ECE/ME/EMA/CS 759
High Performance Computing for Engineering Applications

Debugging w/ **gdb**

Launching a job on Euler

Basic elements of computer architecture and micro-architecure

September 4, 2015

Quote of the Day

“Be who you are and say what you feel, because those who mind don't matter and those who matter don't mind.”

Dr. Seuss

Debugging on Euler

[with gdb]

Slides on gdb include material from Purdue University

gdb: Intro

- gdb: a utility that helps you debug your program
- Learning `gdb` is a good investment of your time
  - Yields significant boost of productivity
- A debugger will make a good programmer a better programmer
- In ME759, you should go beyond sprinkling “`printf`” here and there to try to debug your code
  - Avoid: Compile-link, compile-link, compile-link, compile-link, compile-link, compile-link, compile-link, compile-link,…

Compiling a Program for gdb

- You need to compile with the “-g” option to be able to debug a program with gdb.

- The “-g” option adds debugging information to your program

```
gcc -g -o hello hello.c
```

Running a Program with gdb

- To run a program called **`progName`** with **`gdb`** type

```
>> gdb progName
```

- Then set a breakpoint in the main function

```
(gdb) break main
```

- A breakpoint is a marker in your program that will make the program stop and return control back to gdb

- Now run your program

```
(gdb) run
```

- If your program has arguments, you can pass them after run.

Stepping Through your Program

- Your program will start running and when it reaches “main()” it will stop:
  `(gdb)`

- You can use the following commands to run your program step by step:
  `(gdb) step`

  It will run the next line of code and stop. If it is a function call, it will enter into it

  `(gdb) next`

  It will run the next line of code and stop. If it is a function call, it will not enter the function and it will go through it.

Printing the Value of a Variable

- The command
  `(gdb) print varName`
  … prints the value of a variable

E.g.

```
(gdb) print i
$1 = 5
(gdb) print s1
$1 = 0x10740 "Hello"
(gdb) print stack[2]
$1 = 56
(gdb) print stack
$2 = {0, 0, 56, 0, 0, 0, 0, 0, 0, 0}
(gdb)
```

Setting Breakpoints

- A breakpoint is a location in a program where the execution stops in **gdb** and control is passed back to you

- You can set breakpoints in a program in several ways:

```
(gdb) break functionName
```

Set a breakpoint in a function. E.g.

```
(gdb) break main
```

```
(gdb) break lineNumber
```

Set a break point at a line in the current file. E.g.

```
(gdb) break 66
```

It will set a break point in line 66 of the current file.

```
(gdb) break fileName:lineNumber
```

It will set a break point at a line in a specific file. E.g.

```
(gdb) break hello.c:78
```

```
(gdb) break fileName:functionName
```

It will set a break point in a function in a specific file. E.g.

```
(gdb) break subdivision.c:paritalSum
```

Watching a Variable

- Many times you want to keep an eye on a variable that for some reason assumes a value that is not in line with expectations

- To that end, you can "watch" a variable and have the code break as soon as the variable is read or changed

- You can watch a variable in several ways:

  `(gdb) watch varName`

  Program breaks whenever **`varName`** gets written by the program

  `(gdb) rwatch varName`

  Program breaks whenever **`varName`** gets read by the program

  `(gdb) awatch varName`

  Program breaks whenever **`varName`** gets read/written by the program

- Get a list of all watchpoints, breakpoints, and catchpoints in your program:
  `(gdb) info watchpoints`

Example:

[watching a variable]

```
#include <iostream>

int main(){
  int arr[2]={266,5};

  int * p;
  short s;

  p = (int*) malloc(sizeof(int)*3);

  p[2] = arr[1] * 3;

  s = (short)( *(p+2) );

  free( p );

  p=NULL;

  p[0] = 5;
  return 0;
}
```

Example: Watching Variable “s”

- Below is a copy-and-paste from gdb, for our short program

```
(gdb) awatch s
Hardware access (read/write) watchpoint 2: s
(gdb) continue
Continuing.
Hardware access (read/write) watchpoint 2: s

Old value = 0
New value = 15
main () at pointerArithm.cpp:15
15              free( p );
```

Regaining the Control

- When you type

  ```
  (gdb) run
  ```

  the program will start running and it will stop at a breakpoint

- If the program is running without stopping, you can regain control again typing ctrl-c

- When you type

  ```
  (gdb) continue
  ```

  the program will run until it hits the next breakpoint, or exits

Where Are You?

- The command

  `(gdb)where`

  Will print the current function being executed and the chain of functions that are calling that fuction.

  This is also called the backtrace.

  Example:

```
(gdb) where
#0  main () at test_mystring.c:22
(gdb)
```

Seeing Code Around You...

- The command **`list`** shows you code around the location where the execution is "break-ed"

```
(gdb)list
```

It will print, by default, 10 lines of code.

There are several flavors:

```
(gdb)list lineNumber
```

…prints code around a certain line number

```
(gdb)list functionName
```

…prints lines of code around the beginning of a function

```
(gdb)set listsize someNumber
```

…controls the number of lines showed with **`list`** command

Exiting gdb

- The command “quit” exits **gdb**.

```
(gdb) quit
The program is running.  Exit anyway?
  (y or n) y
```

Debugging a Crashed Program

- Also called “postmortem debugging”
- When a program segfaults, it writes a **core file**.

```
bash-4.1$ ./hello
Segmentation Fault (core dumped)
bash-4.1$
```

- The core is a file that contains a snapshot of the state of the program at the time of the crash
  - Information includes what function the program was running upon crash

Example:

[Code crashing]

```
#include <iostream>

int main(){
  int arr[2]={266,5};

  int * p;
  short s;

  p = (int*) malloc(sizeof(int)*3);

  p[2] = arr[1] * 3;

  s = (short)( *(p+2) );

  free( p );

  p=NULL;

  p[0] = 5;
  return 0;
}
```

This is why it’s crashing…

Running gdb on a Segmentation fault

- Here’s what gdb says when running the code…

```
[negrut@euler CodeBits]$ gdb badPointerArithm.out
GNU gdb (GDB) Red Hat Enterprise Linux (7.2-50.el6)
Copyright (C) 2010 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/negrut/ME964/Spring2012/CodeBits/badPointerArithm.out...done.
(gdb) run
Starting program: /home/negrut/ME964/Spring2012/CodeBits/badPointerArithm.out
warning: the debug information found in "/usr/lib/debug//lib64/libc-2.12.so.debug" does not match
"/lib64/libc.so.6" (CRC mismatch).

warning: the debug information found in "/usr/lib/debug/lib64/libc-2.12.so.debug" does not match
"/lib64/libc.so.6" (CRC mismatch).


Program received signal SIGSEGV, Segmentation fault.
0x0000000000400641 in main () at pointerArithm.cpp:19
19                      p[0] = 5;
(gdb)
```

Debugging – Departing Thoughts

- Debug like a pro (graduate from the use of **`printf`**…)
- **`dbg`** can save you time
- If a GUI is helpful, use “**`ddd`**” on Euler – under the hood it uses **`gdb`**
- Under Windows, VisualStudio has an excellent debugger

Moving Beyond gdb:
Improving your productivity as a programmer

- Turn on flags that make the compiler be picky and whiny
- Use clint – good semantic checker (at compile time)
- Use valgraind – dynamically monitors the execution of your program (at run time)
- Keep your code simple
- Comment your code
- Use revision control

[Colin Vanden Heuvel]→

JOB SUBMISSION W/ SLURM

SLURM

- SLURM: “Simple Linux Utility for Resource Management”
- Euler uses SLURM to manage; i.e., queue for execution, you jobs
- SLURM documentation: http://slurm.schedmd.com/documentation.html

Job Submission

- Two modes: batch and interactive
- Option 1: Batch Mode
  - Compute task written as shell script, with SLURM-specific comments
- Option 2: Interactive Mode
  - You get access to an interactive shell on a compute node

Job Submission Option 1: Batch Mode

bandwidthTest.sh

(you'll have to create this file)

```
#!/bin/bash                                   → Shell script
#SBATCH -p slurm_me759                        → Use Class Queue
#SBATCH --job-name=bandwidthTest              → Name of job
#SBATCH –N 1 -n 1 --gres=gpu:1                → Resource selection
#SBATCH -o bandwidthTest.o%j                  → Set output file
cd $SLURM_SUBMIT_DIR                          → Set Work Directory
./bandwidthTest                               → Run!
```

Submit with:

```
$ sbatch bandwidthTest.sh
```

Output placed in bandwidthTest.o[0-9]*

Job Submission Option 2: Interactive

```
me759@euler $ srun -p slurm_me759 -u bash -i

me759@node $ ./bandwidthTest
```

- Note that the examples on this and the previous slide use the slurm_me759 queue. It is a special queue reserved for this class.
- Jobs not submitted to the class queue lack context and end up cancelled

Euler: Resource Selection

- Request can follow a flag such as `-N` or `–n`, and/or it can follow a `--gres=…` (Generic RESource) flag.

- Examples
  - One node with one GPU

    ```
    -N 1 --gres=gpu:1
    ```

  - Two nodes with one GPU/node

    ```
    -N 2 --gres=gpu:1
    ```

  - Two nodes with three processors per node

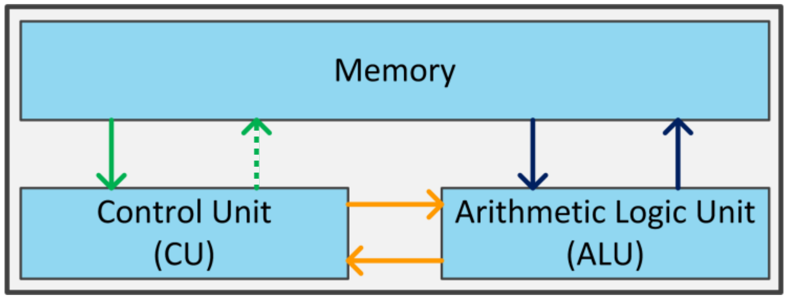
    ```
    -N 2 -n 3
    ```

- Note: must request GPUs for GPU jobs

Basic Elements of Computer Architecture and Microarchitecture

Today’s Computer

- Follows computational paradigm formalized by von Neumann in late 1940s
- The von Neumann model:
  - There is no distinction between data and $\boxed{\text{instructions}}$
  - Data and instructions are stored in memory as a string of 0 and 1 bits
    - Instructions are fetched & decoded & executed
    - Data is used to produce results according to rules specified by the instructions



Line of Code vs. Machine Instruction

```
#include <iostream>

int main(){
  int arr[2]={266,5};

  int * p;
  short s;

  p = (int*) malloc(sizeof(int)*3);

  p[2] = arr[1] * 3;

  s = (short)( *(p+2) );

  free( p );

  return 0;
}
```

For the record:
This is a line of C code,
and not a machine instruction

From Code to Machine Instructions

- There is a difference between a line a code and a machine instruction
- Example:
  - Line of C code:

```
a[4] = delta + a[3]; //line of C code
```

  - MIPS *assembly code* generated by the compiler
    - Three instructions are generated for the above line of C code:

```
lw $t0, 12($s2)  # reg $t0 gets value stored 12 bytes from address in $s2
add$t0, $s4, $t0 # reg $t0 gets the sum of values stored in $s4 and $t0
sw $t0, 16($s2)  # store at 16 bytes from address in $s2 what’s in $t0
```

  - Three corresponding MIPS *machine instructions* produced by the compiler:

```
10001110010010000000000000001100
00000010100010000100000000100000
10101110010010000000000000010000
```

[Cntd.]

From Code to Instructions

- C code – what you use to implement an algorithm
- Assembly code – intermediate step for compiler, something that humans can read
- Machine code/Instructions – what the assembly code gets translated into by the compiler and the CU understands
  - Machine code: what you see in an editor like `notepad` or `vim` or `emacs` if you open up an executable file
  - There is a one-to-one correspondence between a line of assembly code and an instruction (most of the time)

[Cntd.]

From Code to Instructions

- Observations:
  - The compiler typically goes from C code directly to machine instructions
  - People used to write assembly code
  - Today coding in assembly done only for critical parts of a program by people who want to highly optimize the execution and don’t trust the compiler

Instruction Set Architecture (ISA)

- The same line a C code can lead to a different set of instructions on two different computers
- This is so because two CPUs might draw on two different Instruction Set Architectures (ISA)
- ISA: defines a “vocabulary” used to express at a very low level the actions of a processor
  - ISA: the set of words that can be used to tell the CU what to do
- Example:
  - Microsoft’s Surface Tablet
    - RT version: uses a Tegra chip, which implements an ARM Instruction Set
    - Pro version: uses an Intel Atom chip, which implements x86 Instruction Set

Example: the same C code leads to different assembly code (and different set of machine instructions, not shown here)

C code

```
int main(){
  const double fctr = 3.14/180.0;
  double a = 60.0;
  double b = 120.0;
  double c;
  c = fctr*(a + b);
  return 0;
}
```

x86 ISA

```
call    ___main
        fldl   LC0
        fstpl  -40(%ebp)
        fldl   LC1
        fstpl  -32(%ebp)
        fldl   LC2
        fstpl  -24(%ebp)
        fldl   -32(%ebp)
        faddl  -24(%ebp)
        fldl   LC0
        fmulp  %st, %st(1)
        fstpl  -16(%ebp)
        movl   $0, %eax
        addl   $36, %esp
        popl   %ecx
        popl   %ebp
        leal   -4(%ecx), %esp
        ret
LC0:
        .long  387883269
        .long  1066524452
        .align 8
LC1:
        .long  0
        .long  1078853632
        .align 8
LC2:
        .long  0
        .long  1079902208
```

MIPS ISA

```
main:
        .frame $fp,48,$31    # vars= 32, regs= 1/0, args= 0, gp= 8
        .mask  0x40000000,-4
        .fmask 0x00000000,0
        .set   noreorder
        .set   nomacro
        addiu  $sp,$sp,-48
        sw     $fp,44($sp)
        move   $fp,$sp
        lui    $2,%hi($LC0)
        lwc1
          …
        mul.d  $f0,$f2,$f0
        swc1   $f0,32($fp)
        swc1   $f1,36($fp)
        move   $2,$0
        move   $sp,$fp
        lw     $fp,44($sp)
        addiu  $sp,$sp,48
        j      $31
          …
$LC0:
        .word  3649767765
        .word  1066523892
        .align 3
$LC1:
        .word  0
        .word  1078853632
        .align 3
$LC2:
        .word  0
        .word  1079902208
        .ident "GCC: (Gentoo 4.6.3 p1.6, pie-0.5.2) 4.6.3"
```

RISC vs. CISC

- RISC Architecture – “Reduced Instruction Set Computing” Architecture
  - Each instruction has fixed length, be it 32 or 64 bits (no mixing of the two)
    - Move to 64 bits took place relatively recently
  - Promoted by: ARM Holding, company that started as ARM (Advanced RISC Machines)
    - Used in: embedded systems, smart phones – Intel, NVIDIA, Samsung, Qualcomm, Texas Instruments
    - Somewhere between 8 and 10 billion chips based on ARM manufactured annually

- CISC Architecture – “Complex Instruction Set Computing” Architecture
  - Instructions have various lengths
    - Examples: 32 bit instruction followed by 256 bit instruction followed later on by 128 bit instruction, etc.
  - Intel’s X86 is the most common example
  - Promoted by Intel and subsequently embraced and augmented by AMD
    - Used in: laptops, desktops, workstations, supercomputers

RISC vs. CISC

- RISC is simpler to comprehend, provision for, and work with
- Decoding CISC instructions is not trivial and eats up power
- A CISC instruction is usually broken down into several micro-operations (uops)
- CISC Architectures invite spaghetti type evolution of the ISA and require complex microarchitecture
  - On the upside, they provide the freedom to do as you wish

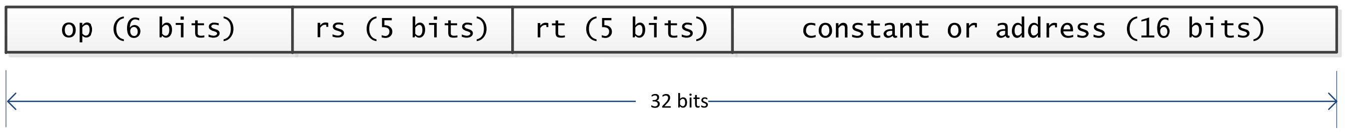
The FDX Cycle

- FDX stands for Fetch-Decode-Execute
- FDX is what keeps the CU busy
  - The CU does a FDX for instruction after instruction until program completes

- Fetch: an instruction is fetched from memory
  - Recall that it will look like this (on 32 bits, MIPS, `lw $t0, 12($s2)`):

    10001110010010000000000000001100

- Decode: this strings of 1s and 0s are decoded by the CU
  - Example: here's an "I" (eye) type instruction, made up of four fields

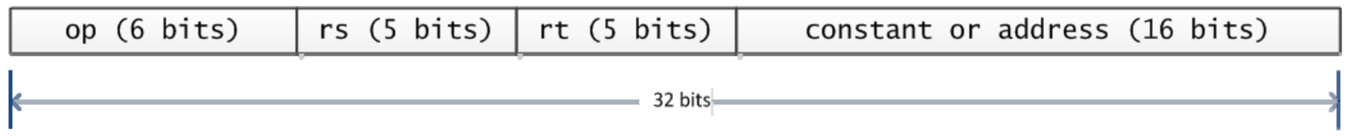


[Cntd.]

Decoding: Instructions Types

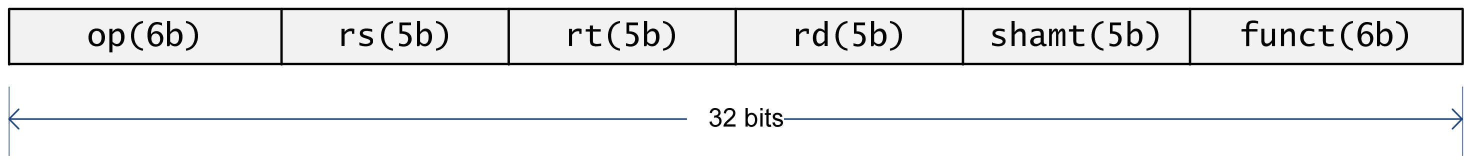
- Three types of instructions in MIPS ISA
  - Type I
  - Type R
  - Type J

Type I (MIPS ISA)
[I comes from “Immediate”]



- The first six bits encode the basic operation; i.e., the opcode, that needs to be completed
    - Examples: adding (000000), subtracting (000001), etc.
- The next group of five bits indicates in which register the first operand is stored
- The subsequent group of five bits indicates the destination register
- The last 16 bits: the “immediate” value, usually used as the offset value in various instructions
    - “Immediate” means that there is no need to read other registers or jump through other hoops. What you need is right there and you immediately can use it

Type R (MIPS ISA)



- Type R has the same first three fields op, `rs`, `rt` like I-type
- Packs three additional fields:
  - Five bit `rd` field (register destination)
  - Five bit `shamt` field (shift amount)
  - Six bit `funct` field, which is a function code that further qualifies the `opcode`

Instruction Set Architecture vs. Chip Microarchitecture

- ISA – can be regarded as defining a vocabulary
  - Specifies what a processor should be able to do
    - Load, store, jump on less than, etc.

- Microarchitecture – how the silicon is organized to implement the vocabulary promised by ISA

- Example:
  - Intel and AMD both use the x86 ISA yet they have different microarchitectures

The CPU’s Control Unit (CU)

- Think of a CPU as a big kitchen
  - An order comes in (this is an instruction)
  - Some ingredients are needed: meat, pasta, broth, etc. (this is the data)
  - Some ready to eat product goes out the kitchen: a soup (this is the result)
    - The product can also be broth that is stored for later use

- Bringing in the meat, bringing in the pasta, placing them in the proximity (the registers), mixing them in a certain way (op), happens in a coordinated fashion (based on a kitchen clock) that is managed by the CU

- The CU manages/coordinates/controls based on the food order (the instruction)

FDX Cycle – The Execution Part: It All Boils Down to Transistors…

- How does this magic happen?
  - Transistors can be organized to produce complex logical units that have the ability to execute instructions
  - More transistors increase opportunities for building/implementing in silicon functional units that can operate at the same time towards a shared goal

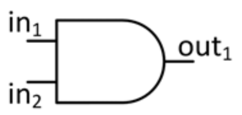
Transistors and then some more transistors

- First, we'll talk about how transistors are used to implement operations (perform tasks)

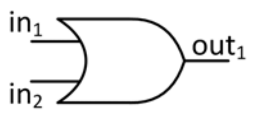
- Later we'll talk about how transistors are used to store data and instructions

Transistors at Work: AND, OR, NOT

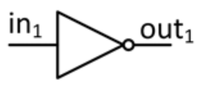
- The NOT logical op. is implemented using one transistor
- AND and OR logical ops require two transistors



AND



OR



NOT

- Truth tables for AND, OR, and NOT

AND	in_2=0	in_2=1
in_1=0	0	0
in_1=1	0	1

OR	in_2=0	in_2=1
in_1=0	0	1
in_1=1	1	1

NOT	
in_1=0	1
in_1=1	0

Example

- Design a digital logic block that receives three inputs via three bus wires and produces one signal that is 0 (low voltage) as soon as one of the three input signals is low voltage.
  - In other words, it should return 1 if and only if all three inputs are 1

Truth Table $\longleftrightarrow$ Logic Equation: $out = \overline{\overline{in_1} + \overline{in_2}} \cdot in_3$

in_1	in_2	in_3	Out
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

- Overbar $\bar{\ }$: negation
- Plus $+$: logical OR
- Product $\cdot$: logical AND

Example
[Cntd.]

- Easy to figure out the transistor setup once Logic Equation is available

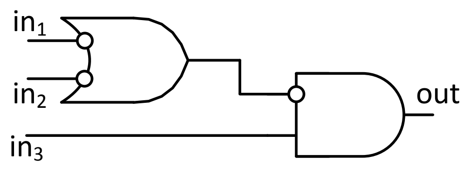
Truth Table

in_1	in_2	in_3	Out
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

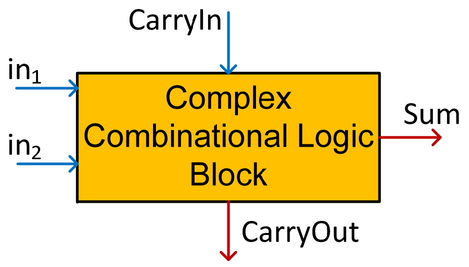
Logic Equation:

$$out = \overline{\overline{in_1} + \overline{in_2}} \cdot in_3$$

- Solution: digital logic block is a combination of AND, OR, and NOT gates
  - The NOT is represented as a circle O applied to signals moving down the bus



Example: One Bit Adder



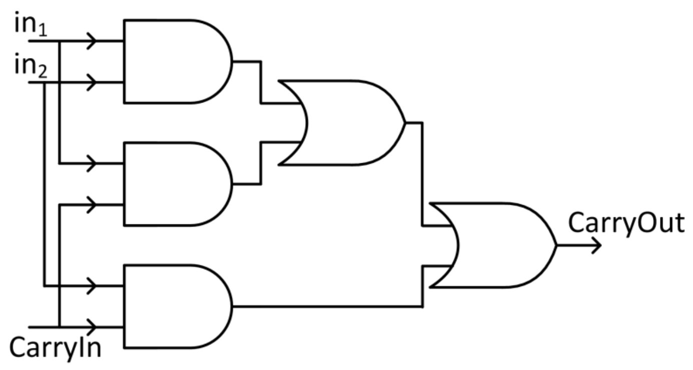
- Implement a digital circuit that produces the Carry-out digit in a one bit summation operation

Truth Table

Inputs			Outputs		Comments
in_1	in_2	CarryIn	Sum	Carry Out	Sum is in base 2
0	0	0	0	0	
0	0	1	1	0	0+0 is 0; the CarryIn kicks in, makes the sum 1
0	1	0	1	0	
0	1	1	0	1	0+1 is 1, but CarryIn is 1; sum ends up being 0, CarryOut is 1.
1	0	0	1	0	
1	0	1	0	1	1+0 is 1, but CarryIn is 1; sum ends up being 0, CarryOut is 1.
1	1	0	0	1	1+1 is 0, carry 1.
1	1	1	1	1	1+1 is 0 and you CarryOut 1. Yet the CarryIn is 1, so the 0 in the sum becomes 1.

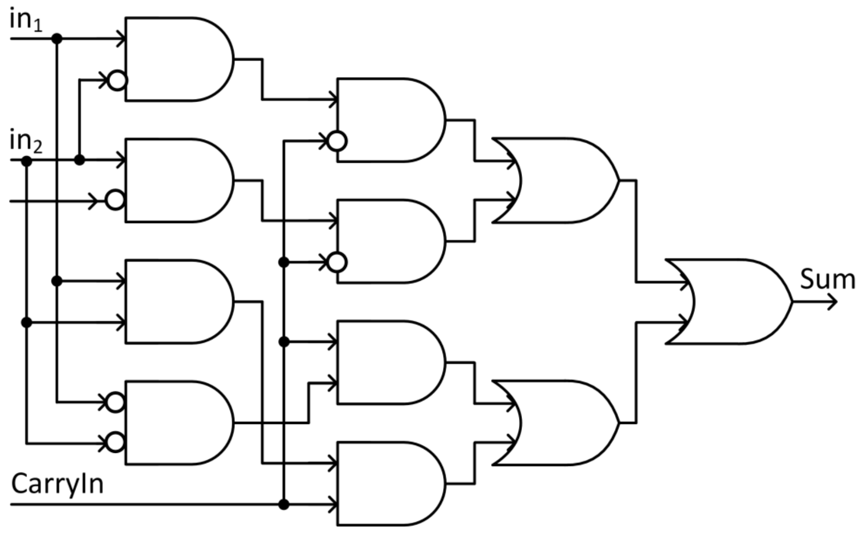
Logic Equation:

$$\text{CarryOut} = (in_1 \cdot \text{CarryIn}) + (in_2 \cdot \text{CarryIn}) + (in_1 \cdot in_2)$$

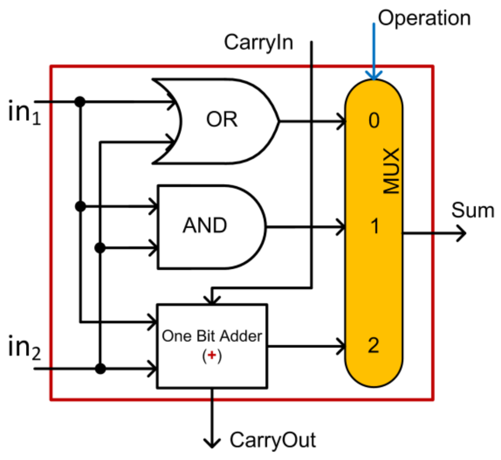


Integrated Circuits-A One Bit Combo: OR, AND, 1 Bit Adder

- 1 Bit Adder, the Sum part

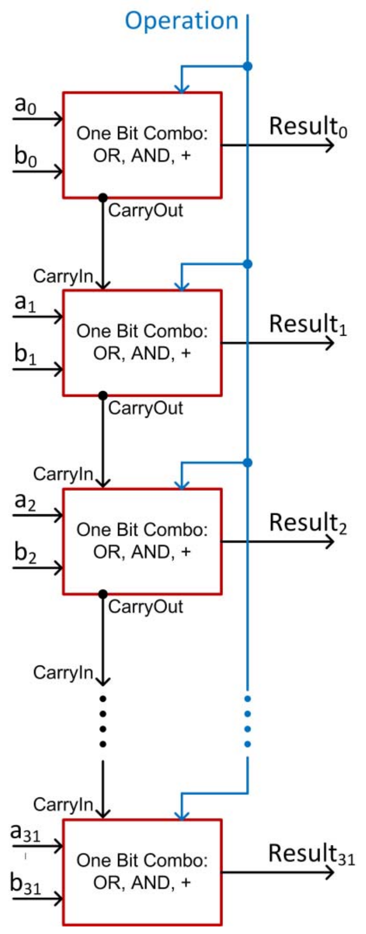


- Combo: OR, AND, 1 Bit Sum
  - Controlled by the input "Operation"

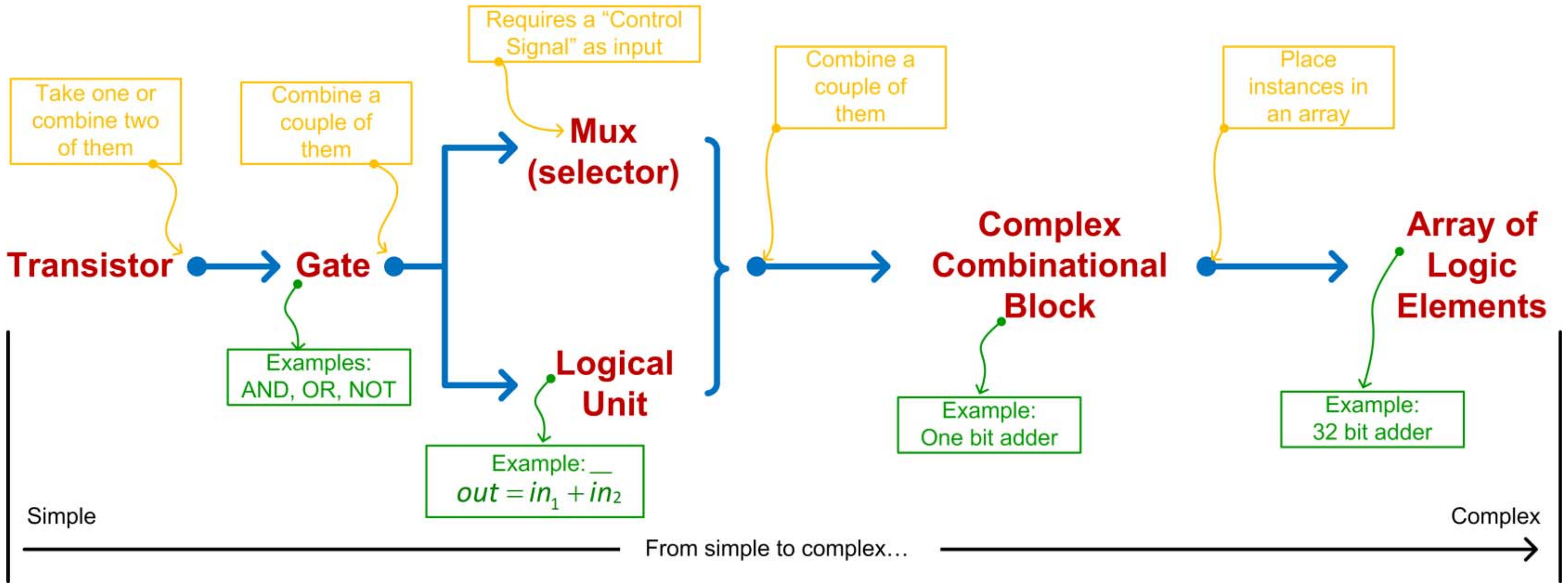


Integrated Circuits: Ripple Design of 32 Bit Combo

- Combine 32 of the 1 bit combos in an array of logic elements
  - Get one 32 bit unit that can do OR, AND, +



Integrated Circuits: From Transistors to CPU



[FDEX Cycle: Execution Closing Remarks]
It All Boils Down to Transistors…

- Every 18 months, the number of transistors per unit area doubles (Moore’s Law)
  - Current technology (2014-2015): feature length is 14 nm (Intel)
  - Next wave (2016-2017): 10 nm (Intel)
  - Looking ahead (Intel)
    - 7 nm – 2017-2018
    - 5 nm – 2020-2021

- No clear path forward after 2021
  - Maybe Carbon Nanotubes?

Number of Transistors, on GPUs

- NVIDIA Architectures
  - Fermi circ. 2010:
    - 40 nm technology
    - Up to 3 billion transistors → about 500 scalar processors, 0.5 d.p.Tflops
  - Kepler circ. 2012:
    - 28 nm technology
    - Chips w/ 7 billion transistors → about 2800 scalar processors, 1.5 d.p. Tflops
  - Maxwell 2015-2016
    - 28 nm technology
    - Chips w/ 8 billion transistors → 3072 scalar processors, 6.1 s.p. Tflops