CS151: Computer Architecture

Professor Reinman

Thilan Tran

Winter 2021

Contents

CS151: Computer Architecture

- eight great ideas of computer architecture design:
    1. design for Moore's law
    2. use abstraction to simplify design
    3. make the common case fast
    4. performance via parallelism
        - eg. memory operations
    5. performance via pipelining
        - ie. using an assembly line of specialized tasks
    6. performance via prediction
        - eg. predicting control flow branches
    7. hierarchy of memories
    8. dependability via redundancy

Overview and Performance

- general hardware components of a computer:
  - processor with:
    - datapaths that perform operations on data
    - control operations that sequence the datapath
    - cache memory
  - main system memory
  - other devices eg. I/O with input devices, external storage devices, network adaptors, etc.
- stack of layers between a user app and the hardware of a computer:
  1. user apps
  2. system apps
  3. Java API framework
  4. native C/C++ libraries
  5. hardware abstraction layer
  6. Linux kernel
  7. the actual silicon hardware, AKA **system on chip (SoC)**
  - need to balance specificity of hardware with portability of applications
- in a SoC, instead of just a single component, the chip contains multiple components packaged together that interact with each other:
  - eg. CPU, GPU, memory, I/O, media, etc.
  - *pros*:
    - better integration and lower latency vs. separated components that do I/O through pins
      - on-chip components are embedded and stacked together more tightly
  - *cons*:
    - may lead to new problems with managing heat
  - manufaturing silicon chips:
    - slice a silicon ingot into wafers
    - process the wafers with patterns through **lithography**
    - dice the wafers into dies
    - package dies
    - at each step, failures can occur
    - **yield** is the proportion of working dies per wafer
- layers of program code:
  1. high-level language
     - very abstracted, portable (across different systems!)
  2. assembly language:
     - textual representation of instructions

    - specific to a system, eg. MIPS vs. x86, machine-specific instructions
  3. hardware representation:
    - encoded instructions and data in binary
    - what machine actual reads
- the **instruction set architecture (ISA)** exposes a set of primitives to the layers *above* it in order to work with the silicon hardware *below* the ISA:
  - ie. the interface or template for interactions between the CPU and the drivers, OS, apps above it
  - defines instructions like `add` or `movsql` , calling protocols, registers, etc.
  - in this class: how do we implement the silicon *hardware* to actually implement an ISA, using the building blocks of a microprocessor?

Performance

- evaluating performance:
  - performance is one aspect of evaluating a microprocessor:
    - other areas include power management, reliability, size, etc.
    - can be evaluated in terms of latency or response time (delay), vs. throughput (bandwidth)
  - big-picture constraints on performance are power, instruction-level parallelism, and memory latency
- consider the following equations for response time:

$$ET = IC \times CPI \times T_C$$

$$ET = \frac{IC \times CPI}{T_R}$$

  - performance is inversely related to response time
  - ET is the execution time
  - IC are the instruction counts
    - *dynamic* runtime count, rather than *static* count in a program
  - CPI are the cycles per instruction:
    - clocks are used due to the synchronous nature of memory in circuits
    - but since different instruction *types* will have different cycle counts, depends on both:
      - mix of instructions used
      - hardware implementation of these instructions
    - thus to find the CPI as a *weighted average*, need to compute the

percentages and cycles of mix of instructions that *make up* the IC:

$$CPI = \sum_{i=1}^{n} CPI_i \times f_i$$

        - where f_i is the relative frequency of the instruction type CPI_i
    - note that the CPI is a count dependent on the *particular* program and the hardware it runs on!
  - T_C is the cycle time ie. clock period:
    - inverse to clock rate T_R
    - at a shorter clock period, CPI may increase since there are fewer cycles for a more complex instruction to complete
      - tradeoff between CPI and T_C
    - eg. T_R is commonly between 3.5-4 GHz
      - recently topping out due to power efficiency limitations
  - note that while total elapsed time records processing, I/O, and idle times, etc. CPU time records the time spent processing a single job
    - discounts the time of other jobs' shares, I/O, etc.
- affect on performance:
  1. algorithm affects IC, possibly CPI
     - could reduce number of instructions, or type of instructions used, eg. multiplication vs. addition
  2. programming language affects IC, CPI
  3. compiler affects IC, CPI
  4. ISA affects IC, CPI, T_C
     - ISA may limit expressiveness of higher-level languages or of underlying hardware implementations
  5. Hardware affects CPI, T_C
     - eg. change clock period by reducing wire delay, change CPI by improving implementation of an instruction
- performance pitfalls:
  - **Amdahl's law** explains how improving one aspect of a computer affects the overall performance:

$$T_{improved} = T_{unaffected} + \frac{T_{affected}}{improvement\ factor}$$

    - ie. strive to make the common case fast
  - using **millions of instructions per second (MIPS)** as a performance metric can be extremely misleading:
    - ie. ignores instruction count when considering execution time
    - doesn't account for different ISAs and different instruction complexities

Power

- consider the following equation for power:

$$power = capacitive\ load \times voltage^2 \times frequency$$

  - hitting a *"wall"* of power, ie. voltage is not scaling down enough recently
    - problems with removing heat caused by so much power
  - there is a need to find other ways to decrease power usage
    - eg. dynamic voltage scaling or dynamic frequency voltage to decrease power proportionally depending on the usage of the chip
- while performance has been steadily improving, power demands for chips have similar increased dramatically:
  - thus, rather than pushing for more performant, power-hungry, monolothic cores, instead design towards multi-core designs in a **many-core revolution**
  - in a multi-core design, many cores can work together or multitask

Multiprocessors

- put more than one processor per chip
- *pros*:
  - can dramatically improve performance, past the power wall limitation
- *cons*:
  - requires explicitly parallel programming by programmers
    - most programs are not *"embarassingly"* parallel
  - vs. instruction level parallelism:
    - where the *hardware* executes multiple instructions at once
    - hidden from programmers

Computer Instructions

ISAs and ISA Decisions

- the **instruction set architecture (ISA)** is the repertoir of instructions of a computer:
  - different computers have different ISAs
  - early computers had simple ISAs
  - some modern computers continue to use simple ISAs
- key ISA decisions:
  - number, length, type of operations
  - number, location, type of operands
    - also how to specify operands
  - instruction format, eg. limits on size
- main classes of ISAs are CISC and RISC
- **complex instruction set computers (CISC)** provide a large number of instructions
  - *pros*:
    - many specialized complex instructions with dedicated hardware components
    - can also optimize for specialized operations
    - good for tight memory restrictions where program size was optimized
  - eg. VAX, x86
  - eg. for `a*b + c*d` , may be implemented with a single multiply-add-multiply (MAM) instruction
    - need more overall opcodes to account for many different instructions, takes 5 operands
- **reduced instruction set computers (RISC)** have relatively fewer instructions
  - *pros*:
    - enables pipelining and parallelism because of simplicity of hardware:
      - smaller, simpler pieces
      - no need to individually parallelize the hardware for many complex instructions
    - easier to optimize and *reuse* hardware for efficiency
    - easier to validate hardware
  - eg. MIPS, PowerPC, ARM
    - this class uses MIPS

  - eg. for `a*b + c*d`, may be implemented with two multiplies and an add instruction
    - reusing basic instructions, but uses 3 opcodes total for the 3 instructions, takes $3 \times 3 = 9$ total operands
- blurring distinction between CISC and RISC:
  - on the CISC side, x86 reduces instructions to **micro-ops** that look like RISC instructions
    - downside is that it is more difficult for compiler to optimize these complex instructions that will break down into micro-ops
  - on the RISC side, ARM is also performing a process called **micro-op fusion** that takes smaller instructions and fuses them into more complex operations, until they can be broken apart again:
    - with some more complex instructions, may take up less space even in the *hardware* to implement
    - in part also motivated by the limited instruction window used in the out-of-order execution optimization
- performance tradeoff of RISC vs. CISC:
  - RISC typically has a higher IC
    - more expressive instructions
  - RISC can have a lower cycle time or lower CPI:
    - CT can decrease since latency for more complex operations is gone
    - average CPI can drop since expensive complex operations are broken down
- considerations with ISA instruction lengths:
  - variable length (VL) instructions have different sizes, eg. `ret` instruction may only need 8 bits, while a MAM instruction may need 64 bits (with some alignment rules)
    - contrasted to fixed length (FL) instructions
  - usually, CISC uses VL and RISC uses FL
  - performance tradeoff:
    - less memory fragmentation with VL
    - more flexibility and less bit-width related limits with VL
      - thus VL can have a lower IC, but may also have a higher CT and average CPI, due to increased decoding time
    - easier instruction decoding with FL
- ISA instruction operands can generally refer to different values:
  1. an immediate, stored directly in the instruction code
  2. a register in a register file:
     - latency of 1-3 cycles
     - a 5 bit register number specifies one of the 32 MIPS registers (one level of indirection)
     - less bits required than the full register address
  3. memory:

    - latency of 100s of cycles
    - 2^{32} addressable locations in MIPS
    - one way to form an effective address to memory is to refer to a register that then contains a memory address (two levels of indirection)
    - could also supply an offset off of an address in register to form an effective address
  - note that branch instructions treat addresses as word offsets, while load and store word treat them as byte offsets (since neither instruction type can take a 32-bit address)
- the types of operands a MIPS instruction can take (ie. the **addressing mode**) is based off its type:
  - **R-type** instructions take 3 registers eg. `add`
    - `0` is the single R-type opcode, but many possible instructions are specified through the `func` field eg. `add, sub, or`
  - **I-type** instructions take 2 registers and an immediate eg. `addi, lw, sw`
    - multiple opcodes
  - **J-type** instructions just take an immediate eg. `j, jal` :
    - multiple opcodes
- on the other hand, in the addressing mode of x86, every instruction can refer into memory:
  - leads to increased complexity for all instructions
    - more difficult to isolate memory loads and stores in instructions and perform optimizations like prefetching
  - thus for x86-like addressing, IC can go down, but the CPI and CT could go up
  - in addition, in RISC, *more* pressure on the register file since the only way to load memory is to use `lw` to load it into a register
    - typically have larger register files
- instruction format bit field tradeoffs:
  - the 5-bit field to refer to registers is dependent on the number of registers
    - eg. if we increased the register file size, may need 6 bits in the instruction format for all registers
  - with more registers, there is less spilling and fewer `lw` instructions, which have the most latency:
    - have to go to memory (after effective address computation from immediate and register) and afterwards back to register file
    - thus lower IC and CPI
    - but must take the extra bits for the register fields from somewhere else
  - could reduce the number of bits for immediates
    - may increase IC, since instructions may need more instructions to

build the same immediates
  - could reduce the number of bits for an opcode:
    * may lose the number of functions we can specify
    * in addition, every format's opcode size must change, since they must be equal
    * may increase IC
  - could reduce the number of bits for a shift
    * less expressiveness, may increase IC
  - but if we don't use a certain feature, and perform many spills, may be beneficial to move around bit widths
    * eg. if not many shifts, but are often spilling to memory, it may be a good idea to expand the register fields
  - on the other hand, a hardware impact of increasing register file size is increased latency
    * *more* hardware in order to implement the register file
- pipelining and parllelism sidenote:
  - pipelining in hardware:
    * pipelining the actual hardware components, eg. staggering frontend / backend instruction execution at the same time
  - pipelining in software:
    * software pipelining acts more as a compiler optimization
    * although the *hardware* isn't being overlapped, the code is, and compiler has more layered code to optimize
  - parallelism in hardware:
    * designing hardware components that can run at the same the time
  - pipelining in software:
    * using multithreading

Arithmetic Operations

- in **three-op code**, two sources and one destination are given:
  - all arithmetic operations follow this form
  - eg. in `add a, b, c` , `a` gets `b + c`

Translating `f = (g+h) - (i+j)` :

```
; "pseudocode" that abstracts memory locations of identifiers
add t0, g, h ; uses temporary registers
add t1, i, j
sub f, t0, t1
```

- register operands are often used in arithmetic operations:

  - as part of memory hierarchy, **registers** are used for frequently accessed data:
    * smaller memory is typically faster, related to physical design of the wired delay
    * much faster than main memory
    * very important for compilers to perform **register allocation** to use registers as much as possible
  - in MIPS, registers numbered from 0 to 31 in a 32 by 32-bit register file
  - `t0...t9` for temporaries, `s0...s7` for saved variables
    * note that 5 bits (to store 0 to 31) are used to store *which* location the desired *32-bit* address is in

Memory Operands

- since main memory is mainly used for large data (eg. arrays, structures):
  - memory values must be *loaded* into registers
  - results must then be *stored* from registers to memory
- MIPS specifics:
  - memory is byte addressed
    * though larger values may be pulled in from memory than just a byte (eg. a word or 4 bytes)
  - addresses must be a multiple of 4
  - MIPS is big endian
- memory reference operands take in a static offset:
  - I-type format takes two registers and an immediate
  - eg. in `lw t0, 32(s3)`, `rt = t0` gets the contents of the address at `rs = s3` offset by `i = 32` bytes
  - eg. while `sw t0, 16(s3)`, the contents of `rt = t0` are stored into the address at `rs = s3` offset by `i = 16` bytes
  - unlike most instructions, `lw, sw` have *non-deterministic* latency since they have to go into main memory
    * introduces a lot of different problems with scheduling, etc.

Translating `g = h + A[8]`:

```
; g in s1, h in s2, base address of A in s3
lw  t0, 32(s3) ; 4 bytes per word, index 8 requires offset of 32
add s1, s2, t0
```

Translating `A[12] = h + A[8]`:

```
lw  t0, 32(s3) ; load word
add t0, s2, t0
sw  t0, 48(s3) ; store word *back* into memory
```

Immediate Operands

- immediates are *constant* data specified in an instruction
  - ie. literal values that will not change
  - eg. `addi s3, s3, 4` or `addi s2, s1, -1`
  - no subtract immediate instruction (minimizing instructions)
  - has a limit of 16 bits, in order to make the common case fast and avoid a load

Other Operations

- logical ops:
  - `<<` is `sll`
    - `shamt` instruction field specifies how many positions to shift
  - `>>` is `srl`
  - `&` is `and, andi`
    - 3-op code, two sources, one destination
  - `|` is `or, ori`
  - `~` is `nor`
- sometimes, a 32-bit constant is needed instead of the typical 16-bit immediate:
  - `lui rt, constant` is the load-upper-immediate instruction:
    - copies the 16-bit constant to the left 16 bits of `rt`
    - while clearing the right 16 bits of `rt` to 0
  - `ori rd, rt, constant` is the or-immediate instruction:
    - can use to load the lower 16-bits of the 32-bit constant
  - together, can be used to form a 32-bit constant
    - now can be used for arithmetic operations, or for jumping to a larger 32-bit address

Conditional Operations

- with branch instructions, control of the program is conditioned on data:
  - ie. modifying the program counter out of seqeuence
  - `beq rs, rt, L1` will branch to the instruction labeled `L1` if `rs == rt`
  - `bne rs, rt, L1` will branch to the instruction labeled `L1` if `rs ≠ rt`
  - `j L1` is an unconditional jump to the instruction labeled `L1`
- the branch instructions `beq, bne` are both I-type instructions that only take a 16-bit immediate:
  - but the PC holds a 32-bit address

  - most branches don't go very far, so PC-relative addressing is used (forward or backward):
    * *not-taken* address is `PC + 4`
    * target ie. *taken* address is `(PC + 4) + offset * 4` where `offset` is the immediate value:
      · instructions are always on a 32-bit granularity, so `offset * 4` allows an 18-bit address to be formed from 16 bits of space using word-level addressing (rather than byte-level addressing of `lw, sw`)
      · using a sign extension plus shift by two
    * initial `PC + 4` is a convention since the program counter has already been incremented
  - note that the labels given to the instructions become encoded as immediates during linking based on the layout in memory
  - if the branch target is too far to encode in a 16-bit immediate, assembler will rewrite the code using a jump

Branching far away:

```
    beq s0, s1, L1
    ; becomes rewritten as
    bne s0, s1, L2
    j L1
L2:
```

- the jump instructions `j, jal` are J-type instructions that take a 26-bit immediate:
  - no need for registers for branch test
  - need 32-bit address for PC:
    * like the branch instructions, the address is a word address ie. really a 28-bit address
      · all PC-related instructions use word-level addressing
    * upper 4 bits are taken from the current PC
    * this is called **pseudodirect addressing**
  - `jal` sets register `ra` to `PC + 4` as part of its execution:
    * this is an example of an explicit operand
  - alternatively, `jr` is an R-type instruction that jumps to the 32-bit address in a register:
    * allows for a full 32-bit PC jump
      · requires something like a `lw` or a `lui, ori` to build the full address up before the `jr`
    * but may unecessarily pollute registers, so `j` is provided as another way to jump
- summary of addressing modes:

1. **immediate addressing** using an immediate in the instruction
2. **register addressing** that uses the address stored in a register
3. **base addressing** that adds a register address base with an immediate address
4. **PC-relative addressing** that adds the current PC with an immediate address
5. **pseudodirect addressing** that concatenates the current PC with an immediate address

- a **basic block** is a sequence of instructions with:
  - no embedded branches (except at the end)
  - no branch targets (except at beginning)
  - compiler can treat this block as a *coarser* granularity of a basic unit for optimizations

Translating C `if-else` :

```
if (i==j) f = g+h;
else f = g-h;
```

To MIPS:

```
      bne s3, s4, Else
      add s0, s1, s2
      j Exit ; assembler will calculate addresses of labels
Else: sub s0, s1, s2
Exit: ...
```

Translating C `while` :

```
while (save[i] == k) i+=1;
```

To MIPS:

```
; i in s3, k in s5, base address of save in s6
Loop: sll  t1, s3, 2 ; array of 4 bytes
      add  t1, t1, s6
      lw   t0, 0(t1)
      bne  t0, s5, Exit
      addi s3, s3, 1
      j    Loop
Exit: ...
```

- other conditional operations:
  - `slt rd, rs, rt` will set `rd = 1` if `rs < rt` else `rd = 0`
  - `slti rd, rs, constant` will set `rd = 1` if `rs < constant` else `rd = 0`
    - also `sltu, sltui` for unsigned comparisons

  - often used in combination with branch equations
  - `blt, bge` are not real instructions since the hardware would be much slower, penalizing with a slower clock:
    * a good design compromise from optimizing for the common case
    * but these are pseudo instructions that will later be compiled down into real instructions

Procedures

- procedure call steps:
  1. place parameters in registers
  2. transfer control to the procedure
  3. acquire storage for procedure on stack
  4. execute procedure code
  5. place return value in register for caller
  6. return to address of call
- some register usages:
  - `a0-a3` are arguments
    * rest of arguments are stored on the stack
  - `v0, v1` are result values
  - `t0-t9` are temporaries
  - `s0-s7` are callee-saved
  - `gp` is the global pointer for static data
  - `sp` is the stack pointer
  - `fp` is the frame pointer
  - `ra` is the return address
- procedure call instructions:
  - `jal ProcedureLabel` jump and links:
    * address of the following instruction is put in `ra`
    * jumps to target address at procedure label
  - `jr ra` jumps to a register:
    * copies `ra` to program counter (could use another register)
    * can also be used for computed jumps eg. `switch` statements

Translating C function:

```
int foo(int g, h, i, j) {
  int f;
  f = (g+h) - (i+j);
  return f
}
```

To MIPS:

```
foo:
    addi sp, sp, -4   ; push stack
    sw   s0, 0(sp)    ; saving callee-saved register
    add  t0, a0, a1
    add  t1, a2, a3
    sub  s0, t0, t1
    add  v0, s0, zero ; $zero register is always 0
    addi sp, sp, 4    ; pop stack
    jr   ra
```

Translating recursive C function:

```
int fact(int n) {
  if (n < 1) return 1;
  else return n * fact(n-1);
}
```

To MIPS:

```
fact:
    addi sp, sp, -8
    sw   ra, 4(sp)   ; recursive, need to save return
    sw   a0, 0(sp)   ; save arg
    slti t0, a0, 1   ; test n < 1
    beq  t0, zero, L1
    addi v0, zero, 1 ; return 1
    addi sp, sp 8
    jr   ra
L1: addi a0, a0, -1
    jal  fact
    lw   a0, 0(sp)   ; restore arg
    lw   ra, 4(sp)   ; restore return
    addi sp, sp, 8
    mul  v0, a0, v0  ; multiply
    jr   ra
```

Representation

- instructions are encoded in binary, AKA machine code
- MIPS instructions:

  - encoded as 32-bit instruction words
  - opcode and register (5-bit) formats
  - very regular

- MIPS R-type instruction fields:
  - 6-bit `op` for opcode:
    - specifies the format of the instruction
    - always `0` in R-format
  - 5-bit `rs` for first source register number
  - 5-bit `rt` for second source register number
  - 5-bit `rd` for destination register number
  - 5-bit `shamt` for shift amount
  - 6-bit `funct` for function code (extends opcode):
    - specific function to perform when in R-type
    - allows the opcode field to be as short as possible, while allowing a specific instruction format to have extra functionality:
      - increase decoding complexity while allowing for more functionality
      - is possible since the class of R-type instructions don't have a large immediate field

Representing the instruction `add $t0, $s1, $s2` :

```
000000 10001 10010 01000 00000 100000
^R-type opcode           ^no shift
       ^s1   ^s2   ^t0         ^32 represents add function (part of ISA)
```

- MIPS I-format instruction fields:
  - 6-bit `op` for opcode:
    - specifies the format of the instruction
  - 5-bit `rs` for source register number
  - 5-bit `rt` for source *or* destination register number
  - 16-bit constant or address
    - used for immediates
  - supporting different formats complicate decoding, but want to keep formats as similar as possible
- segments in the memory layout:
  - reserved segment
  - **text** segment containing program code:
    - instructions themselves lie *in* memory
    - program pointer points somewhere here
  - **static** data segment for global variables eg. static variables, constants, etc.
    - global pointer points here

- **heap** for dynamic data eg. created by `malloc` in C or `new` in Java
- **stack** for function frames ie. automatic storage
  * stack frame pointer point here

Arithmetic

- integer addition:
  - addition is done bitwise
  - **overflow** may occur if the result is out of range:
    - adding a positive and negative operand never causes overflow
    - adding two positive operands overflows if the result sign is 1
    - adding two negative operands overflows if the result sign is 0
- integer subtraction:
  - comparable to addition, just add the negation of second operand (flip all bits and add 1)
  - overflow can again occur:
    - subtracting two operands of the same size never causes overflow
    - subtracting positive from negative operand overflows if result sign is 0
    - subtracting negative from positive operand overflows if result sign is 1
- dealing with overflow:
  - some languages such as C ignore overflow
    - would directly use MIPS `addu, addui, subu`
  - while other languages like Ada, Fortran raise exceptions:
    - would use MIPS `add, addi, sub`

ALU

- the **arithmetic logic unit (ALU)** in the CPU handles arithmetic in the computer:
  - eg. addition, subtraction, multiplication, devision
  - must handle overflow as well as floating point numbers
  - in the CPU pipeline, ALU is used during instruction execution
    - after instruction fetch / decode and operand fetch
- general ALU inputs and outputs:
  - two inputs `a` and `b`
  - some way to specify a specific ALU operation
    - this ALU `op` comes from both the instruction `opcode` and `func` field together being interpreted by a secondary controller
  - four outputs `CarryOut, Zero, Result, Overflow`
    - `Zero` is operation independent, used for branch conditions
- Figure 1 indicates a 1-bit implementation of an ALU:
  - contains an `AND` gate, `OR` gate, and a 1-bit adder

  - note that *all* three operations are performed at once, and a multiplexer is used to select one result

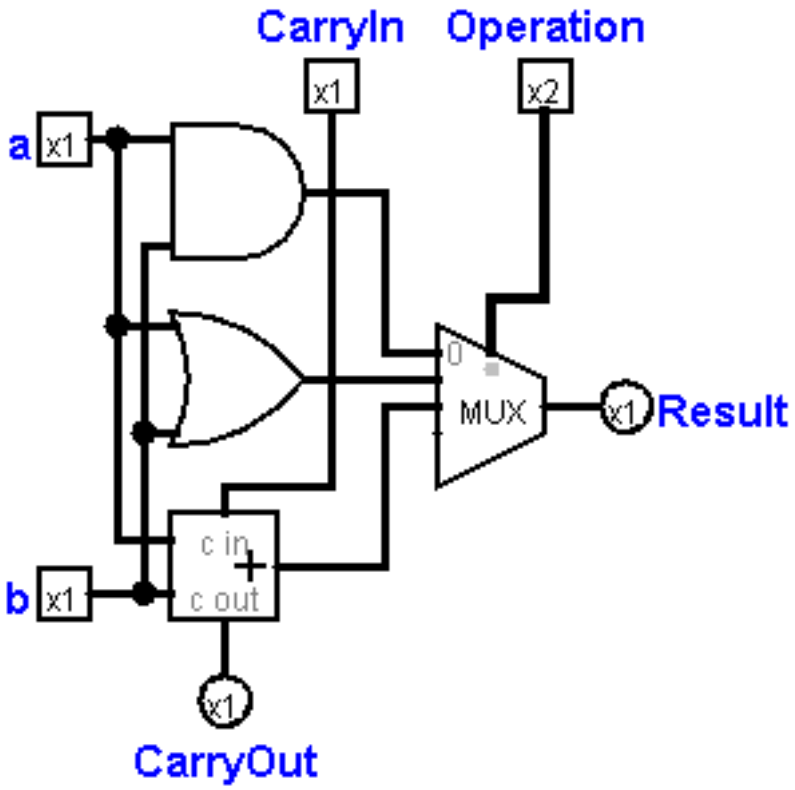

Figure 1: 1-bit ALU

- a 1-bit **full adder** AKA a (3,2) adder:
  - adds two 1-bit inputs
  - a **half adder** would have no `CarryIn` input

From its truth table, the full adder follows the boolean equations:

```
CarryOut =  (b & CarryIn) | (a & CarryIn) | (a & b) // sum of products
Sum =  (!a &!b & CarryIn) | (!a & b & !CarryIn) |
      (a & !b & !CarryIn) | (a & b & CarryIn)
      // Alternatively, a 3 input XOR (outputs 1 when odd number of input 1s).
```

- we can chain together 1-bit ALUs to create a 32-bit ALU:
  - called a **ripple carry ALU**
  - each 1-bit ALU handles one of the places of the computation
  - each `CarryOut` gets propagated to the next place's `CarryIn`
    - there is an overall delay associated with this propagation
  - gives an overall `CarryOut` and 32-bit result
- to handle subtraction:
  - as in Figure 2, a `Binvert` signal can select the flipped bits of input `B`

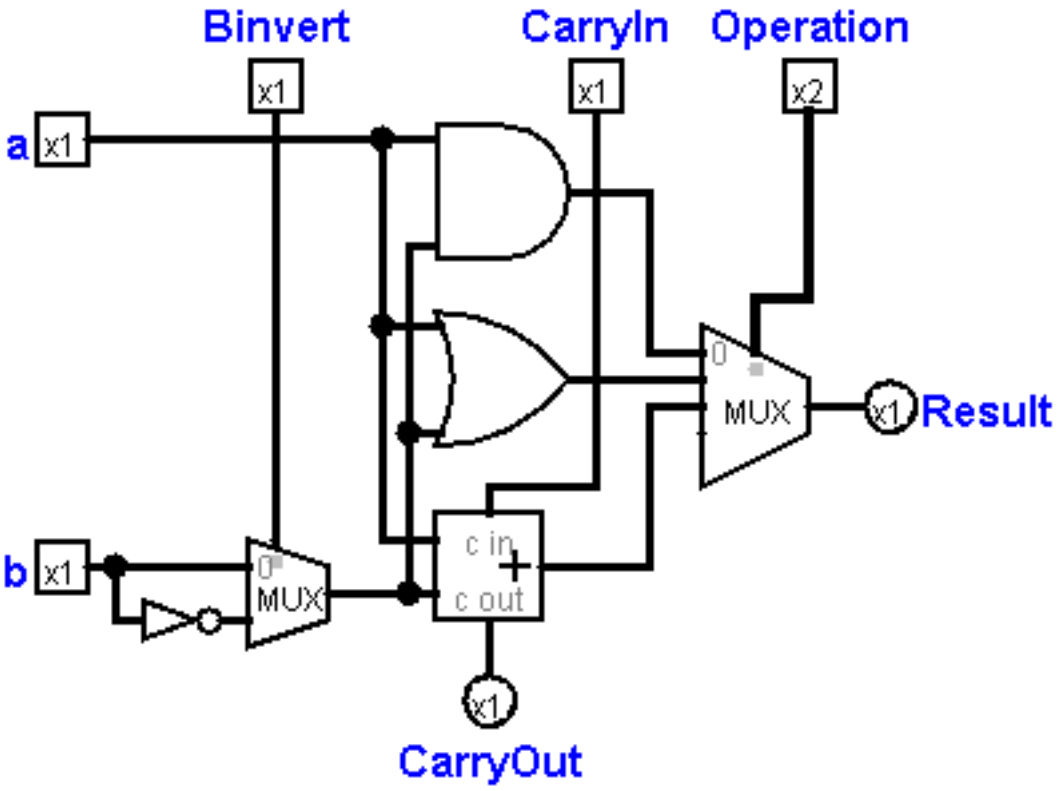

Figure 2: 1-bit ALU with Subtraction

  - but still need to add 1 to properly negate the input, so ALU0 would have a `CarryIn` set to 1 during subtraction
  - adding a `Ainvert` signal for inverting `A` as well allows the ALU to perform a `NOR` ie. `~A & ~B`
- to detect overflow in an n bit ALU:
  - need to compare the carry-in and out of the most significant bit
  - `Overflow = CarryIn[n-1] XOR CarryOut[n-1]`
- to detect zero:
  - `Zero = (Res_0 + Res_1 + ... + Res_{n-1})`
  - use one large `NOR` gate
- to implement the `slt` or set-on-less-than instruction as in Figure 3:
  - need to produce a 1 in `rd` if `rs < rt` , else 0
    - all but least significant bit of the result in a `slt` is always 0
  - to compare registers, can use subtraction through `rs - rt < 0`
  - add additional input `Less` and output `Set` :
    - `Less` acts as a passthrough that can be selected as another operation
    - `Set` appears as an output only on the most significant bit and gives the output of the adder
      - ie. after the subtraction, `Set` will hold the sign bit of the result
  - `set` in MSB ALU is then connected *back* to the `Less` of the LSB ALU:

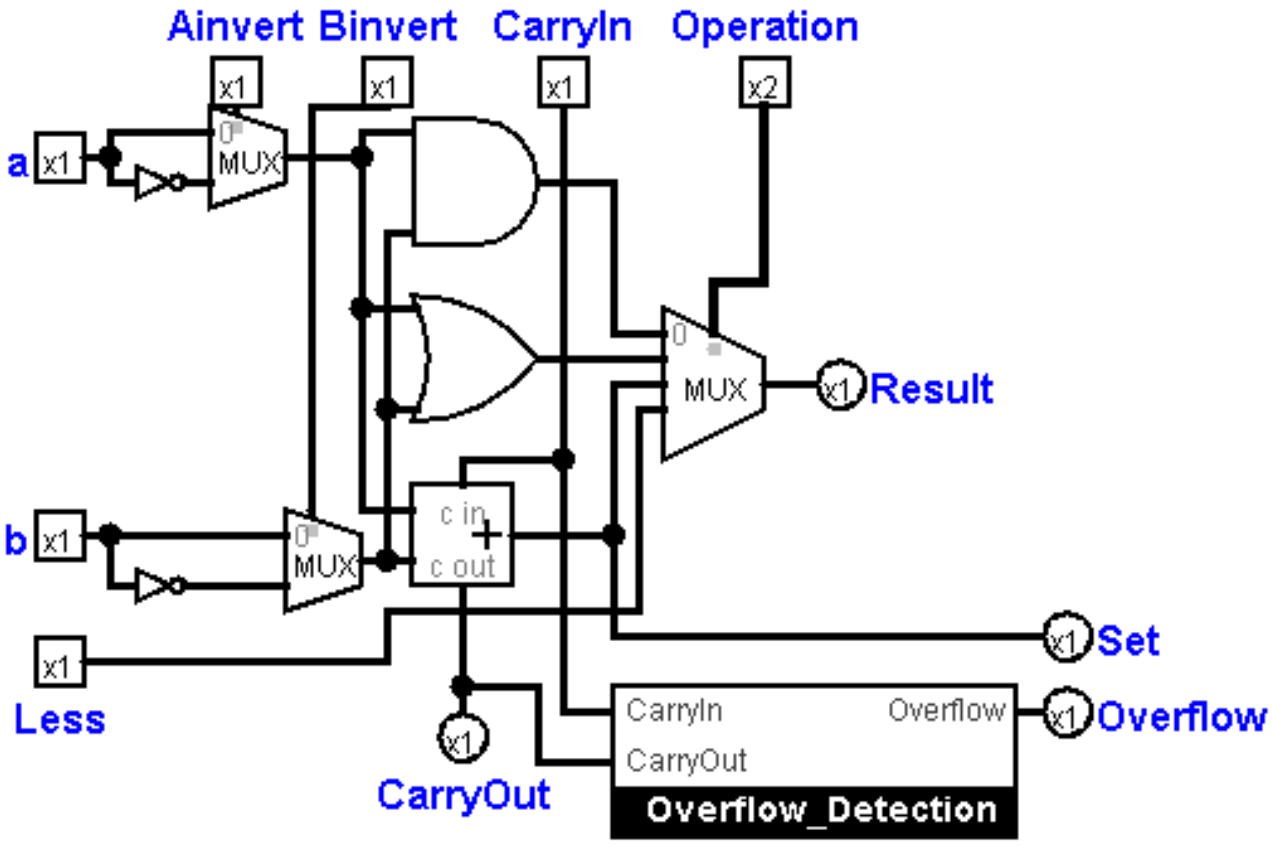

Figure 3: 1-bit ALU handling slt (`Set, Overflow` circuitry only appear in most significant ALU)

    - where as all the other `Less` inputs are hardwired to zero
        - the flexibility of the ALU to perform various operations requires every bit to be specified
    - thus only the least significant bit in `slt` will vary depending on the result of `rs - rt`
    - note this leads to a long delay before `slt` result becomes stable
        - exposes a race condition when pipelining
- another possibility is to just perform a subtraction and *defer* the `slt`:
    - would require addition instruction logic like `beq, bne`

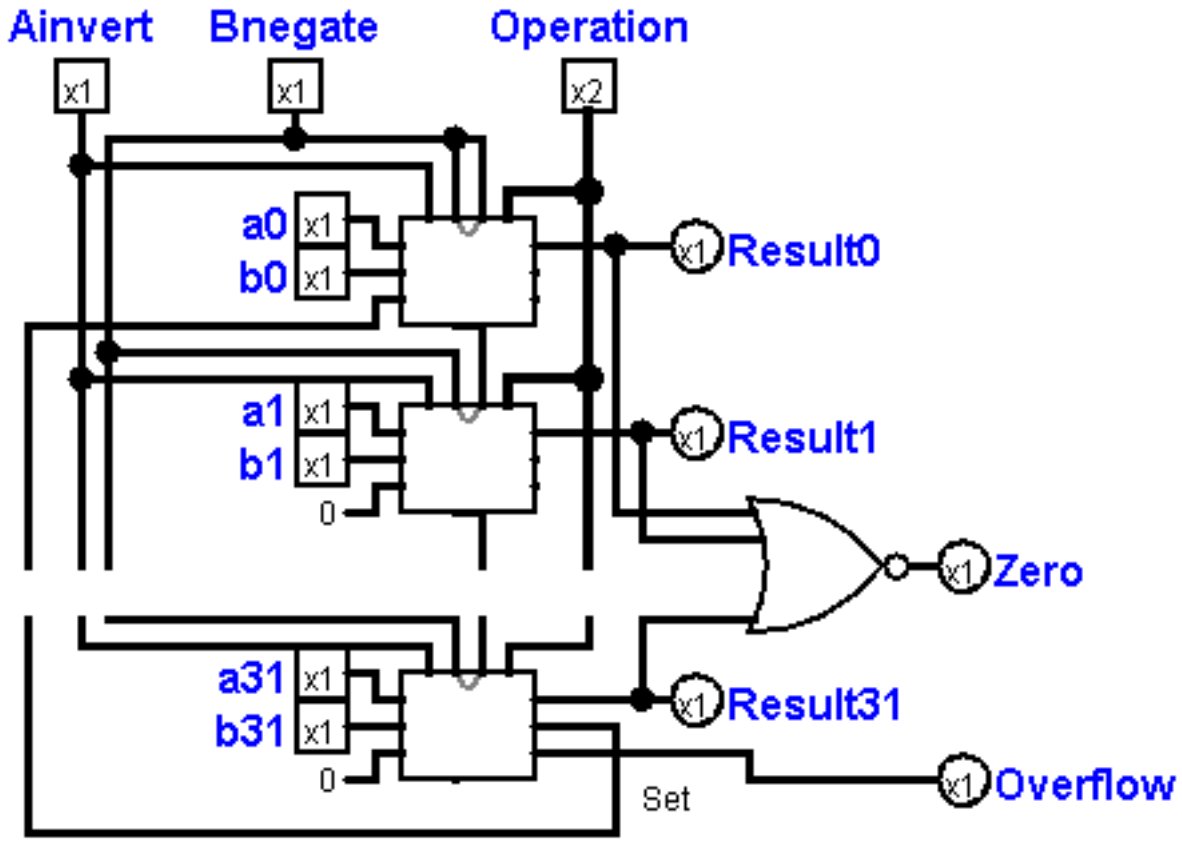

Figure 4: Final 32-bit ALU

- thus the full 32-bit ALU can be implemented as in Figure 4:
    - `Bnegate` hooks up `Binvert` with the `CarryIn` of ALU0
        - since in a `NOR`, setting the carry-in is a no-op
    - altogether, can perform the operations `ADD, SUB, AND, OR, NOR, slt`

Adder Optimizations

- the adder has the longest delay in the ALU:
    - particularly, want to examine the carry chain in the adder:

    - each `CarryOut` calculation must go through 2 gate delays (sum of products calculation)
    - leads to a delay of $2n$
  - how can we lessen this delay?
- timing analysis of a normal ripple-carry adder:
  1. `C1` has delay $5T$ due to sum of products calculation with a triple-OR
     - calculating the sum at bit 1 requires an additional $2T$
  2. `C2` has delay $10T$, with the same structure as `C1`
  3. `C3` has delay $15T$
  4. etc.

Table 1: Carry Look Ahead Logic

A	B	CarryOut	Type
0	0	0	kill
0	1	CarryIn	propagate
1	0	CarryIn	propagate
1	1	1	generate

- in a **carry look ahead** adder, generate the `CarryOut` ahead of time:
  - use logic on the side ie. trade area for parallel performance
  - eg. as seen in Table 1:
    - when `A, B` are both 0, *regardless* of what the carry-in is, a carry-out cannot be generated
    - similarly, when `A, B` are both 1, a carry-out will *always* be generated
    - when `A, B` are different, then the carry-in will be propgated
  - thus calculating the carry-out without considering the carry-in in some scenarios
  - augment each adder with two signals `G = A & B` and `P = A XOR B`
    - the `CarryIn` of one adder is no longer connected to the `CarryOut` of the next
    - there is no more delay *between* the adders, since each `CarryOut` can be calculated purely in terms of `G, P` which depend on `A, B`
  - *pros*:
    - all `G, P` can be calculated in parallel, and then all carry-outs can be calculated in parallel
  - *cons*:
    - the growing fan-in of the sum of products calculation can become quickly expensive area-wise
- timing analysis of a flat CLA adder, given assumption the delay of a gate to stabilize is proportional to its fan-in:
  - every `G, P` have delay $2T$, since they only depend on `A, B`

1. `C1` has delay $6T$, since it sums up `G0` and `C0*P0`, which have delay $2T$ and $4T$ respectively
   - to actually perform the sum at bit 1, requires two more `XOR` calculations, one of which uses `C1`, so the sum at bit one has delay $8T$
2. `C2` has delay $8T$, since it has a triple-OR with triple-AND in the worst case, and the sum at bit 2 has delay $10T$
3. `C3` has delay $10T$, and the sum at bit 3 has delay $12T$
   - vs. in ripple carry, sum requires $15T + 2T = 17T$
4. etc.

Carry-out calculations in a look ahead adder:

```
C1 = G0 + C0*P0 // generate at 0, or propagate earliest carry-in
C2 = G1 + G0*P1 + C0*P0*P1 // generate at 1, or propagate at 1 AND generate at 0,
                           // or propagate at 1 AND propagate earliest carry-in
C3 = G2 + G1*P2 + G0*P1*P2 + C0*P0*P1*P2
// etc.
// Note how C0 is the only carry-in that appears in calculations.
// Otherwise, the calculations rely solely on G, P.
```

- multiple loohakead adders can be connected together:
  - eg. four 8-bit carry lookahead adders can form a 32-bit **partial carry lookahead** adder
  - ie. a rippling of lookahead adders
- another way to chain together lookahead adders using **hierarchical CLA**:
  - create more *layers* of generate and propagate values `G_a, P_a`
  - eg. to get the overall `G_a, P_a` for a unit of four 1-bit adders, we can use the following equations:
    - `G_a = G0*P1*P2*P3 + G1*P2*P3 + G2*P3 + G3` ie. generate at one of the adders, and propagate through the rest
    - `P_a = P0*P1*P2*P3` ie. propagate through all adders
  - thus we can add another hierarchy of carry lookaheads
    - identical logic to a normal CLA, but using a different *layer* of `G, P`
  - *pros*:
    - helps mitigates the expensive area cost of the normal CLA fan-in growth
- in a **carry select adder**, trade even more area for parallel performance:
  1. calculate adder results for *both* possible carry-ins, 1 and 0
     - twice as many ALUs for redundant calculations
  2. then, select one of the results depending on the actual carry-in
     - the delay of the multipexer is often less than the delay of the entire adder chain