
  - The compiler ignores the indentation, and uses the braces
  - Programmers tend to focus on the indentation and may overlook the braces
  - This is more error-prone

14.5 GOTO

- Usually programs are more readable and reliable if written with while loops, if -then-elif-else, for loops, and similar high-level constructs
- Programs that jump around randomly with goto statements are harder to understand, and likely to contain bugs
- This gave the goto statement a bad reputation
- But the goto statement is simply a jump instruction, and it is essential for use at low-level
- There are some circumstances where goto may be the best solution, but these are rare

14.5.1 Goto spelled differently

- Some programming languages provide restricted forms of goto
  - **break**
  - **continue**
- These are goto statements without a label, but with a predefined destination they go to
  - Advantage – you may recognise the pattern being used by a programmer
  - Disadvantage – you need to know where the goto goes

14.5.2 The break statement

- C and its descendants have **break**, as does Python
- Break is a goto that goes to the end of the innermost loop it's in
- In Python, you can have an else clause to execute when a for or while loop finishes, but this is skipped if you terminate the loop with break
- What if you want to break out of several loops?
  - There is no good way to do this in Python
  - It's best to restructure your program to avoid break

14.5.3 The continue statement

- In C and Python, **continue** goes to the end of the current loop which continues executing
- It's a way of staying in the loop, but skipping the statements after the continue
- **Warning:** several programming languages have a statement that is spelled continue – but it does nothing and is equivalent to the **pass** statement in Python

14.6 SEMANTICS

- Semantics means the meaning
- The semantics of a language is what a program in the language means
- The semantics of a program is the meaning of the program
    - Given its inputs, what are its outputs?

14.6.1 How are the semantics of a language defined?

- Using natural language to describe it
    - Vague description in English of what each construct does
    - A carefully written description in English, written to be as precise as possible
- Using mathematics or program transformation
    - Denotational semantics: a precise mathematical specification. Given a program, it gives a mathematical function from program inputs to outputs
    - Operational semantics: gives a sequence of reduction rules that give a precise model of the program's execution
    - Transformational semantics: a translation from a program into a simpler language, where the semantics is assumed to be clear

14.6.2 Why do the translation from high to low level?

- This is the semantics of the high-level constructs
- It explains precisely what the high-level means
- It shows what the compiler will do with the program (compilers do intermediate level translations like this; many compilers use several intermediate steps)
- It makes the execution time of the construct explicit
- The low level is very close to assembly language
- It's easier to go from high level to assembly language in two small steps, rather than one giant leap

14.6.3 How do we figure out semantics problems?

- Need to understand all the fundamental concepts of the language
- It's best to study **authoritative** source **systematically**
- Develop a clear **model** for what the language constructs really mean

14.6.4 Are two nodes with the same value identical?

- Define **b = a** – this is pointer assignment
- **a = [1, 2, 3, 4]**
- **b = a**
- The values of **a** and **b** are simply pointers, and they point to the **same** node

14.6.5 Low-level list manipulation

- You can assign pointers; if **a** is a list, **b = a** makes **b** point to the same node **a** points to

```
; b = a,    where a is a list
    load   R1,a[R0]     ; R1 = a
    store  R1,b[R0]     ; b := a
```

- You can write expressions that create a new list, but don't modify any existing lists: **a = a + b**
- You can modify an existing list structure:
  - **a.copy()**
  - **a.append(b)**
  - **a.extend(b)**
- All of these are implemented with while loops that traverse **a**

14.6.6 b = a.copy()

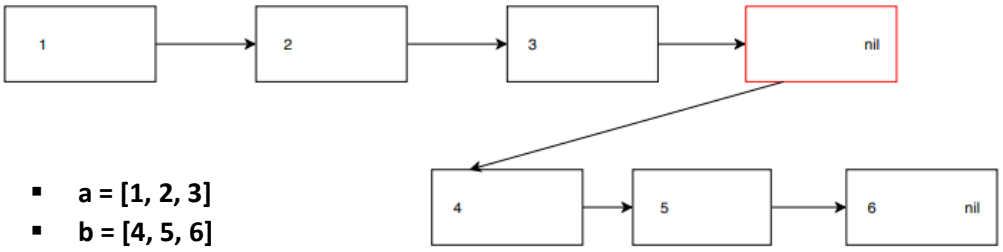
- Traverse **a** and make a new node for each node in **a**; link the new nodes together to form the result. The nodes in **b** have the same values as the nodes in **a**, but they are distinct nodes

```
p := a
b := nil
while p /= nil do
    nn := newnode()
    *nn.value := *p.value
    *nn.next := nil
    *b.next := nn
```

- If you modify one of the lists (**a, b**) the other is not affected

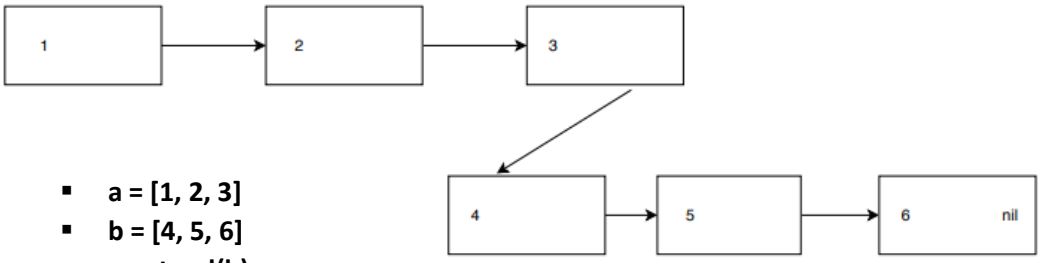
14.6.7 List represented as nodes

14.6.8 Appending to a list



- **a = [1, 2, 3]**
- **b = [4, 5, 6]**
- **a.append(b)**
- A new node is created: call **newnode()**
- The **newnode** has value = **b** and next=**nil**
- The last node in **a** had next = **nil**; that is changed to point to the new node
- Result: **a = [1, 2, 3, [4, 5, 6]]**

14.6.9 Extending a list



- **a = [1, 2, 3]**
- **b = [4, 5, 6]**
- **a.extend(b)**
- The last node in **a** had next=**nil**
- That nil pointer was changed to point to **b**
- Result: **a = [1, 2, 3, 4, 5, 6]**

14.6.10 For loops in Python

- A Python for loop traverses a sequence defined by an iterator
- If the iterator terminates, so does the for loop
- The iterator could also go on forever
- The iterator could give a sequence of numbers, or it could traverse a lost, or some other container
- This is convenient but it hides what's actually going on

14.7 COMPILERS

- We have been writing algorithms in high-level language notation and then translating it manually to assembly language
- A compiler is a software application that performs this translation automatically
- A programming language is a precisely defined high-level language
- The compiler makes programming easier by allowing you to think about your algorithm more abstractly, without worrying about all the details of the machine

14.7.1 Source and object

- The original high-level language program is called the source code – it's what the programmer writes
- The final machine language program which the compiler produces is called the object code – it's what the machine executes

14.7.2 Compilation

- A compiler translates statements in a high-level language into assembly language
- In developing an assembly language program, it's best to begin by writing high-level pseudocode (this becomes a comment) and then translate it
- This approach helps keep the program readable, and reduces the chances of getting confused
- Each kind of high-level statement corresponds to a standard pattern in assembly language

14.7.3 How a compiler works

- Your high-level source program is just a character string – the computer cannot execute this directly
- The compiler reads the source program, checks its syntax, and analyses its structure
- Then it checks the types of all the variables and procedures
- Most advanced compilers translate the program to an intermediate "goto form", just as we are doing
- The program is finally translated to assembly language: "code generation"
- The assembler translates the assembly language to machine language

14.7.4 Major tasks in compilation

- **Parsing** – check the source program for correct syntax and work out the program structure (similar to "diagramming sentences" in English grammar)
- **Type checking** – work out the data type of each variable (integer, character, etc) – then:
  - Make sure the variables are used consistently
  - Generate the right instructions for that data type
- **Optimisation** – analyse the program to find opportunities to rearrange the object code to make it faster
- **Code generation** – produce the actual machine instructions

14.8 PARSING

- The syntax of a language is its set of grammar rules
- If the source program contains a syntax error, the compiler will not understand what you mean
- If there is a syntax error, you do no want the compiler to guess what the meaning is – that leads to unreliable software

14.9 TYPES

- Type checking is one of the most important tasks the compiler performs
- There are many data types supported by a computer
  - Binary integer
  - Two's complement integer
  - Floating point
  - Character
  - Instruction
- The computer hardware works on words, and the machine does not know what data type a word is
- It's essential to use the right instruction, according to the data type

14.9.1 Integer and floating point

- An integer is a whole number
- A floating point number may have a fraction and exponent
- We have seen how an integer is represented: two's complement
- Floating point representation is different from two's complement
- Most computers have separate instructions for arithmetic on integers and floating point numbers
  - **add R1,R2,R3** – integer addition
  - **addf R1,R2,R3** – floating point addition
- The machine doesn't know what the bits in the registers mean – you must use the right instruction according to the data type

14.10 TYPECHECKING

- The compiler checks that each variable in the source program is used correctly
  - If you add a number to a string, it's a type error
- Then it generates the correct instructions for the type
  - For integer variables it uses **add**
  - For floating pointer variables it uses **addf**
  - These are different instructions
- This eliminates one of the most common kinds of error in software

15 FURTHER NOTES

- If you wish to read up on more advanced notes towards the end of the semester, I have included all the lecture notes in the **Resources** folder
- Here are the slides that were not covered:
  - CompSysLec20CMOS_VLSI.pdf
  - CompSysLec21SpectreMeltdown.pdf
  - CompSysLec22Retrospective.pdf

16 LINKS AND REFERENCES

- Lecture recordings: https://echo360.org.uk/section/87d4126e-6c4a-422b-8879-0720416fe818/home
- Full credits go to John O'Donnell – I just copied his notes, change some small things, and put it in a nicer format
- All lab content has been put into the folder named **Labs**
- All code examples have been put into the folder **Assembly Examples**
- Visit **https://johncat.co.uk** for a list of useful resources and notes