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 the immediate `i` (which has been sign-extended from 16 to 32 bits)
  - eg. while `sw t0, 16(s3)`, the contents of `rt = t0` are stored into the address at `rs = s3` offset by the sign-extended immediate `i`
  - 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

- used for all arithmetic and logic instructions

- MIPS I-type instruction fields:
  - 6-bit `op`
  - 5-bit `rs`
  - 5-bit `rt`
  - 16-bit immediate value
  - used for branches, immediates, data transfer

- MIPS J-type instruction fields:
  - 6-bit `op`
  - 26-bit target address
  - used for jump instructions

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

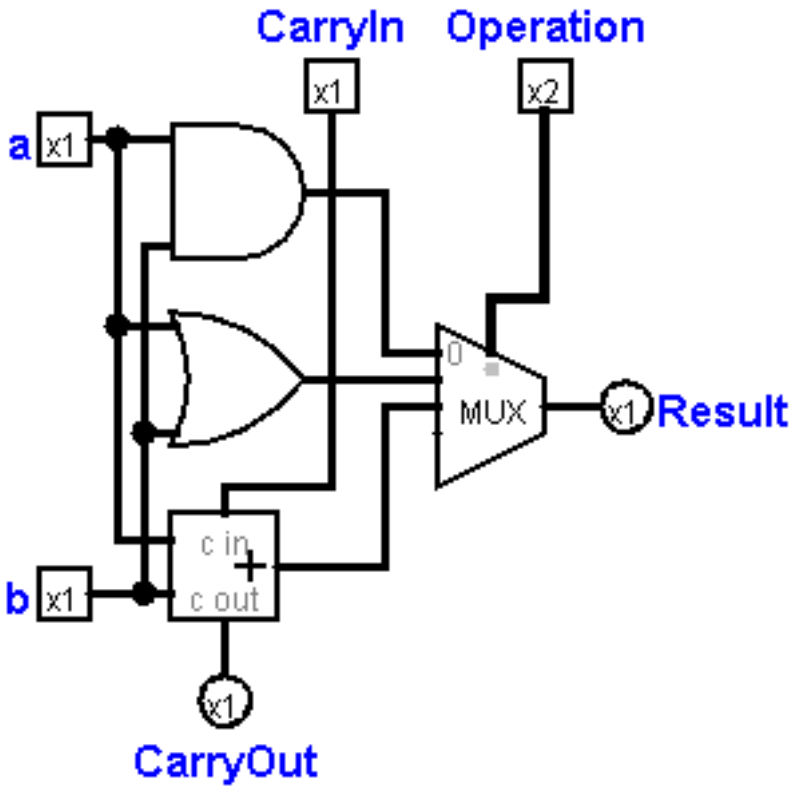

Figure 1: 1-bit ALU

- 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`

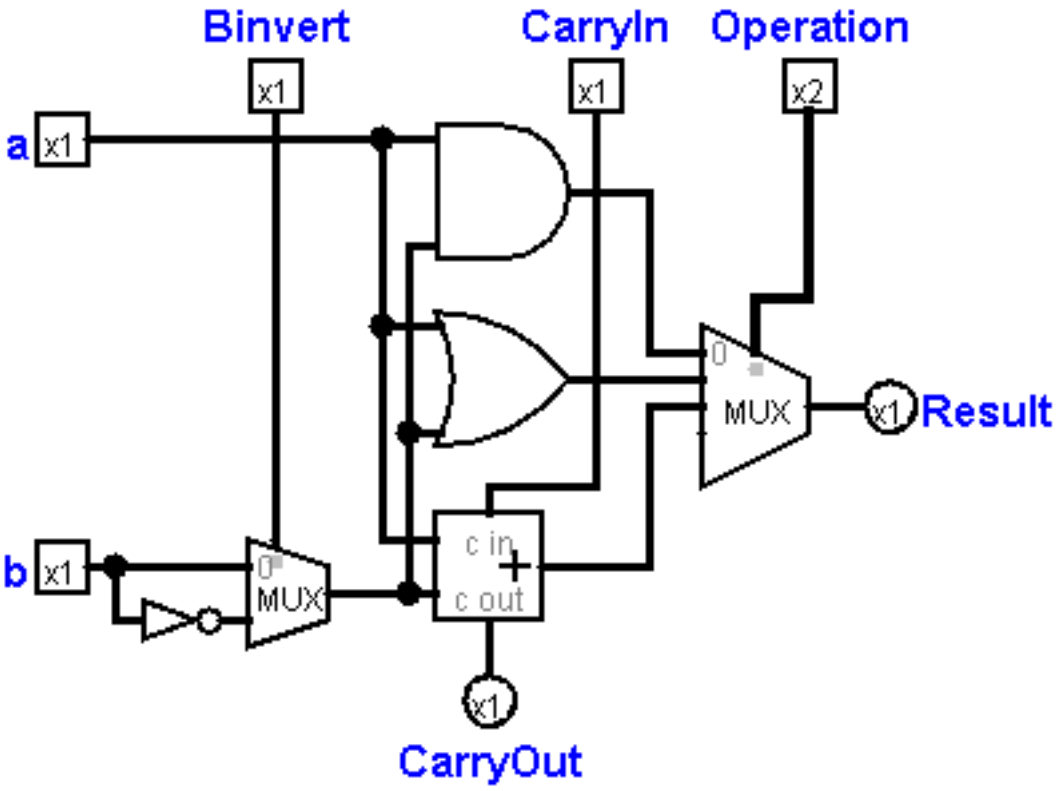

Figure 2: 1-bit ALU with Subtraction

  - 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:

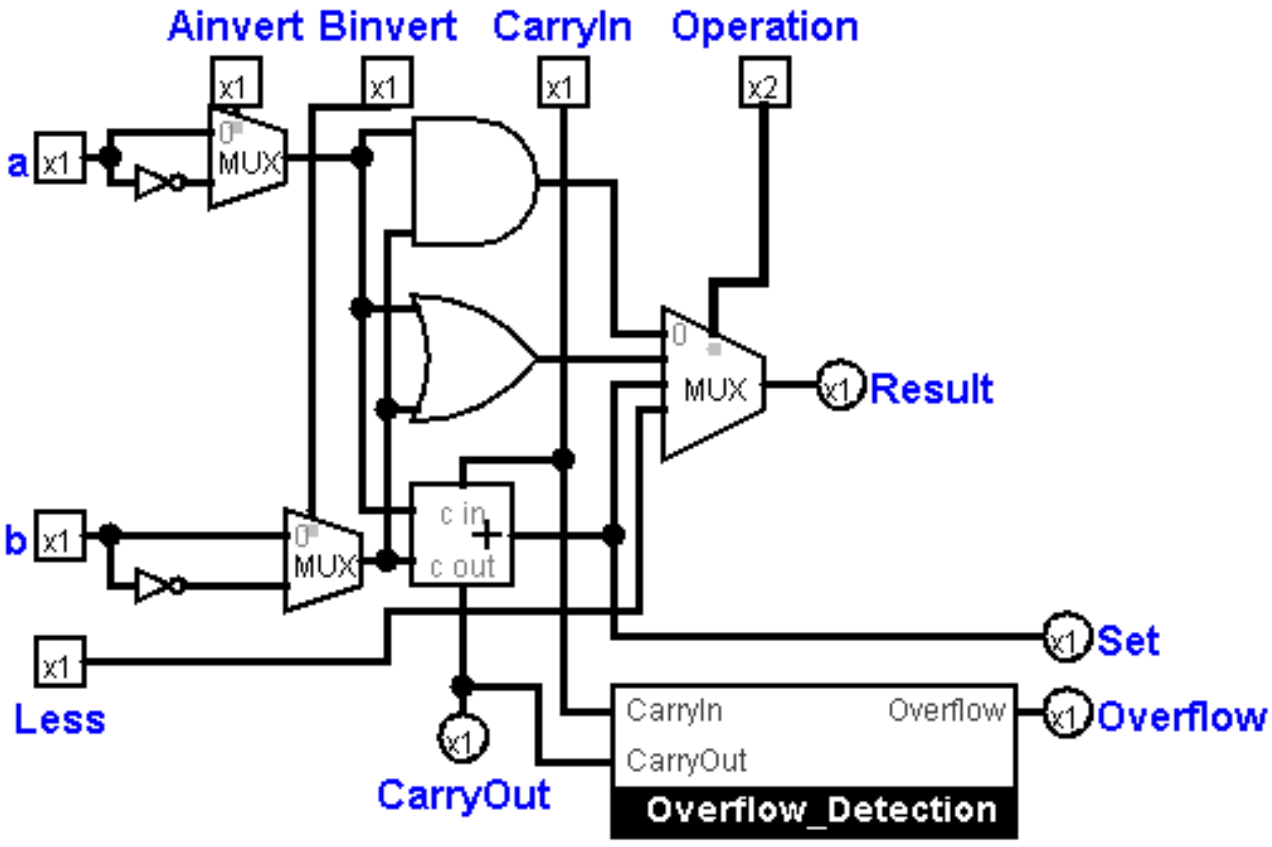

Figure 3: 1-bit ALU handling slt (`Set, Overflow` circuitry only appear in most significant ALU)

    - 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`

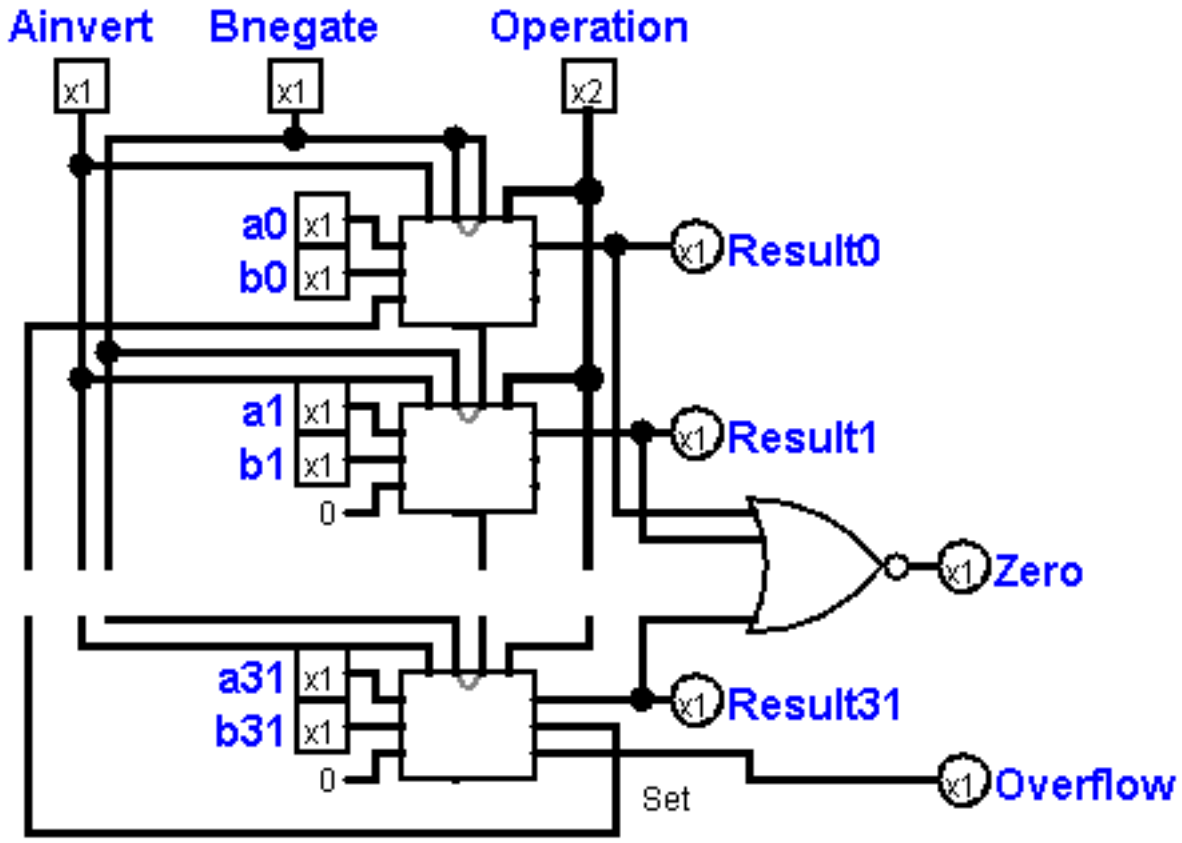

Figure 4: Final 32-bit ALU

- 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 (HCLA)**:
  - 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
    - note that `G_a, P_a` are *independent* of carry-ins
  - then we can calculate a higher level of carry-ins that feed back into the 4-bit adders:
    - eg. can calculate `C4, C8, C12, ...` from `G_a, P_a, G_b, P_b, ...`
    - eg. `C4 = G_a + C0*P_a` and `C8 = G_b + G_a*P_b + C0*P_a*P_b`
    - note that at this higher level, the the delays of `G_a, G_b, ...` are the same and the delays of `P_a, P_b, ...` are the same, similarly to the `G, P` delays at the lowest levels
  - 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

Table 2: Granularity of Dependecies

Pass 1	Pass 2	Pass 3	Pass 4	Pass 5
S_0	G_j, P_j	$C_4, C_8,$	$C_5{-}C_7, C_9{-}C_{11}, C_{13}{-}C_{15}$	$S_5{-}S_7, S_9{-}S_{11},$
G_i, P_i	$C_1{-}C_3$	C_{12}, C_{16}	S_4, S_8, S_{12}	$S_{13}{-}S_{15}$
		$S_1{-}S_3$	S_4, S_8, S_{12}	

- can also use a **partial CLA**:
  - ripple carry two CLAs
  - *pros*:
    - simpler design with less area
  - *cons*:
    - not as fast as hierarchical CLA
    - since they are rippled together, there is a dependence between each CLA that causes additional delay
- 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
- timing analysis of a CSA, given the delay of a gate is proportional to its fan-in:
  - have to use MUXs to choose between the carry-in calculations:
    - need to choose correct carry-out and correct sum
    - sum calculation will usually dominate the carry-out calculation
  - however the main carry-in will have a delay of 0 since both possibilities are calculated
    - this delay reduction is emphasized when CSA is used at the end of a addition chain

Multiplication

- starting with the long multiplication approach for binary:
  - analagous to long multiplication in base-10 ie. sequence of adds and shifts
  - the length of the product is the sum of operand lengths
    - eg. 32-bit multiplier would use a 64-bit multiplicand and have a 64-bit product

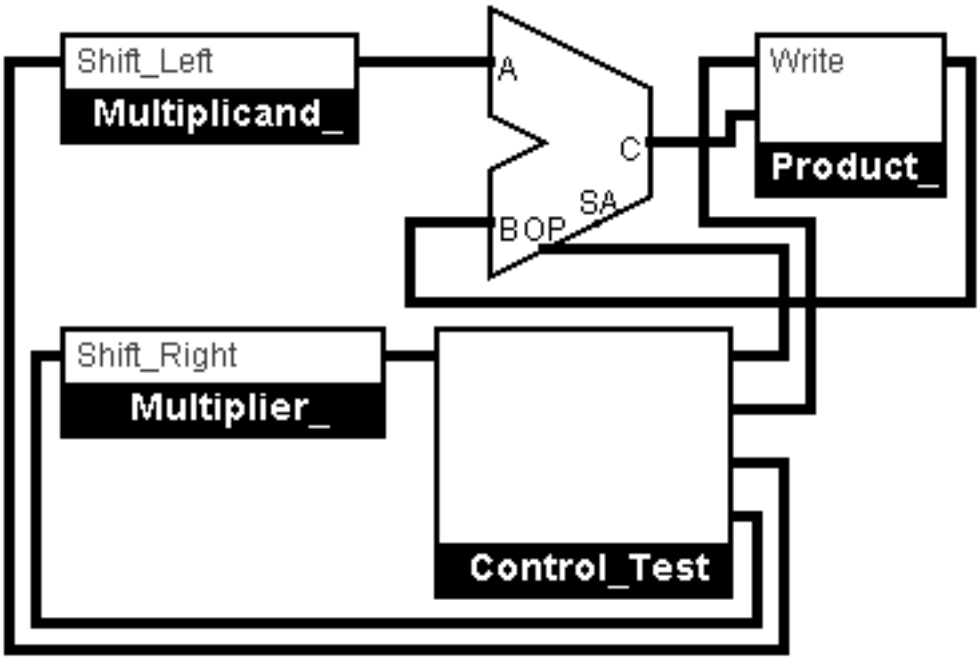

Figure 5: Multiplication Hardware with a 64-bit ALU

- datapath for Figure 5:
  - product is initially zero
  1. test the least significant bit of the multiplier
     - if 1, add multiplicand to product and place the result in the `Product` register
  2. shift the `Multiplicand` register left by 1 bit
  3. shift the `Multiplier` register right by 1 bit
     - gets new least significant bit
  4. if this is the 32nd repeititon, we are done, otherwise continue to loop
- can we optimize the ALU down from a 64-bit ALU to a 32-bit one?
  - would save latency and power on the ALU
  - yes, as seen in Figure 6, use a 32-bit ALU and fix the multiplicand at 32 bits, instead of shifting it to the left:
    - instead, equivalently shift the *product* register (which remains at 64 bits) to the *right*
    - ie. add into the *upper* bits of the product, and then shift that to the right
  - in addition, remove the multiplier shifting and load it directly into the lower 32 bits of the product:
    - while still zeroing out upper bits of the product
    - thus as we shift the product register right, we strip off the individual bits of the multiplier on the right side of the product and use them in the control test
  - the control test thus handles:
    - whether we write to the register (if stripped bit is 1)

    * performs a right-shift of the product every iteration

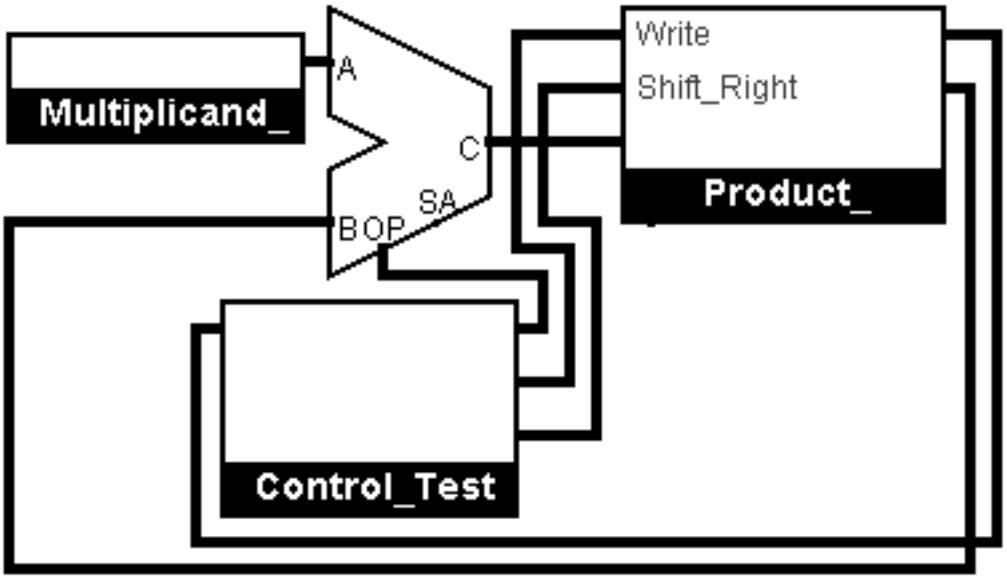

Figure 6: Multiplication Hardware with a 32-bit ALU

- MIPS multiplication details:
  - there are two 32-bit registers for the product, `HI, LO` respectively
  - `mult rs, rt` and `multu rs, rt` builds a 64-bit product in `HI/LO`
  - `mfhi rd` and `mflo rd` move from `HI/LO` to `rd`
    * can test `HI` to see if product overflows 32 bits
  - `mul rd, rs, rt` sets `rd` to the least-significant 32 bits of the product

Signed Multiplication

- different approaches:
  1. make both positive, and complement final product if necessary
     - need to sign-extend partial products and subtract at the end
  2. using **Booth's algorithm**
     - uses the same hardware as before and also saves cycles
- motiviation behind Booth's algorithm:
  - want to change the sequence of additions in multiplication with some subtractions
  - eg. can compose a run of adds `14 = 2 + 4 + 8` as `14 = -2 + 16` :
    * ie. shifting the highest number by 1 and then subtracting the lowest number
    * reduces multiple additions into a single add and a subtract
- ex. to calculate `0b0010 * 0b0110` :
  - would add `0b0100` and `0b1000` (after shifts)

  - in Booth's algorithm, would instead first subtract `0b0100` and then add `0b010000` in an extra shift
  - in other examples, would end up saving more adds
    * eg. with a longer run of 1s in the multiplier
- Booth's algorithm in full:
  - in addition to keeping track of the current least significant bit, also need to track the "bit to the right" ie. the previous bit from the last iteration
  - different sequences and their meanings:
    1. `10` indicates beginning a run of 1s, so subptract
    2. `11` indicates middle of the run, so don't add or subtract
    3. `01` indicates end of the run, so add
       * this is already one order of magnitude past the original multiplicand would have been, so already accounts for the extra shift
    4. `00` indicates middleof a run of 0s, so don't add or subtract

Floating Point Format

- IEEE floating point format:
  - sign bit (1 bit)
  - exponent (8 bits single, 11 bits double)
    * a bias ensures exponent is unsigned (127 single, 1203 double)
  - fraction (23 bits single, 52 bits double)
    * normalized significand has a leading hidden bit $(1 + F)$
  - $x = (-1)^S \times (1 + F) \times 2^{E-B}$
- complicates addition:
  - 3 discrete parts
  - much repackage result after addition
- ex. $9.999 \times 10^1 + 1.610 \times 10^{-1}$
  - must align decmal points ie. shift number with smaller exponent
    * $9.999 \times^1 + 0.016 \times 10^1$
  - add significands
    * 10.015×10^1
  - normalize result and check for over/underflow:
    * assuming can only have one digit followed by three digits of precision
    * 1.0015×10^2
  - round and renormalize if necessary
    * 1.002×10^2

Processor Design

- multiple ways to optimize processor design:
  - focus on CPI ie. a single-cycle implementation
  - focus on cycle time ie. a pipelined implementation
- logic design basics:
  - information is encoded in binary
  - one wire per bit
    - multi-bit data is encoded on multiple wires
  - in a **combinational** element, output is a function of input
    - ie. operate on data
  - state ie. **sequential** elements store information:
    - eg. a register or flip-flop
    - uses a *edge-triggered* clock signal to determine when to update the stored value
    - may have an additional write control input that updates only when write-enable is high
  - combinational logic transforms data during clock cycles:
    - ie. between clock edges
    - input from state elements, output to state element
- instruction execution steps:
  1. fetch instruction from where program counter points to in instruction memory
  2. decode the instruction and read out registers based on their register numbers from the register files
  3. depending on the instruction class:
     - use ALU to calculate arithmetic result, memory addresses for load / stores, branch target address, etc.
     - access data memory for load /store
     - set PC to target address or incremet to PC + 4

- components of the processor design in Figure 7:
  - datapath components:
    1. PC
    2. instruction memory with an address input and instruction output
    3. register component with 3 register numbers as input and several outputs
    4. ALU with two numeric inputs and an output
       - input may come from the register file or directly from instruction memory as an immediate
    5. data memory with an address and data input
       - may perform either a read or write

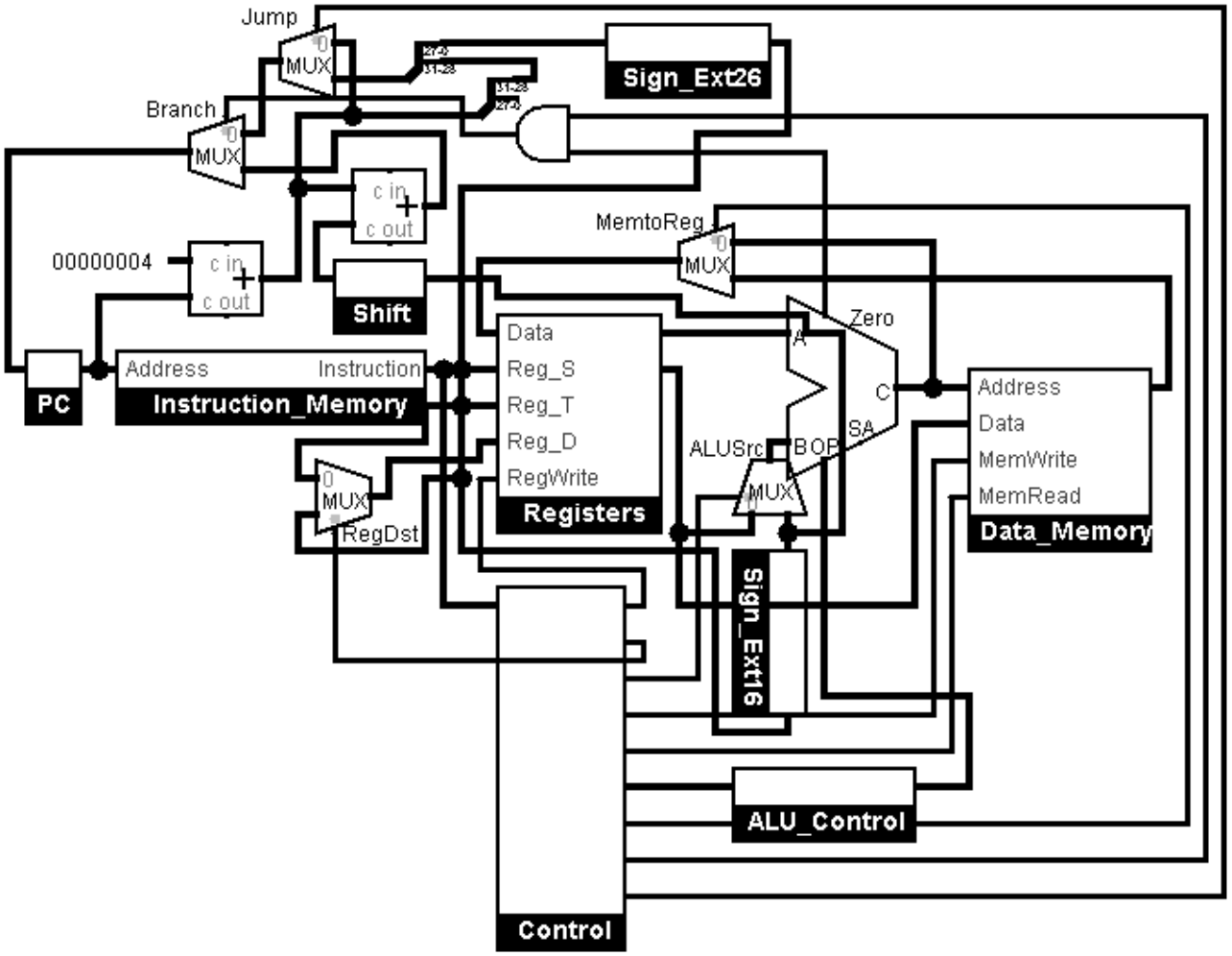

Figure 7: Processor Control and Datapath

    - data flows along the datapath
  - control path components:
    - various MUXs and operation controls
    - controls where the data goes and what operations are performed

Building a Datapath

- the **datapath** as seen in Figure 7 is composed of the elements that process data and addresses in the CPU
  - eg. registers, ALUs, MUXs, memories, etc.
- instruction fetch components:
  - the PC is a 32-bit register pointing to the next instruction to execute
    - the PC is fed to an adder that increments it by 4 for the next instruction, unless a branch otherwise changes the PC
  - the instruction memory component:
    - takes the 32-bit address on its address port and produces its corresponding instruction within the same cycle
    - ie. is an unclocked component
  - these operations occur at once
- the register component:
  - takes in 3 5-bit register numbers
  - two register numbers refer to read registers whose values are produced on the two outputs of the register component
  - the last register number refers to a register that can be written to:
    - depending on the `RegWrite` control input, analagous to the write-enable signal on flip-flops
    - the value at the write-data port is written the to register
      - write-data port is MUXed by the `MemtoReg` control between a data memory value or an ALU result
- the ALU component is the same as previously discussed in Figure 4:
  - controlled by a 4-bit operation control signal
  - note that one input is MUXed between either register data or an immediate that has been sign-extended
- the data memory component contains:
  - two inputs:
    - an address port to specify into memory
    - a write data port to optionally specify data to write into memory
  - one optional read data output port
  - a `MemWrite` control port that places a value on the write data port into memory
  - a `MemRead` control port that pulls values out of memory onto the read

data port
    * this is a new control port, was not used in the instruction memory component
- handling branch instructions:
    1. read register operands
    2. compare operands with the ALU and check zero output to see if they are equal
    3. calculate target address:
        - if jumping, sign-extend displacement, shift left 2 places, and add to PC + 4
        - MUXed by the `PCSrc` control
    - note that we need additional components to adjust the instruction offset:
        * a sign-extend component that replicates the sign-bit wire
        * a shift-left-2 component that reroutes wires
- the datapath in Figure 7 performs an instruction in one clock cycle:
    - each datapath element can only do one function at a time
        * reason why instruction and data memory is separate, so we can read an instruction and work with data memory in the same cycle
    - note that it is fine to read and write from the same source in one cycle:
        * write occurs only on the rising clock edge, after a value has been read out
        * eg. reading and writing the same register, updating PC, etc.
    - need to use multiplexers to handle alternate data sources:
        * eg. immediates vs. registers into ALU
        * data flows no matter what through the datapath, so we need to use control to indicate which parts are useful

Controlling the Datapath

- the ALU control specifies its different functions:
    - baseline 6 MIPS functions are `AND` , `OR` , add, subtract, slt, and `NOR` (all R-types)
        * control codes `0b0000` through `0b1100` respectively
    - for load and stores, need to function as an add for effective address computation
        * note that this hardwired addition operation is set by a different `ALUOp` control than the other possible R-type addition, since loads and store instructions do not have space to specify a `funct` field
    - for branch, need to function as a subtraction to compare register values
    - for other R-types, the function depends on the `funct` filed of the instruction

- Table 3 indicates how the ALU control codes may be extracted from an opcode
  - specifically, controlled by a 2-bit `ALUOp` code embedded within the overal opcode combined with the `funct` field

Table 3: ALU Control Codes Based on Opcode and Operation

Opcode	`ALUOp`	Operation	`funct`	ALU Function	ALU Control
`lw`	`00`	load word	`XXXXXX`	add	`0010`
`sw`	`00`	store word	`XXXXXX`	add	`0010`
`beq`	`01`	branch equal	`XXXXXX`	subtract	`0110`
R-type	`10`	add	`100000`	add	`0010`
R-type	`10`	subtract	`100010`	subtract	`0110`
R-type	`10`	AND	`100100`	AND	`0000`
R-type	`10`	OR	`100101`	OR	`0001`
R-type	`10`	slt	`101010`	slt	`0111`

- the main control unit must generate control signals based on three main types of instructions:
  1. R-type instructions that specify three registers, a `funct` field, etc.
     - the third register `rd` will be written to
  2. load / store instructions that specify two registers and an address immediate:
     - for loads, the second register `rt` will be written to
     - note that we want a control signal to specify whether data memory should be read from:
       - instead of always reading on every cycle and corrupting our memory cache
       - note that this is not needed for the register file due to its speed and the fact that registers are read in almost every operation
  3. branch instructions that specify two registers and an address immediate
  - the first register `rs` is always read in all three types
- main control unit signals:
  1. `RegDst` specifies whether the second or third register should be written to as a destination through `RegWrite`
  2. `Branch` specifies whether the PC will branch
  3. `MemRead` determines whether memory is being read
  4. `MemtoReg` specifies whether the memory or ALU output is sent to the write register
     - in this datapath, `MemtoReg` should only be high when `MemRead` is
  5. `ALUOp` specifies the operation for the ALU controller to use

6. `MemWrite` determines whether memory is written to
   - `MemRead` and `MemWrite` should not be high at the same time in a single-cycle datapath
7. `ALUSrc` specifies whether a sign-extended immediate or register value is fed into the ALU
8. `RegWrite` determines whether the third write register is written to

- control signals for the R-type instruction:
  1. `RegDst` should be set to 1 to choose the third register as the write destination
  2. `Branch` should be 0
  3. `MemRead` should be 0
     - R-type instructions do not access memory
  4. `MemtoReg` should be 0 to send ALU instead of memory output to the write register
  5. `ALUOp` should be `0b10` to specify the ALU operation depends on the `funct` field
  6. `MemWrite` should be 0
  7. `ALUSrc` should be 0 since we do not want to use a sign-extended immediate in the ALU
  8. `RegWrite` should be 1 since we want to write to the register file
- control signals for load instruction:
  1. `RegDst` should be set to 0 to choose the second register as the write destination
  2. `Branch` should be 0
  3. `MemRead` should be 1
  4. `MemtoReg` should be 1 to send memory output to the write register
  5. `ALUOp` should be `0b00` to specify an addition
  6. `MemWrite` should be 0
  7. `ALUSrc` should be 1 since we want to use the sign-extended immediate to build up an effective address
  8. `RegWrite` should be 1 since we want to write to the register file
- control signals for branch instruction:
  1. `RegDst` is a don't-care since we are not writing to registers
  2. `Branch` should be 1
  3. `MemRead` should be 0
  4. `MemtoReg` is a don't-care since we are not writing to registers
  5. `ALUOp` should be `0b01` to specify a subtraction
  6. `MemWrite` should be 0
  7. `ALUSrc` should be 0 since we want to compare register values
  8. `RegWrite` should be 0 since we do not write to the register file

Table 4: Main Controller Outputs

	R-format	`lw`	`sw`	`beq`	`j`
Opcode	`000000`	`100011`	`101011`	`000100`	`000010`
`RegDst`	1	0	X	X	X
`ALUSrc`	0	1	1	0	X
`MemtoReg`	0	1	X	X	X
`RegWrite`	1	1	0	0	0
`MemRead`	0	1	0	0	0
`MemWrite`	0	0	1	0	0
`Branch`	0	0	0	1	0
`ALUOp1`	1	0	0	0	X
`ALUOp2`	0	0	0	1	X
`Jump`	0	0	0	0	1

- to implement jumps:
  - jump instruction has a 26-bit address immediate
  - to generate new PC, need to concatenate top 4 bits of old PC with the left-shifted immediate
  - requires an extra control signal `Jump` decoded from the opcode
    - feeds into an additional cascaded MUX for the next PC value
- considerations on this single-cycle implementation:
  - *pros*:
    - relatively simple design
    - CPI is 1
  - *cons*:
    - cycle time must be long enough for *every* instruction to complete
      - large disparity between the amount of time different instruction types will take to execute
  - eg. branches only require instruction fetch, register access, and ALU:
    - while loads require instruction fetch, register access, ALU, memory access, another register access for write back
    - loads /stores have to go through the expensive memory hierarchy
  - more disparity as well with longer floating point operations
  - pipelined implementations are the more modern alternative that address this instruction time disparity

Datapath Extensions

Ex. Implement the I-type instruction `law` with the following pseudocode:

```
R[rt] = M[R[rs]] + SE(i)
```

- some elements of the `law` implementation already exist:
  - writing to register `rt` with `RegDst` set low
  - bringing `SE(i)` into the second ALU port with `ALUSrc` set high
- need to address:
  - having the memory read address specified entirely by a single register
    - note that we *cannot* simply add the addres at `R[rs]` to generate the address since we need the ALU in a single-cycle datapath to perform the later addition with the immediate
  - sending data memory back to the ALU
  - also have to ensure that the new opcode for `law` is *unique*

Table 5: Main Control Signals for `law`

Control		New Control	
`RegDst`	0	`LawC`	1
`ALUSrc`	1		
`ALUOp`	`00`		
`MemtoReg`	0		
`RegWrite`	1		
`MemRead`	1		
`MemWrite`	0		
`Branch`	0		
`Jump`	0		

- thus we need a new signal `LawC` to control two new MUXed modifications as seen in Figure 8:
  1. shortcircuit the data memory address with the contents of register `rs`
  2. send the read data memory port into the top ALU port
     - note that this MUX must read out the data memory *before* the `MemtoReg` MUX since the output of the ALU will eventually be sent through the MUX to write into register `rt`, as intended

Ex. Implement the I-type instruction `blt` with the following pseudocode:

```
if (R[rs] < R[rt])
    PC = PC + 4 + SES(i)
else
    PC = PC + 4
```

- very similar to existing logic for `beq` instruction:
  - `rs` and `rt` are already compared at the ALU in a `beq`

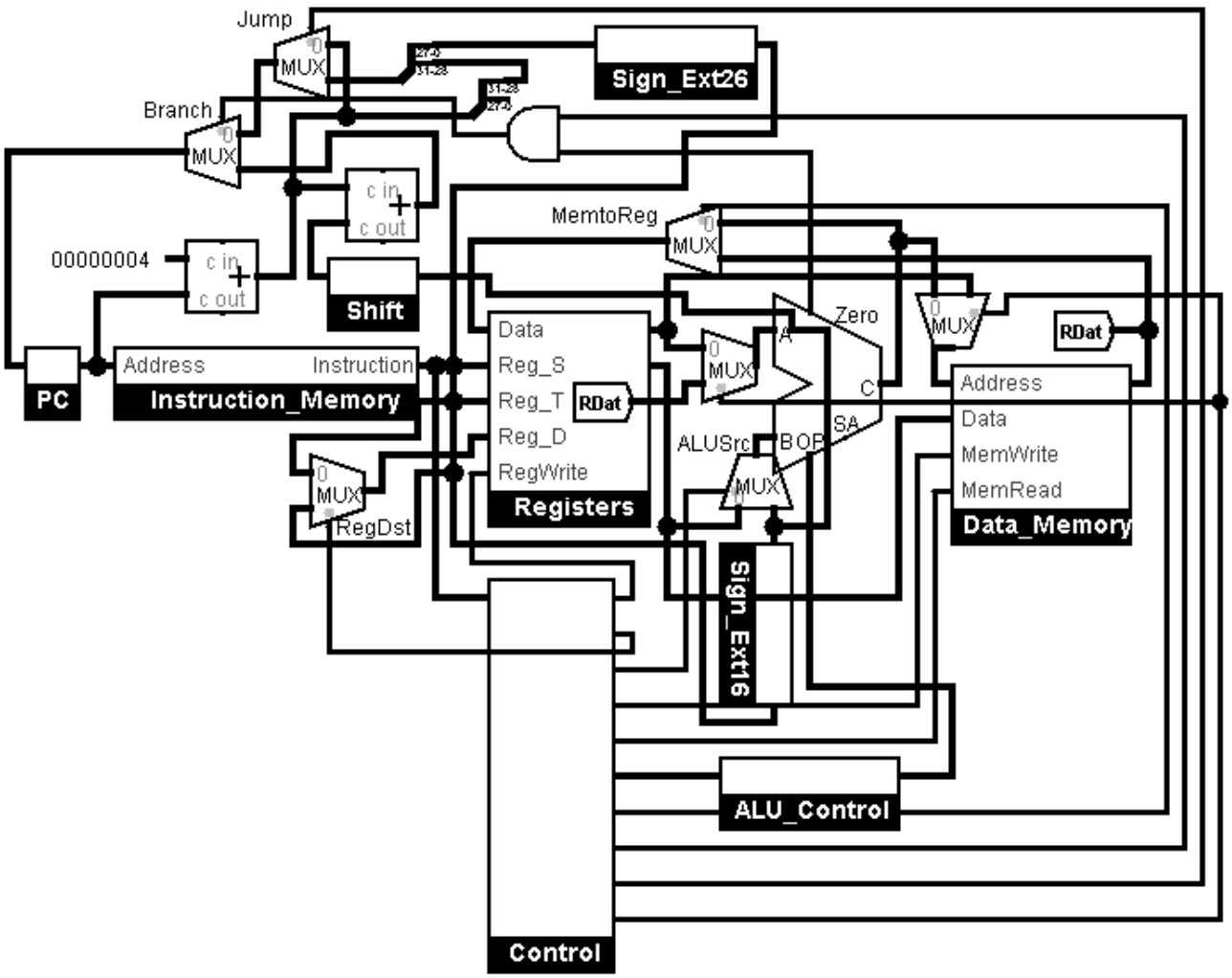

Figure 8: `law` Full Implementation

  - can also reuse the existing calculation of `PC + 4 + SES(i)`

Table 6: Main Control Signals for `blt`

Control		New / Updated Control	
`RegDst`	X	`BranchLt`	1
`ALUSrc`	0	`ALUOp`	`11`
`MemtoReg`	X		
`RegWrite`	0		
`MemRead`	0		
`MemWrite`	0		
`Branch`	X		
`Jump`	0		

- need a new mechanism to perform the `slt` *without* specifying the `funct` field
  - the R-type `slt` cannot be used because there is no space for the `funct` field in an I-type instruction.
    - unless we hardwire the `funct` field using a MUX to perform a `slt` when a `blt` is executed
- augment the main control unit with a new signal `BranchLt` as seen in Figure 9:
  - we can check `R[rs] < R[rt]` by passing a new `ALUOp` value of `11` to the ALU controller
    - also program the ALU controller to tell the ALU to perform an `slt` when passed `ALUOp = 11`
  - finally, the lowest bit of the output of the ALU can be ANDed with the `BranchLt` signal and then ORed with the other ANDed branch calculation:
    - alternatively we can add another MUX controlled by `BranchLt`
    - alternatively we can use the negated zero output to check less than condition
  - note that the `Branch` signal is a don't-care since it will be overriden by the OR gate

Ex. Implement the R-type instruction `cmov` with the following pseudocode:

```
if (R[rt] ≠ 0)
    R[rd] = R[rs]
```

- existing mechanisms:
  - can write to `rd` with `RegDst` set high
- need to address:
  - checking `R[rt]` is nonzero

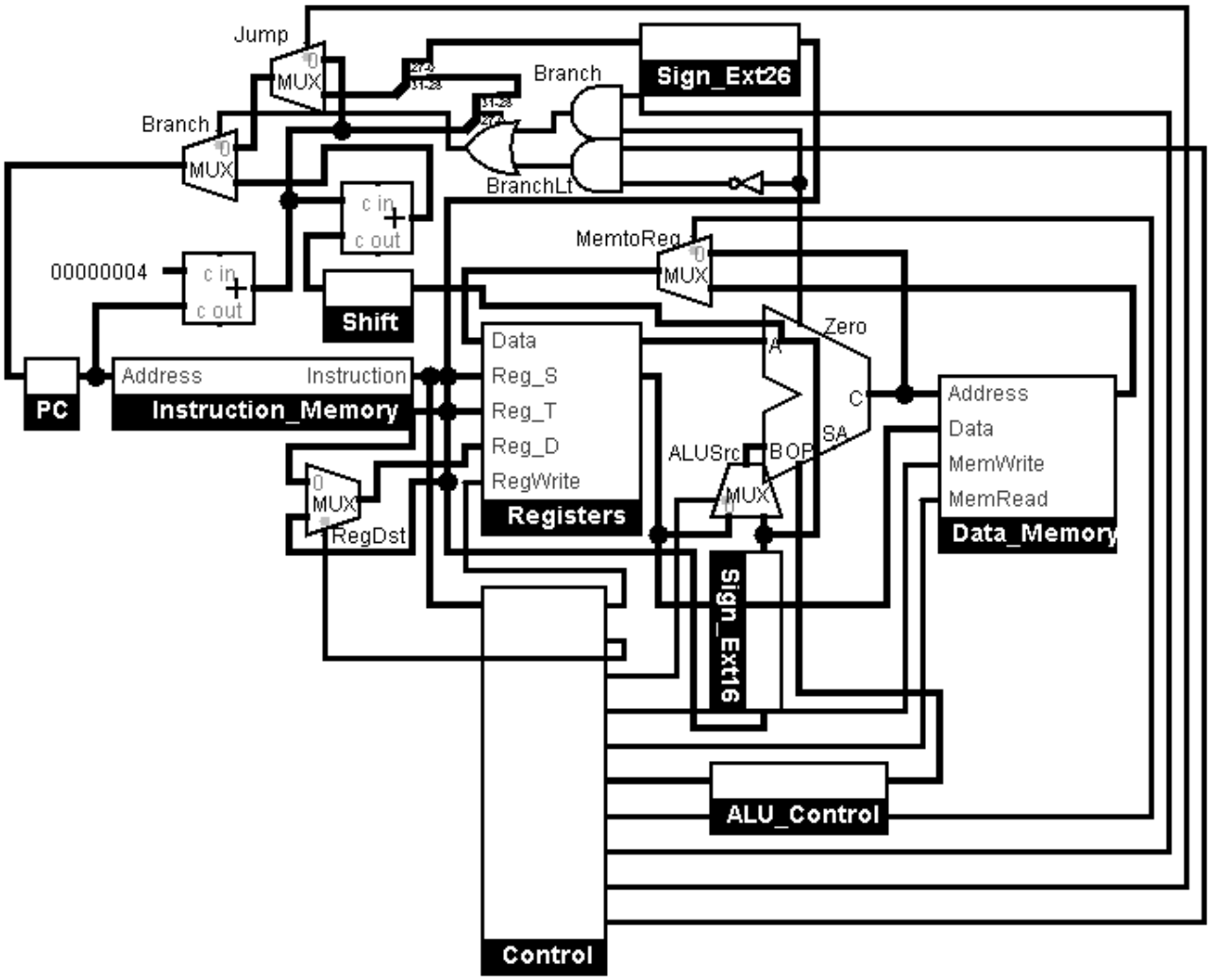

Figure 9: `blt` Full Implementation

  - *conditionally* writing a register value `R[rs]` back to the write data port ie. `RegWrite` needs to be toggleable
- importantly, note that *all* R-type instructions will have the *same* main control signals:
  - since they have the *same* opcode
  - eg. `ALUSrc` will already be 0 in order to send `R[rt]` to the ALU, `RegDst` will already by 1 to write to `rd`
  - instead, we can only take advantage of the `funct` field to change the ALU control behavior
    * the `funct` field should be the only unique difference between R-types

Table 7: Main Control and ALU Control Signals for `cmov`

Control		ALU Control	ALU Function	
`RegDst`	1	`cmov` function field	subtract	`0110`
`ALUSrc`	0			
`ALUOp`	`10`			
`MemtoReg`	0			
`RegWrite`	1			
`MemRead`	0			
`MemWrite`	0			
`Branch`	0			
`Jump`	0			

- based on some specific, unique `funct` field in the `cmov` instruction:
  - augment ALU control to choose an ALU operation of subtraction
    * could alternatively perform an `AND`
  - augment ALU control with a new *outgoing* signal `CmovC`
    * should be high whenever the `funct` field indicates a `cmov`
- using the new `CmovC` control signal as seen in Figure 10:
  - `CmovC` controls a MUX that feeds either the register contents `R[rs]` or the ALU output (no memory because of R-type restriction) back into the write data
  - `CmovC` also controls a MUX that feeds either 0 or `R[rs]` into the top ALU port
    * allows us to test `R[rt] - 0 == 0`
  - finally, we additionally need a mechanism to control `RegWrite` though it is *fixed* for an R-type:
    * negate the `Zero` output after the subtraction to see if we need to perform the move

    * use `CmovC` to MUX the `RegWrite` signal with the negated zero so that the write occurs if `R[rt]` equals 0, instead of depending solely on `RegWrite`

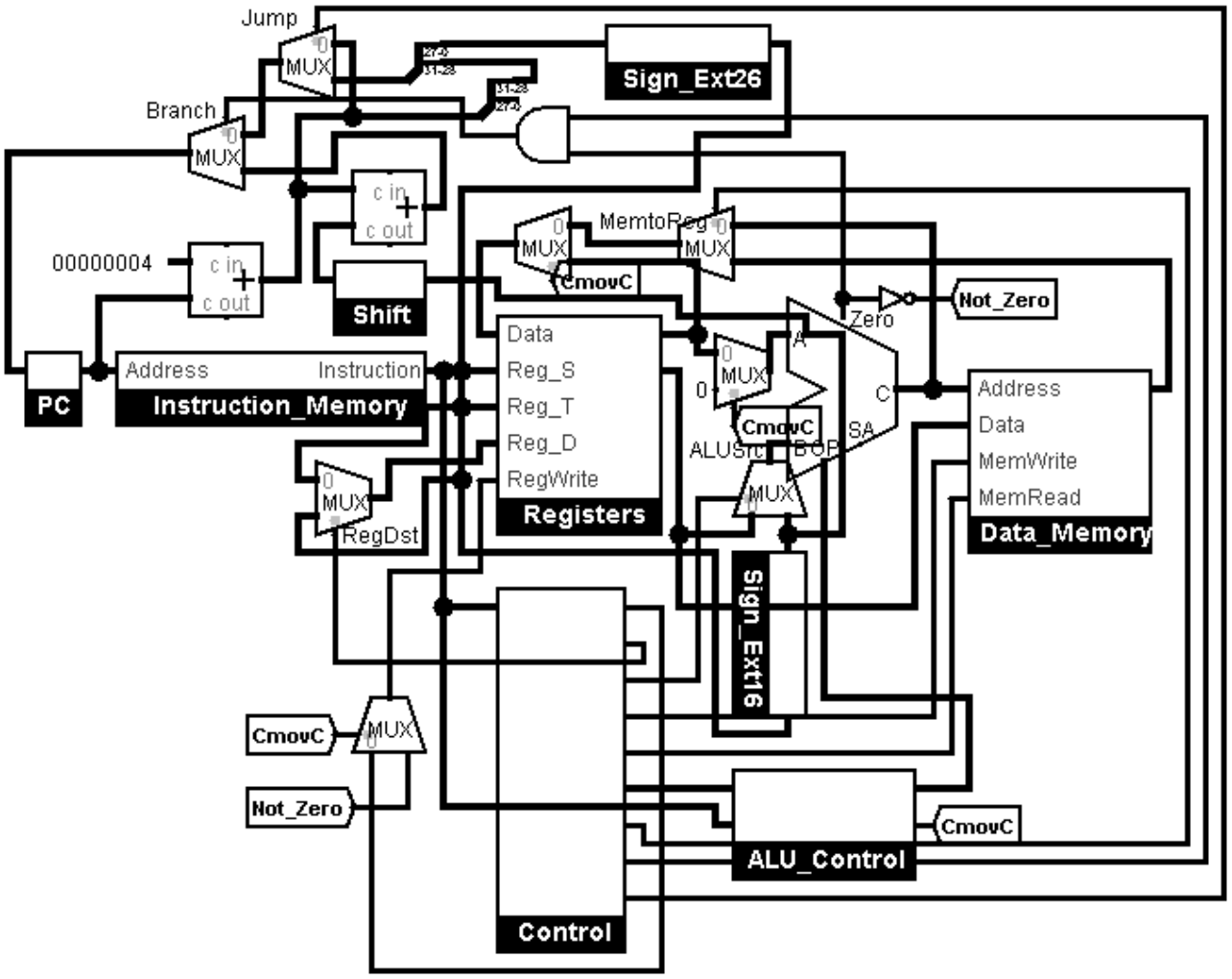

Figure 10: `cmov` Full Implementation

Ex. Implement the I-type instruction `jalm` with the following pseudocode:

```
PC = PC + 4 + M[R[rs]]
R[r31] = PC + 4
```

- note that the immediate field is not used for `jalm`, even though it is an I-type instruction
- need to address:
    - adding a memory read `M[R[rs]]` to `PC + 4`
        * already have an adder that is used in `beq` for `PC + 4 + SES(i)`, can we use this?
    - writing the original PC value to a hardwired register `r31`

Table 8: Main Control Signals for `jalm`

Control		New Control	
`RegDst`	X	`JalmC`	1
`ALUSrc`	X		
`ALUOp`	`XX`		
`MemtoReg`	X		
`RegWrite`	1		
`MemRead`	1		
`MemWrite`	0		
`Branch`	X		
`Jump`	0		

- using the new `JalmC` control signal as seen in Figure 11 to control several different `jalm` -specific MUXes:
  - a MUX that sets the writing register destination to a hardwired `r31` when enabled
  - a MUX that passes through `R[rs]` past the ALU directly into the memory read address port
  - a MUX that sets the write data port of the registers component to the already computed address `PC + 4` when enabled
  - a MUX that sets the lower input of an auxilliary adder to the read data from the memory component:
    - there are multiple ALU adders available, so we are exploiting the adder already wired to perform `PC + 4 + SES(i)` from `beq`
    - feeding in the read data from memory into the adder computes the desired `PC + 4 + M[R[rs]]`
- finally, `JalmC` is also ORed with the previous ANDed branch calculation:
  - thus the `Branch` signal is a don't care since it is overriden by this gate
  - again we are reusing some `beq` branching logic

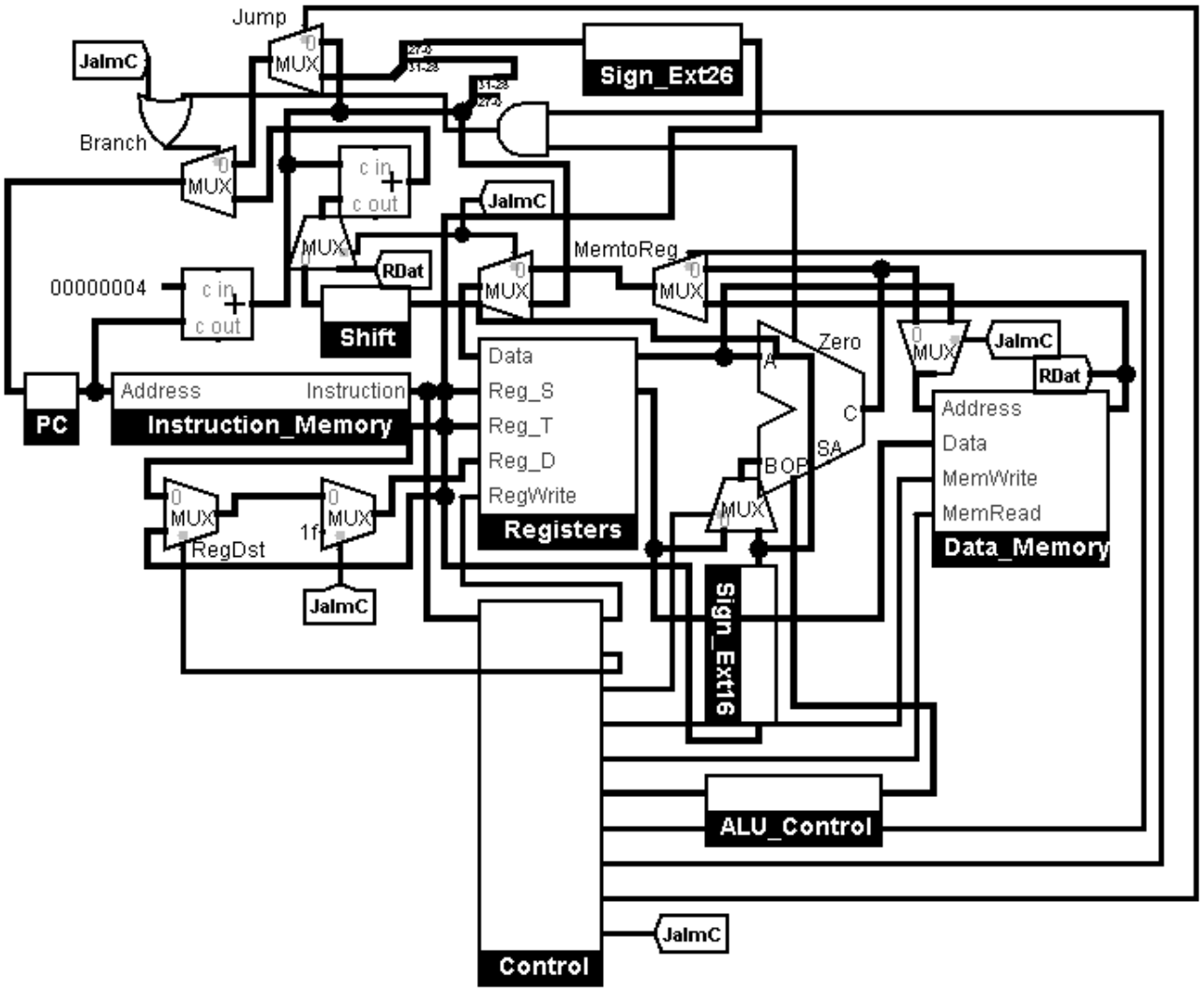

Figure 11: `jalm` Full Implementation

Pipelining

- issues with the single-cycle datapath:
  - longest delay determines clock period
    - the critical path in this case is the load instruction, which must go through every component in the datapath
  - not feasible to vary period for different instructions
  - violates the design principle of making the *common* case fast (common case is usually not memory loads)
  - instead, use pipelining to improve performance
- **pipelining** ie. parallelism is the process of overlapping execution:
  - improves performance by overlapping *discrete* steps
  - eg. for some process that takes four discrete steps in a single load, with each step taking half an hour:
    - the naive approach would just to do loads front to back, which takes 8 hours to complete four loads
    - in the parallel approach, start the second load as soon as the first load goes into the second discrete step, etc.
      - would only take 3.5 hours to complete four loads, which is a 2.3 times speedup
  - in the **steady state**, every stage of the pipeline is full and thus every stage delay produces one unit of work:
    - eg. for the same example, the speedup will approach 4 times since a load can be completed at every step once the pipeline is full
    - over time, the steady state throughput will overcome the initial pipeline *warmup* state ie. pipeline startup before steady state
  - note that while the delay for a particular instruction is its latency, the CPI is a weighted average that gives a measure of throughput:
    - when using pipelining, the total delay of a *particular* instruction may even increase due to added latches
    - however, the *overall* CPI ie. average throughput will approach 1 in the steady state, since an instruction is completed every stage
      - single-cycle also has a CPI of 1, but in pipelining the cycle time can also drop dramatically by a factor of the number of stages
- MIPS ISA is designed for pipelining:
  - all instructions are the 32-bits
    - can fetch and decode in one cycle
  - few and regular instruction formats
    - can decode and read registers in one step
  - load / store addressing becomes simpler
  - alignment of memory operands allows memory accesses to only take

one cycle

- in the MIPS pipeline, there are five stages:
  1. IF is the instruction fetch from memory, involves the instruction memory
  2. ID is the instruction decode and register read, involves reading the register file
  3. EX is the operation execution or address calculation, involves the ALU
  4. MEM is the actual memory operand access, involves main memory ie. data memory
  5. WB is the write back of the result to register, involves writing the register file
  - want to achieve a steady state where a different instruction is being executed in each stage of the pipeline
    - vertical slices indicate which and how many instructions are being pipelined
  - although we are reading and writing to the register file in the same cycle when the pipeline is in steady state:
    - a *transparent* latch is used that separates the reading and writing to registers within the same cycle
    - ie. writing occurs in the first half of the cycle and reading occurs in the second half
    - note that there are no structural hazards to the conflict (different ports are used), just timing conflicts
- comparing pipeline performance with the single-cycle datapath:
  - assume time for stages is 100 ps for register reads or writes and 200 ps for other stages
  - ex. For three loads in a row:
    - in single-cycle, need a $T_C = 800$ ps since every load requires 800 ps
    - when pipelined, need a $T_C = 200$ ps since the maximal latency of an individual stage is 200 ps
      - instructions need to move in lock-step fashion
  - if all stages are balanced ie. take the same time, $T_{piped} = \frac{T_{non-piped}}{stages}$
    - if not balanced, speedup is less, need to take the maximal delay
  - when there are mixed instructions in the pipeline, complications can occur:
    - eg. a load word accesses a data memory before writing back to registers, while an addition goes straight to registers after the ALU
      - this causes a conflict in the `lw, add` instruction sequence, since the `lw` takes one more stage to get to writeback
    - need to *regularize* the path that instructions take through the pipeline

        - adds cannot skip data memory in our current implementation
- pipelining principles:
    1. all instructions that share a pipeline must have the same stages in the same order
        - eg. `add` does nothing during the MEM stage and `sw` does nothing during WB stage
    2. all intermediate values must be latched ie. *saved* each cycles:
        - ensures instructions don't interfere with each others' signals
        - need registers ie. flip-flops to act as banks between stages as seen in Figure 12
            - registers hold information produced in previous cycle
        - latches acts as a separator between the pipelined stages so that the inputs and outputs of each stage can stabilize
    3. there can be no functional block reuse
- some data hazards to consider when pipelining:
    - control hazards:
        1. after a branch, the next instruction that should be run is *dynamic*
            - this is where branch prediction comes into play
        2. some instructions have *dependence* on a previous instruction:
            - ie. checking for a value after a memory load before changing control
            - eg. in `lw, add, sub` , the addition and subtraction wait on the `lw` result
                - since the EX stage occurs before WB of the memory read
    - structural hazards
        - cannot skip over stages ie. paths have to be ordered and regularized
    - these hazards prevent the CPI from ideally approaching 1

Example Instruction Flows

- instruction flow through the pipeline for `lw` :
    1. during IF, instruction corresponding to the PC will be stored into the IF/ID latch, along with the `PC + 4` calculation
        - note that we will have to deal with the hazard of instructions changing the PC eg. `beq`
    2. during ID:
        - the IF/ID latches are read to get register numbers
        - the two corresponding register values are then saved into the ID/EX latch
            - note that the contents of only one register is used in `lw` , but the reads still occur

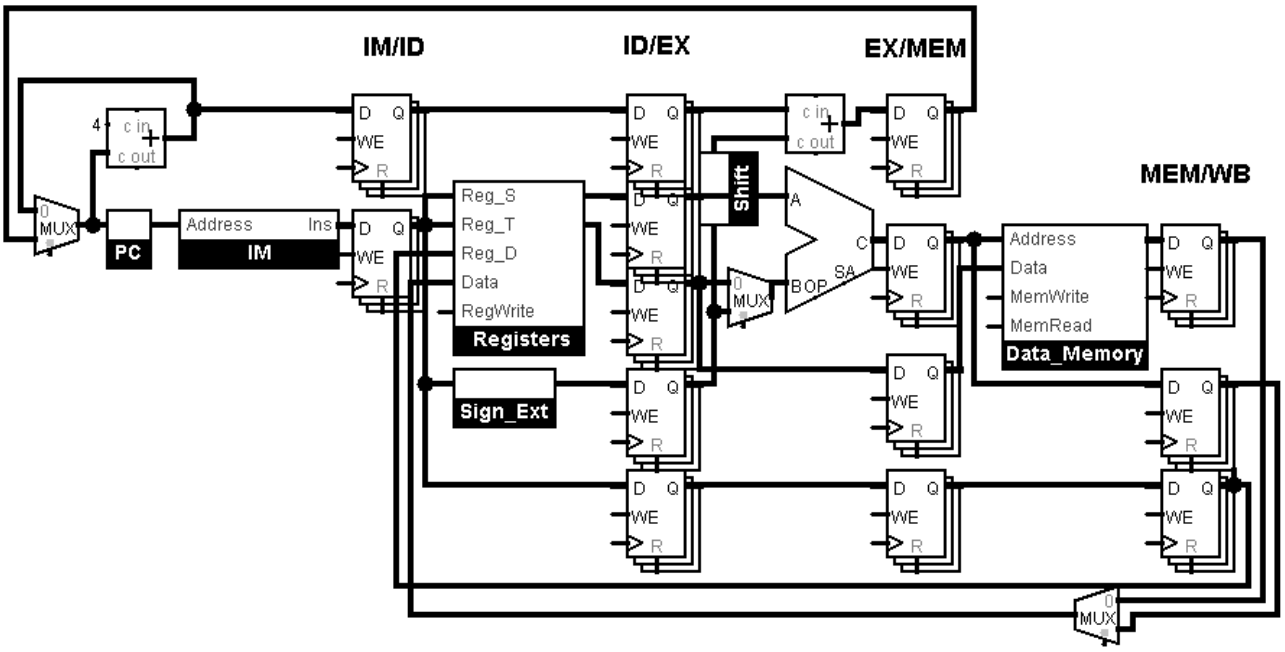

Figure 12: Register Latches in the Pipeline

  - the 16-bit immediate will be sign-extended are stored into the ID/EX latch
  - `PC + 4` is also be propagated into the ID/EX latch
3. during EX:
  - sign-extended immediate from the ID/EX latch is added to `PC + 4` and stored into the EX/MEM latch
    * this value is not used in `lw` , but the calculations still occur
  - sign-extended immediate from the ID/EX latch is MUXed into the ALU with the top register value
    * ALU result is saved into EX/MEM latch
  - the lower register value from the ID/EX latch is also propagated into the EX/MEM latch
    * this value is not used by `lw` , but the latch still occurs
4. during MEM:
  - the result of the ALU from the EX/MEM latch is used to address into data memory
    * the value read in from data memory is stored into the MEM/WB latch
  - the result of the ALU itself is also stored into the MEM/WB latch
5. during WB:
  - the value from memory in the MEM/WB latch is MUXed into the write data port of the registers
    * but this is now the write register destination of a different instruction!
  - to fix this, the write register destination must be propagated *down*

the pipeline to WB *alongside* the memory address calculation and access
    * as seen in Figure 12
  - placing latches in specific places may affect the delay in each pipeline stage
    * eg. which stage ALU controller or MUXes lie between which latches
- instruction flow through the pipeline for `sw` :
  - same steps as `lw` for IF through EX
  4. during MEM:
     - the result of the ALU from the EX/MEM latch is still used to address into data memory
       * but the `MemWrite` signal will be high, and the value of the lower register from the EX/MEM latch will be passed into the write data port of the data memory component
  5. during WB, `RegWrite` is low for `sw` , so nothing occurs in this pipeline stage

Control

- the control inputs for the pipelined datapath are identical to the ones in the single-cycle datapath:
  - however, we now have *multiple* instructions flowing through the pipeline at the same time
  - could have up to 5 set of control oepration signals at the same time, one for each pipeline stage
- to allow for this, the control signals are now *also* latched between stages alongside the instruction data:
  - the instruction itself is still read out of the IF/ID latch into an identical main control unit
  - controls are divided ie. clustered into the stages control signals refer to:
    * EX, MEM, WB specific control signals are passed down the pipeline until they reach their specific stage
    * eg. `AluOp` is only stored in the ID/EX latch since it is immediately consumed by the ALU in the EX stage
  - each stage can now hold insulated control signals for different instructions