CS151: Computer Architecture

Professor Reinman

Thilan Tran

Winter 2021

Contents

CS151: Computer Architecture

- eight central 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. memory hierarchy
    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 eg. may be a **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 managing heat
  - manufacturing silicon chips:
    - slice a silicon ingot into wafers
    - process the wafers with patterns through a **lithography** process
    - 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 ie. machine-specific instructions, eg. MIPS vs. x86
  3. hardware representation:
     - encoded instructions and data in binary
     - what the 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, and apps above it
  - defines different instructions like `add` or `movsql` , calling protocols, register conventions, 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 considering loops and recursion, rather than the *static* count of a program
  - CPI are the clock cycles per instruction:
    * clocks need to be 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 the 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 instruction type CPI_i
        - note that the CPI is a count dependent on the *particular* program and the *particular* hardware it is running 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
            - direct 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, CPU time only records the time spent processing a single job
        - discounts the time of other jobs' shares, I/O, etc.
- affect of stack layers on performance:
    1. algorithm affects IC, possibly CPI
        - could change the number or type of instructions used eg. multiplication vs. addition
            - similar logic for programming language and compiler choice
    2. programming language affects IC, CPI
    3. compiler affects IC, CPI
    4. ISA affects IC, CPI, T_C
        - ISA may limit the expressiveness of higher-level languages as well as the expressiveness 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:

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

        - thus, we should 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:

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

  - recently hitting a *"wall"* of power, ie. voltage has not been scaling down enough
    - 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, monolithic cores, instead design towards multi-core designs in a **many-core revolution**
  - in a multi-core design, many cores can work together or alternatively multitask

Multiprocessors

- put more than one processor per chip
- *pros*:
  - can dramatically improve performance, beyond the power wall limitation
- *cons*:
  - requires explicitly parallel programming by programmers
    - most programs are not *"embarrassingly"* 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 repertoire 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 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
    - a MAM would take 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
        - the 3 isntructions would altogether take $3 \times 3 = 9$ 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 later 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 motivated by the limited instruction window available in out-of-order execution optimization
- performance tradeoff of RISC vs. CISC:
    - RISC typically has a higher IC
        - less expressive instructions available
    - RISC can have a lower cycle time or lower CPI:
        - TC 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
    - typically, 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 TC and average CPI, due to increased decoding time in the hardware
        - 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 added to an address in a register to form an effective address
  - note that branch and jump instructions treat the addresses passed as immediates at a word offset (since neither instruction type accepts a full 32-bit immediate)
- 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 a `funct` field eg. `add, sub, or`
  - **I-type** instructions take 2 registers and an immediate eg. `addi, lw, sw`
    - different opcodes for each I-type
  - **J-type** instructions just take an immediate eg. `j, jal` :
    - different opcodes for each J-type
- 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 in x86-like addressing, IC can go down, but the CPI and TC could go up
  - in addition, in RISC compared to CISC, there is *more* pressure on the register file since the only way to load memory is to use `lw` to load it into a register first:
    - can lead to thrashing and smaller working sets
    - RISC thus typically has larger register files
- instruction format field tradeoffs:
  - the 5-bit fields for register numbers is dependent on the total registers
    - eg. if we increased the register file size, may instead require 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:
    - in a `lw` , 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
    - thus if we don't use a certain feature while performing many spills, it may be beneficial to adjust field bit widths
        - eg. if not many shifts are performed while 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 ie. potentially increased TC
        - *more* hardware in order to implement the register file
- pipelining and parallelism 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 the compiler has access to more layered code to optimize
    - parallelism in hardware
        - designing hardware components that can run at the same the time
    - parallelism 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 the top 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 are 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 the location a desired 32-bit value is in
      - double indirection if 32 bits hold an address

Memory Operands

- main memory is primarily 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 only 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 instructions take 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 values 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 with two sources and 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 in a register
    - can now 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 sequence
  - `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 the byte-level addressing of `lw, sw`)
      - using a sign extension with a 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 to test registers in a 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 a `lw` or `lui, ori` to build the full address up before the `jr`
        - but may unnecessarily pollute registers, so `j` is provided as another way to jump
- summary of addressing modes:
    1. **immediate addressing** uses an immediate in the instruction
    2. **register addressing** uses the address stored in a register
    3. **base addressing** adds a register address base with an immediate address offset
    4. **PC-relative addressing** offsets the current PC with an immediate address
    5. **pseudodirect addressing** 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 in a *coarser* granularity as a basic unit for optimizations

Translating C `if-else` :

```c
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` :

```c
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 elements are each 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:
    - want to instead optimize for the common case
    - 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
    lw   s0, 0(sp)
    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 ie. machine code
- MIPS instructions:
  - encoded as 32-bit instructions
  - specific format for opcode and register numbers (5-bit)
    - supporting different formats complicates decoding, but MIPS keeps formats as similar as possible
  - very regular
- MIPS R-type instruction fields:
  - 6-bit `op` for opcode:
    - specifies the format of the instruction
    - always `0` in R-type
  - 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 expressiveness
      - possible since R-types don't use a large immediate field
  - used for all arithmetic and logical 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)
```

- segments in 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 and frame pointers 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 sign 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
    - directly uses MIPS `addu, addui, subu`
  - while other languages like Ada, Fortran raise exceptions
    - uses 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` is interpreted from both the instruction `opcode` and `funct` field by a secondary ALU controller
  - four outputs `CarryOut, Zero, Result, Overflow`
    - `Zero` is operation independent and is high whenever the ALU result is zero, typically 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 based on the specified `op` operation

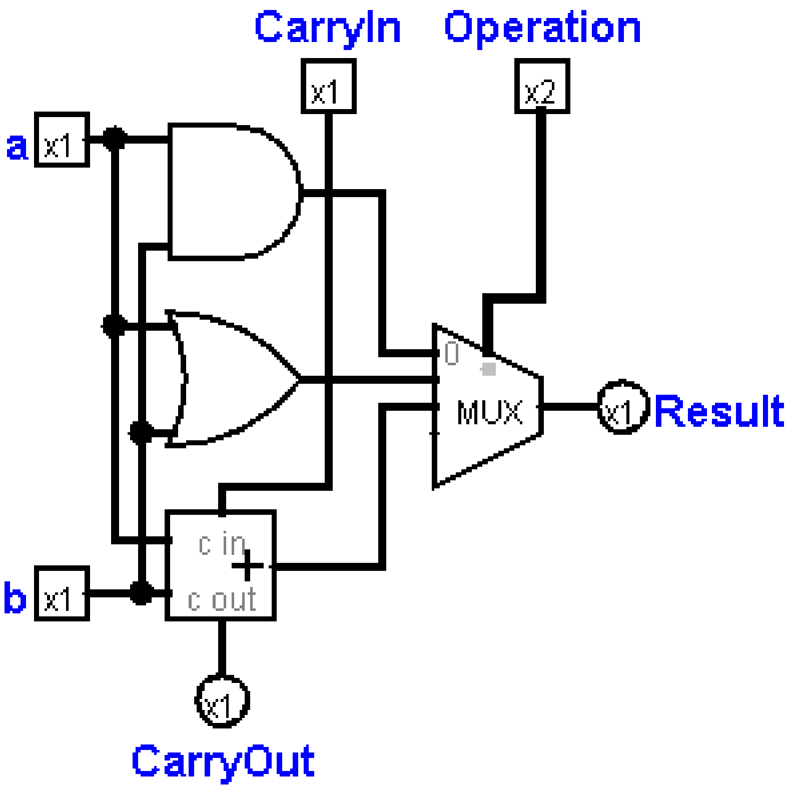

Figure 1: 1-bit ALU

- 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, use a 3 input XOR (outputs 1 when odd number of input 1s).
      // Can be implemented with 2 XORs as: Sum = a XOR b XOR CarryIn
```

- 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:

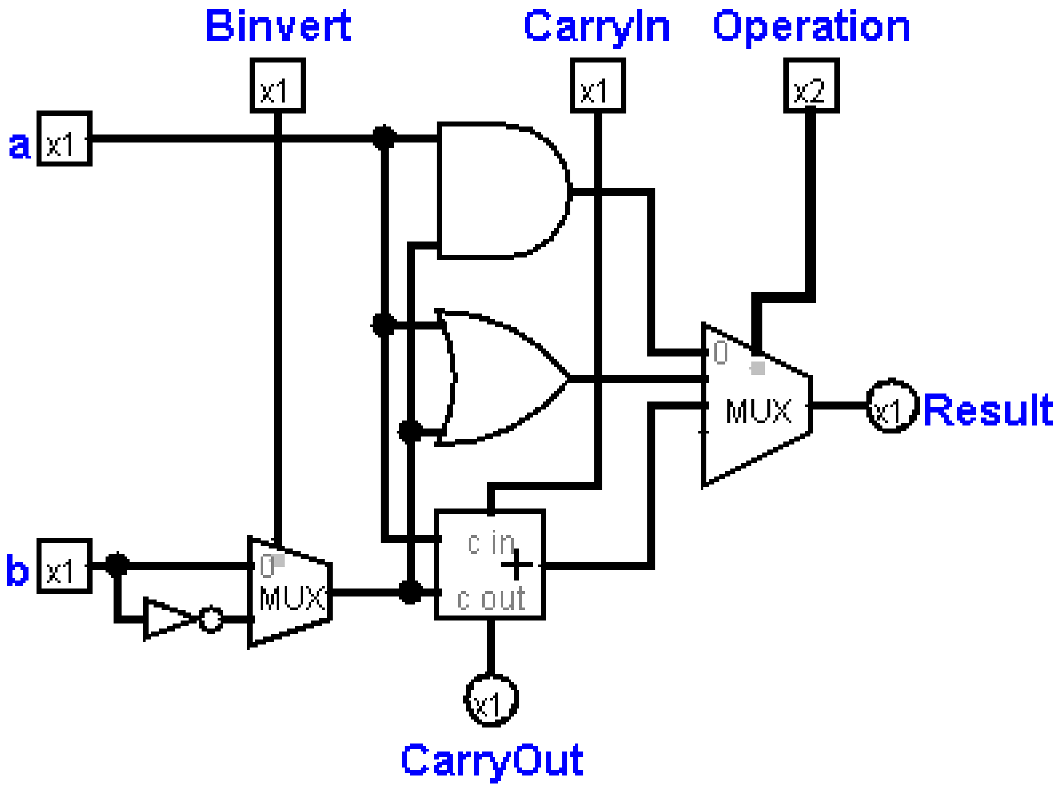

Figure 2: 1-bit ALU with Subtraction

  - as in Figure 2, a `Binvert` signal can select the flipped bits of input `B`
  - but still need to add 1 to properly negate the input, so ALU0 would have a `CarryIn` set to 1 during subtraction
  - adding an `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

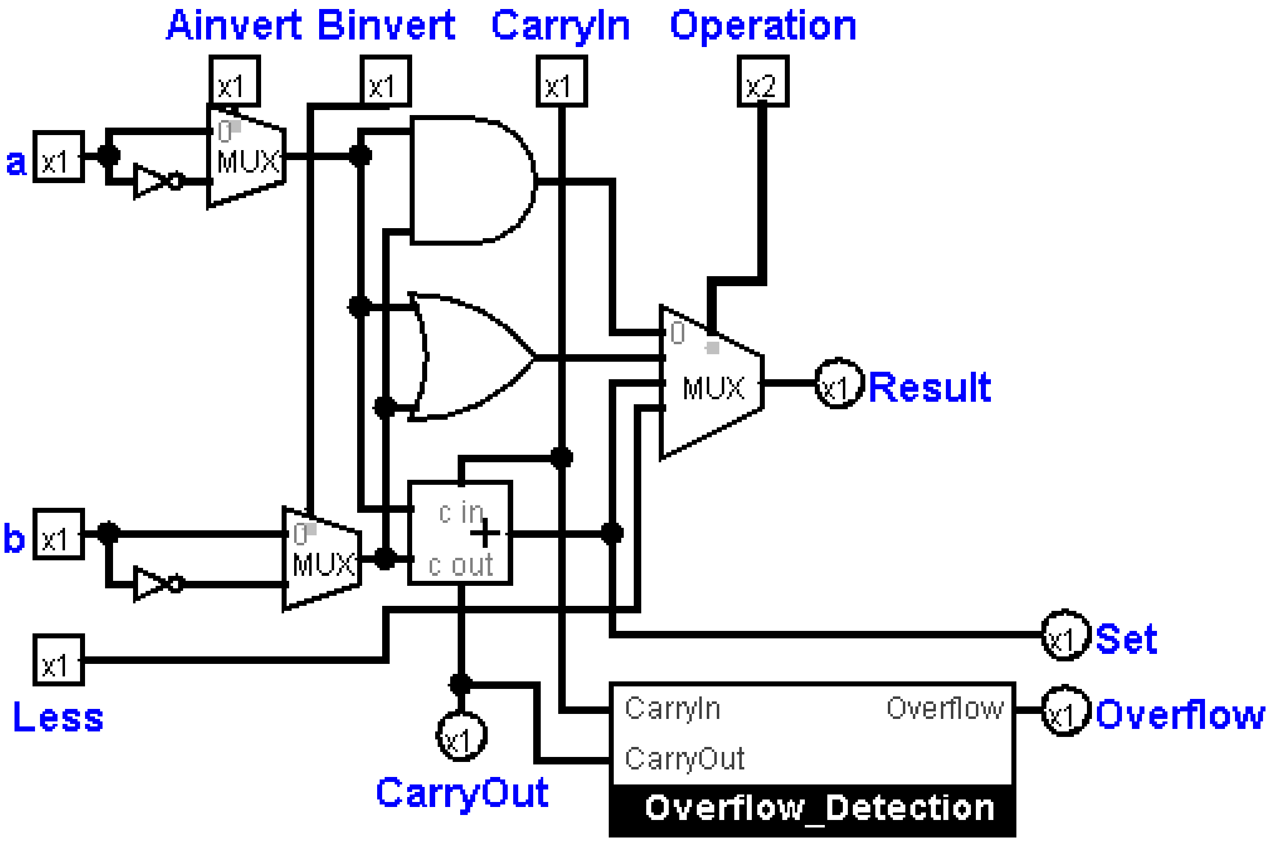

Figure 3: 1-bit ALU handling slt (`Set, Overflow` circuitry only appear in most significant ALU)

  - `set` in MSB ALU is then connected *back* to the `Less` of the LSB 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`

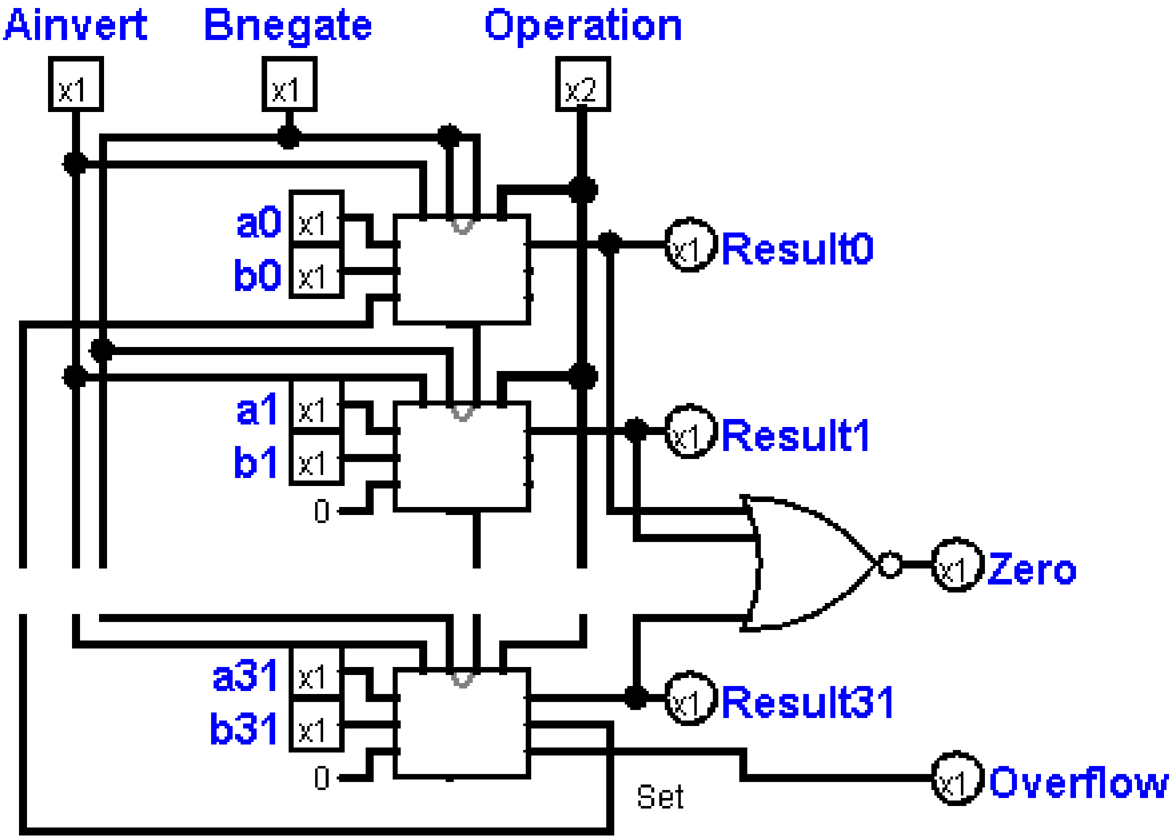

Figure 4: Final 32-bit ALU

- thus the full 32-bit ALU can be implemented as in Figure 4:
  - `Bnegate` hooks up `Binvert` to automatically set the `CarryIn` of ALU0 to high in order to perform negation
    - can still perform a `NOR` , since setting the carry-in is a no-op when not using the adder's output
  - 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:
     - to actually perform the sum at bit 1, requires two more `XOR` calculations, one of which uses `C1` , so calculating the sum requires an additional $2T$
     - the other `A XOR B` calculation can be performed earlier, since it only has delay $2T$
     - thus sum at bit 1 has delay $7T$
  2. `C2` has delay $10T$, with the same structure as `C1`
  3. `C3` has delay $15T$
  4. etc.
  - assuming gate delays are proportional to their fan-in

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 propagated
  - thus we can calculating the carry-out without considering the immediate 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 only on `A, B`

    and the earliest carry-in
  - *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 seen below can become quickly expensive area-wise

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.
```

- timing analysis of a flat CLA adder:
  - 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, and the sum at bit 1 has delay $8t$
  2. `C2` has delay $8T$, since it has a triple-OR with triple-AND along the slowest path, 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, this sum requires $15T + 2T = 17T$
  4. etc.
  - assuming gate delays are proportional to their fan-in
    - in reality, a larger fan-in incurs a nonlinear increase of delay
- another way to chain together lookahead adders is 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 *larger* granularity 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 *completely* independent of carry-ins
  - then we can calculate a higher level of carry-ins that feed back into the units of four 1-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` , etc.
    - note that at this higher level, the delays of `G_a, G_b, ...` are all the

        same and the delays of `P_a, P_b, ...` are all the same, similarly to the `G, P` delays at lower 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 Dependencies for a HCLA with 4-bit Units

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

- can also use a **partial CLA**:
    - multiple loohakead adders can be connected directly together, carry-out to carry-in:
        - eg. four 8-bit carry lookahead adders can form a 32-bit partial CLA
    - ie. a rippling of lookahead adders
    - *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 used for redundant calculations
    2. then, select one of the results depending on the actual carry-in
        - the delay of the multipexer is relatively small
    - more useful when performed in conjunction with HCLA or partial CLA, and/or at the end of a sum 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 an addition chain

Multiplication

- starting with the long multiplication approach for binary:
  - analagous to long multiplication in base-10 ie. follows a 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

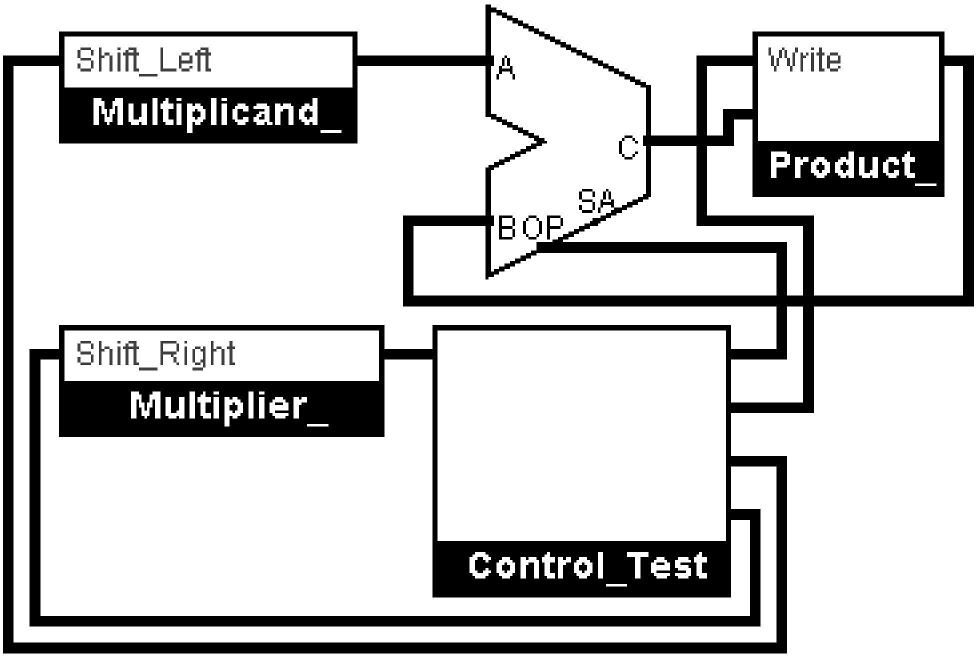

Figure 5: Multiplication Hardware with a 64-bit ALU

- datapath for Figure 5, using a 64-bit ALU:
  - 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

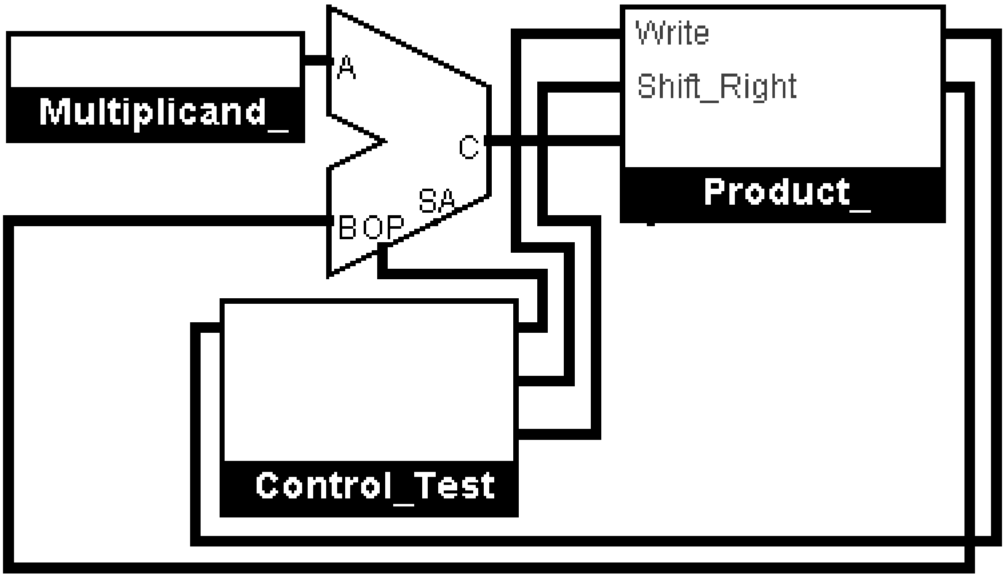

Figure 6: Multiplication Hardware with a 32-bit ALU

- 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 subtract
    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 middle of 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. operates on data
    - note that there is always some output from the element, but it may not stabilize until a certain point in the clock cycle
  - 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 flows from state element, through combination element, to another state element
- instruction execution steps:
  1. fetch instruction from where program counter points in instruction memory
  2. decode the instruction and read values from the register file based on the register operands
  3. depending on the instruction class:
     - use ALU to calculate arithmetic result, memory addresses for load / stores, branch target addresses, etc.
     - access data memory for load /store
     - set PC to target address or increment 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 instruc-

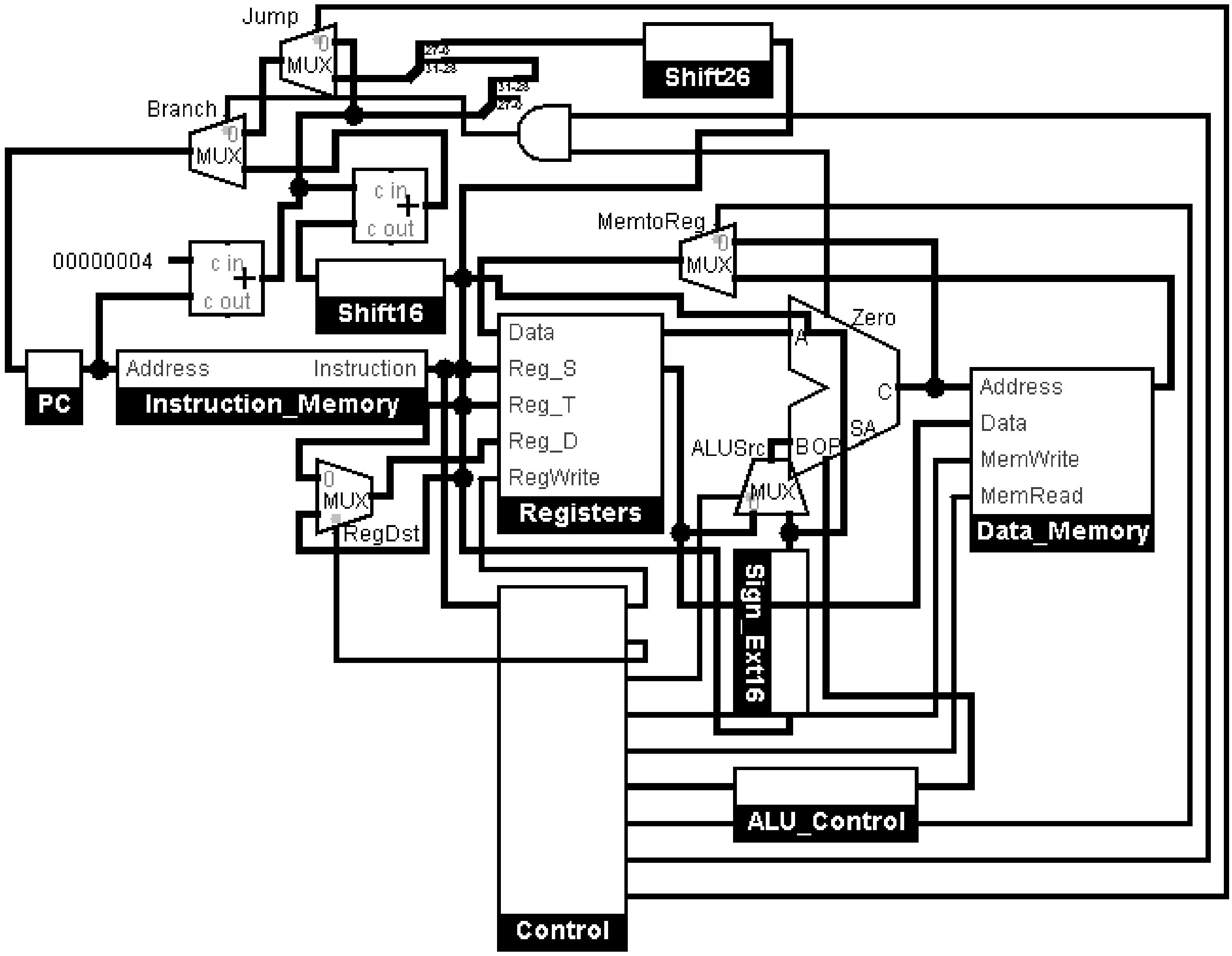

Figure 7: Processor Control and Datapath

        tion memory as an immediate
    5. data memory with an address and data input
        - may perform either a read or write
    - 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, 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 all at once
- the register component:
  - takes in 3 5-bit register numbers
  - two register numbers refer to registers whose values are produced on the two outputs
  - the last register number refers to a writable register:
    - depending on the `RegWrite` control input, analogous 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 value from data memory 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 a register value 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 to specifies specify to write a value on the write data port into memory
    - a `MemRead` control to specify whether to pull a value out of memory onto the read data port
        - note that we did not use a control signal to read registers since they will be read in almost every operation
            - whereas main memory is not always read in every instruction
- handling branch instructions eg. `beq` :
    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 `Branch` 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 executes a single instruction in one clock cycle:
    - each datapath element can only do one function at a time
        - thus 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:
        - data flows no matter what through the datapath, so we need to use control to indicate which parts are useful
        - eg. immediates vs. registers into ALU
    - in this single-cycle implementation CPI is constant, but the cycle time will be quite high since especially long instructions eg. `lw, sw` must all complete in a single cycle

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)
        - each specified by a different ALU control code
    - for load and stores, need to function as an add for effective address com-

putation
    * 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` field of the instruction
  - Table 3 indicates how the ALU control codes may be extracted from an opcode
    * specifically, controlled by a 2-bit `ALUOp` embedded within the instruction opcode

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 into a register:
       * 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
   - if R-type, ALU controller needs to check the `funct` field as well
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 write register specified by `RegDst` 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 want to use the second register value 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 the memory output to the write register
  5. `ALUOp` should be hardwired to `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-type	`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 with `Branch` to determine 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 slower 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 output of ALU to register `rt` with `RegDst` and `MemtoReg` both 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 address at `R[rs]` with zero 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 for every new instruction implementation that the new opcode for the instruction 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:
  - `LawC` control is the lowest output from the main control unit
  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

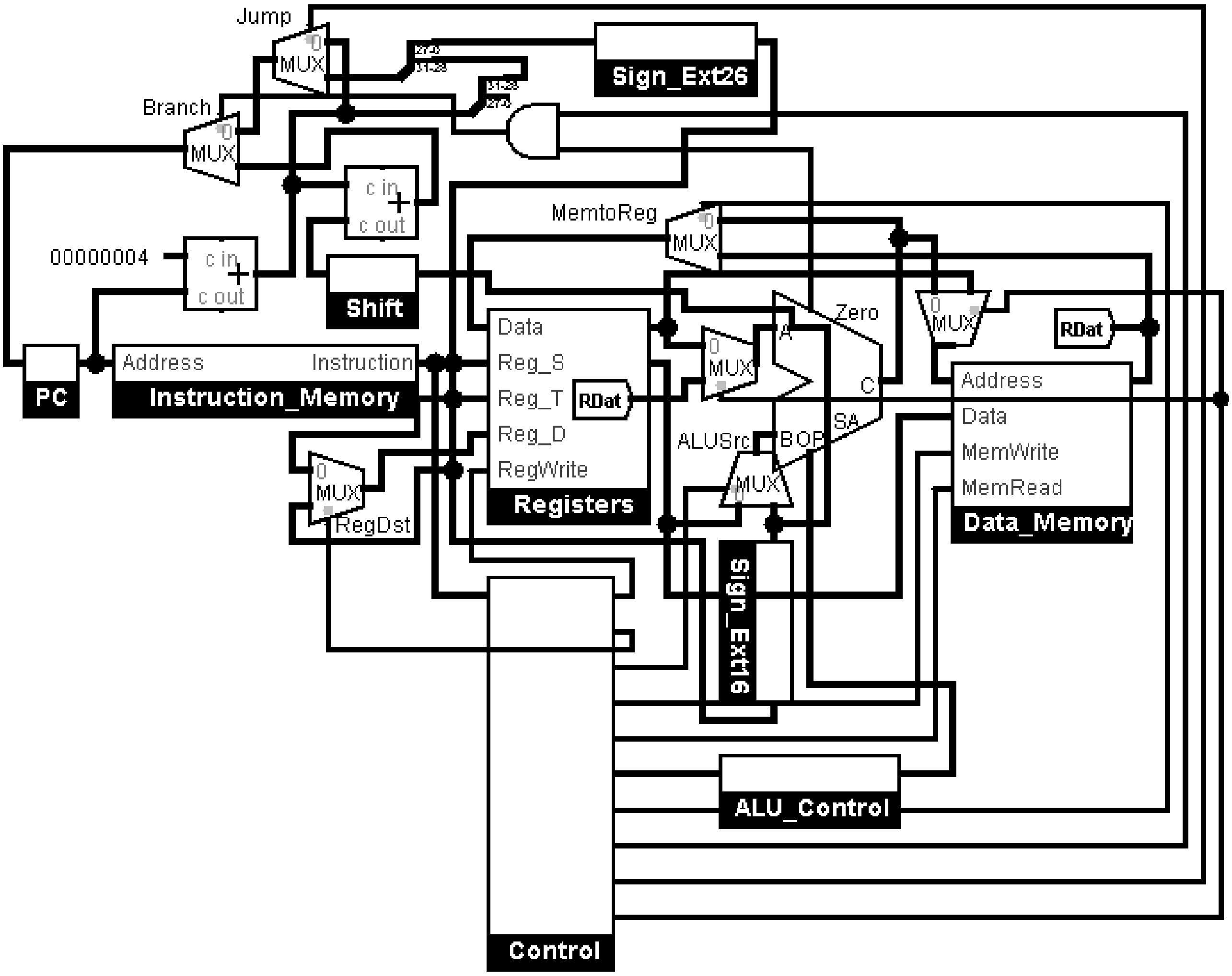

Figure 8: `law` Full Implementation

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`
  - 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
  - *conditionally* writing a register value `R[rs]` back to the write data port

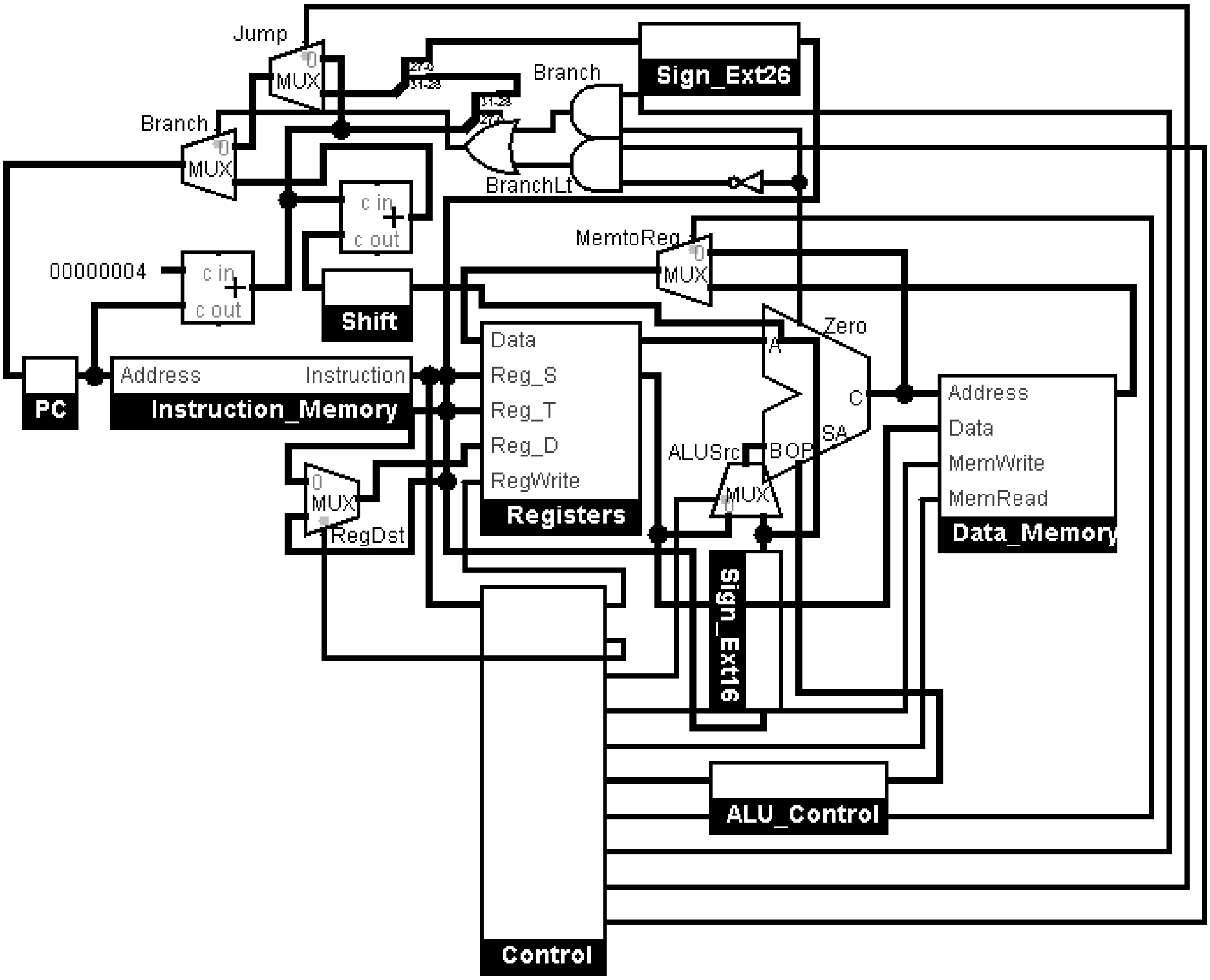

Figure 9: `blt` Full Implementation

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 be 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 only if `R[rt]` equals 0, instead of depending solely on `RegWrite`

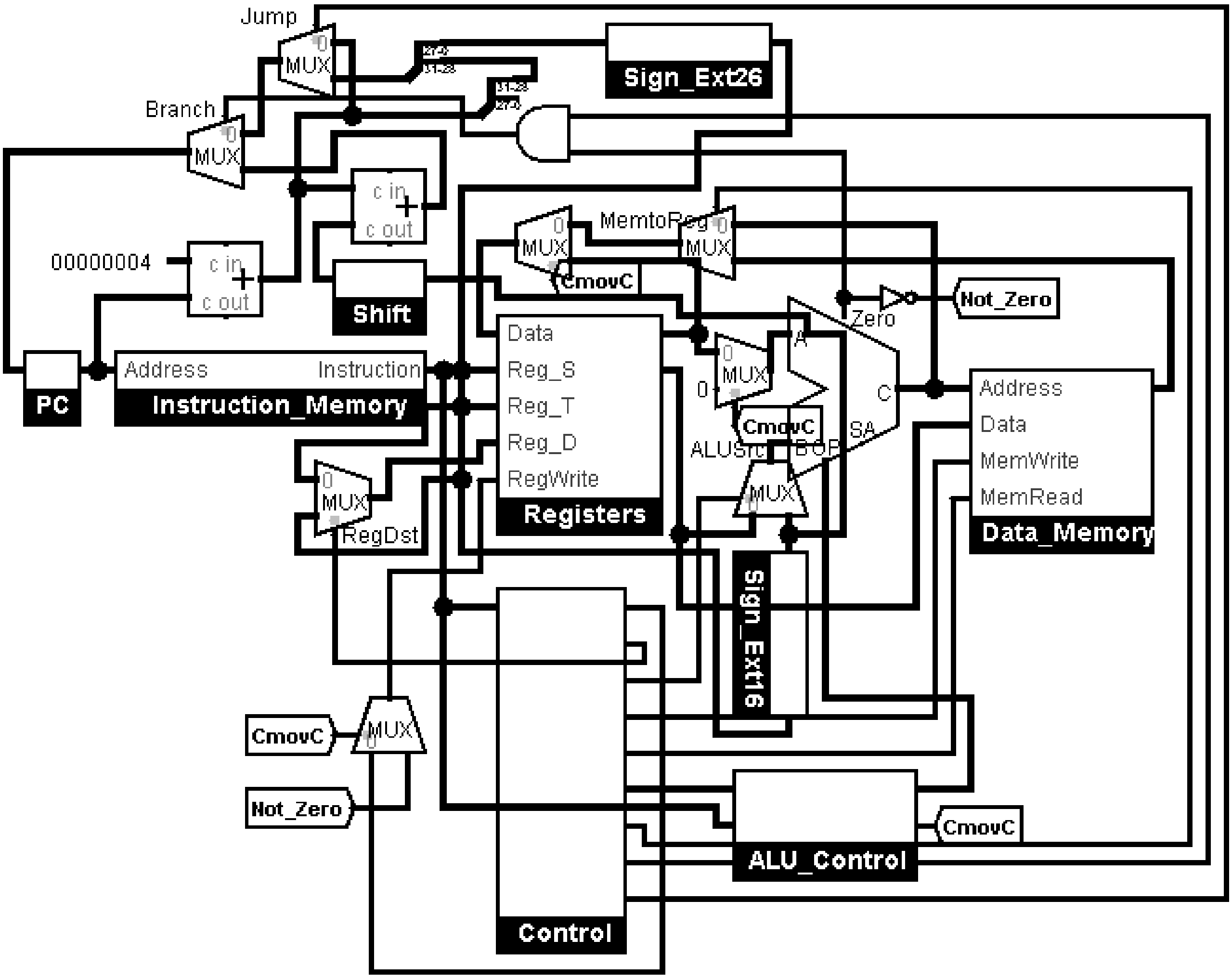

Figure 10: `cmov` Full Implementation

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 MUXs:
  - 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 auxiliary 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 overridden by this gate
  - again we are reusing some `beq` branching logic

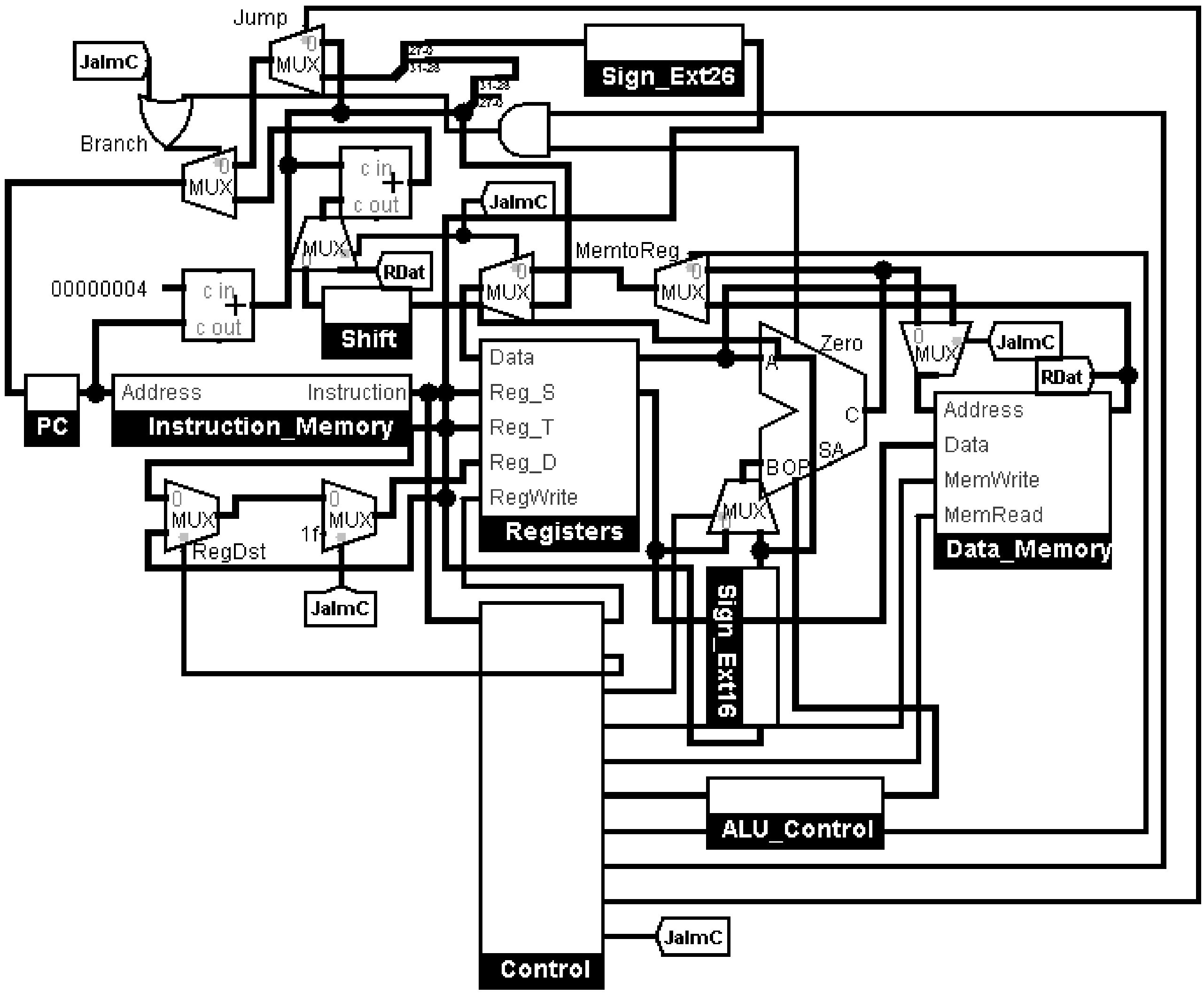

Figure 11: `jalm` Full Implementation

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** is a type of parallelism where we overlap 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 *dramatically* drop by a factor of the number of stages
- MIPS ISA is designed for pipelining:
  - all instructions are 32-bits
    - can fetch and decode quickly in a single 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 just 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 cycle:
     - 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

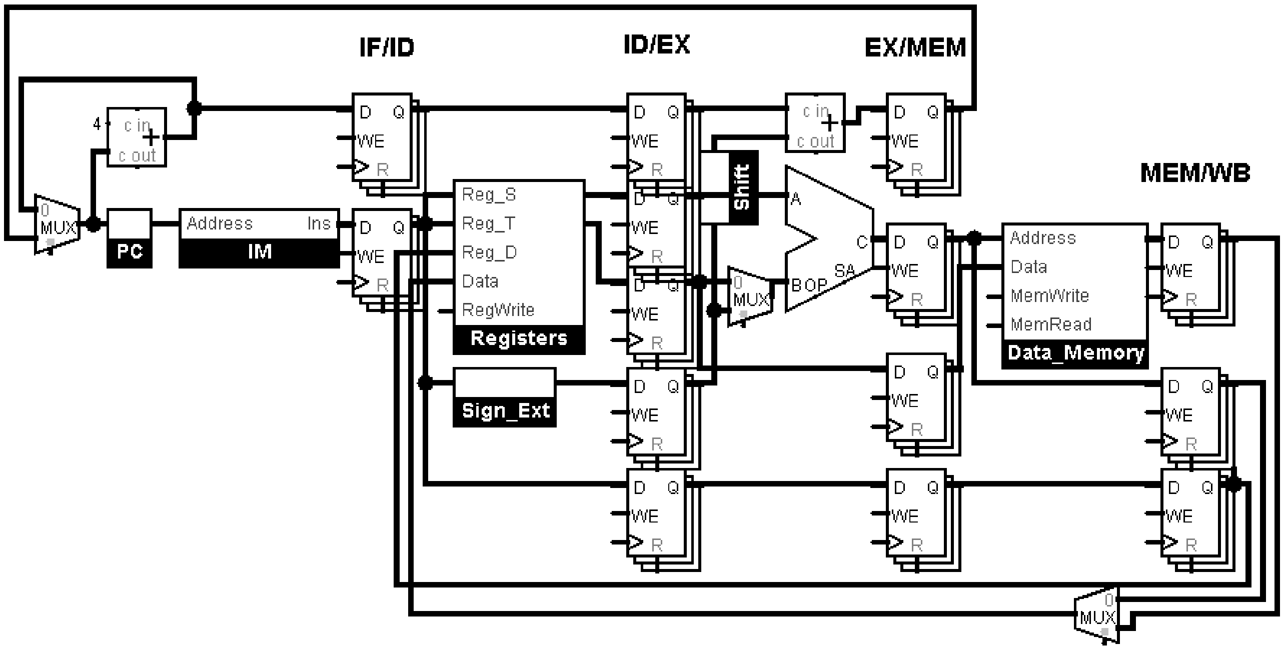

Figure 12: Register Latches in the Pipeline

  - 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 MUXs 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

Hazards

- **hazards** are situations in the pipeline where performance drops and CPI goes

above 1:
  - ie. situations that prevent starting the next instruction in the next cycle
  - in a **data hazard**, need to wait for previous instructions to complete their data read or writes
  - in a **control hazard**, deciding on the next control action depends on a previous instruction
  - in a **structure hazard**, a required resource is busy:
    - in the given pipeline, there is no conflict for instructions since each instruction is in exactly one stage
    - eg. if we only had one single memory for data and instructions, instruction fetch may have to *stall* for a cycle
      - creates a pipeline "bubble" where the pipeline could open up and certain stages cannot move forward

Data Hazards

- when a result is needed in the pipeline before it is available, a data hazard occurs:
  - note that it is possible to write *and* then read from a register in the same cycle by using transparent latches
  - ie. in the best case, we can line up register reads and the writes they are dependent on in the *same* cycle
  - a read after write (RAW) is an example of a *true* **data dependency**:
    - while write after read (WAR) and write after write (WAW) are **antidependencies** that affect scheduling instead of being true data hazards
    - antidependencies affect reordering of instructions, but can usually be addressed through register renaming
- ex. Consider the instructions sequence `add r3 r10 r11` , `lw r8 50(r3)` , and `sub r11, r8, r7` :
  - this is a *straightline* dependency where each instruction produces a value used by the next
    - ie. each instruction depends on the one before it
  - when the `add` is in EX adding two register values together, the `lw` is in ID already reading out from register `r3` :
    - but `r3` has not been written to with the result of the ALU
    - `lw` is reading a *stale* value to form an incorrect address
    - similarly for when the `sub` moves to ID and reads another incorrect value for `r8`
  - this is an example of a data hazard
- to deal with data hazards, can use one of multiple software or hardware tech-

niques

- in software, can insert independent instructions or no-ops:
    - simply insert enough no-ops to *effectively* stall so that the data hazard doesn't occur:
        - eg. shifting a register by 0 is a no-op
        - occupies space in the pipeline, but doesn't affect the functionality of the program
    - *pros*:
        - hardware thus does not need to expect any hazards:
            - no complexity required to add to the hardware
            - much easier to patch software
        - CPI stays close to 1, since an instruction is still executed every cycle
    - *cons*:
        - ruins the portability of the software since pipelines may be extremely machine-dependent
            - the no-op insertion or reordering operation depends on length of pipeline on mahine, etc.
        - increased software complexity
        - instruction count does increase and performance decreases
        - note that the instruction cache may need to grow to keep track of additional instructions, which can increase latency and thus the CPI
- alternatively, we can *reorder* code to avoid use of written results in the next instruction:
    - this optimization is possible, but difficult when reordering *distant* instructions
    - eg. moving loads above stores since they may alias one another, moving instructions around branches, etc.
    - *pros*:
        - same CPI advantages as no-op approach, without any of the IC or latency downsides
    - *cons*:
        - not always possible
        - even more software complexity
    - software vs. hardware handling of hazards is analogous to static vs. dynamic optimization

Example with adding no-ops and reordering:

```
; assuming the delay to write and then read a value is 2 instructions
sub r2, r1, r3
and r4, r2, r5
```

```
or  r8, r2, r6
add r9, r4, r2
slt r1, r6, r7

; only adding no-ops
sub r2, r1, r3
sll, r0, r0, 0
sll, r0, r0, 0
and r4, r2, r5
or  r8, r2, r6
sll, r0, r0, 0
add r9, r4, r2
slt r1, r6, r7

; reordering instructions removes one no-op
sub r2, r1, r3
slt r1, r6, r7
sll, r0, r0, 0
and r4, r2, r5
or  r8, r2, r6
sll, r0, r0, 0
add r9, r4, r2
```

- in hardware, can insert bubbles ie. stalling the pipeline without inserting a no-op or alternatively perform data forwarding:
  - stalls reduce performance, but are required to get correct results
  - stalls are necessary even if we are using data forwarding
- inserting bubbles:
  - similar to injecting no-ops, but in hardware
  - when an instruction realizes a previous instruction still has a register value that it needs, it *waits* in its current stage
  - independently control pipe stages such that *later* stages may continue executing, but the current stage can stall
    - stalling a stage will also stall the previous instructions *behind* it
- to ensure proper pipeline execution through stalls, we must be able to:
  - detect the hazard
  - stall the pipeline:
    - in the current pipeline, need to prevent the IF and ID stages from making progress
      - note that the bubble will only start in IF or ID but the bubbled instruction will still proceed down the pipeline
    - ID needs to read the dependent instruction write and we don't want the IF stage to lose any instructions

- detecting hazards ie. what comparisons tell us when to stall between two instructions A and B?
    1. need to compare the register destination where A will write against the two source operands for B
        - can be either `rs` or `rd` depending on instruction
    2. as well as the `RegWrite` signal for A to see if A even needs to write
    3. may also want to check type of B
        - eg. B is a J-type and doesn't use the second read register, so there is no need to stall
    - at *minimum*, need to check `RegWrite` for A and the register destination of A matching one of the read registers for B

Initial stalling conditions:

```
if ((ID/EX.RegWrite &&
        ((ID/EX.RegisterDest == IF/ID.RegisterRs)) ||
         (ID/EX.RegisterDest == IF/ID.RegisterRt))
    (EX/MEM.RegWrite &&
        ((EX/MEM.RegisterDest == IF/ID.RegisterRs)) ||
         (EX/MEM.RegisterDest == IF/ID.RegisterRt)))
{
    PCWrite = 0
    IF/IDWrite = 0
    ... insert control no-ops ...
}
```

- to stall ie. prevent the IF and ID stages from proceeding:
    1. skip writing to the PC
        - add a new control signal `PCWrite` that can lock down the instruction in the IF stage
    2. skip writing the IF/ID latch
        - add a new control signal `IF/IDWrite` that can lock down the instruction in the ID stage
    3. insert no-ops to inject the bubble by setting all control signals propagating to EX/MEM/WB to zero
        - eg. doesn't read or write from memory, branch, write to registers, etc.
    - *pros*:
        * IC stays the same
    - *cons*:
        * CPI increases due to bubbled parts of the pipeline
        * hardware complexity
- to perform data **forwarding** AKA register bypassing:
    - use the result when it is computed instead of waiting for it to be stored

  in a register:
    - eg. in a `add`, `sub` sequence, just pass the ALU result of the addition back to the input of the ALU in order to be used just in time for the next subtract instruction
    - requires extra connections in the datapath
  - specifically, *forward* values stored in the EX/MEM latch as input into the ALU
    - as well as possibly the MEM/WB latch
  - *pros*:
    - IC still stays the same
    - less increase in CPI by forwarding over stalling
  - *cons*:
    - CPI still increases due to inavoidable bubbles
    - hardware complexity

Example with forwarding:

```
sub r2,  r1, r3
and r12, r2, r5 ; r2 has not been written, but available in EX/MEM latch
or  r13, r8, r2 ; r2 has not been written, but available in MEM/WB latch
add r14, r2, r2 ; r2 has now been written, no more forwarding needed
sw  r15, 100(r2)
```

- detecting the need to forward:
  - need to know the register input sources for later instructions to detect if they should receive a forward
    - also need to compare these input sources with the register destination for older instructions
  - need to pass register numbers along the pipeline
    - eg. `ID/EX.RegisterRs` is the register number for `rs` sitting in the ID/EX pipeline register
  - data hazards occur when:
    1. `EX/MEM.RegisterDest == ID/EX.RegisterRs`
       - ie. check currently executing instruction in the ID/EX latch with instruction currently in the memory stage (previous instruction)
    2. `EX/MEM.RegisterDest == ID/EX.RegisterRt`
    3. `MEM/WB.RegisterDest == ID/EX.RegisterRs`
       - ie. check currently executing instruction in the ID/EX latch with instruction currently in the writeback stage (two instructions ago)
    4. `MEM/WB.RegisterDest == ID/EX.RegisterRt`
  - also need to check if the forwarding instruction will even write to a register

        - `EX/MEM.RegWrite` and `MEM/WB.RegWrite`
    - in addition, only if `rd` for that instruction is not the zero register, since the zero register value never changes
- actually forwarding paths:
    - add additional MUXs in front of each ALU input:
        - each MUX has three register inputs:
            1. original register value that was read and latched in ID/EX
            2. register value from the EX/MEM latch, from instruction in the MEM stage
            3. register value from the MEM/WB latch, from instruction in the WB stage
                - needs to come *after* the `MemtoReg` MUX to either get the result from memory or the ALU
        - in addition to the existing `ALUSrc` MUX
    - add a new control unit ie. the forwarding unit:
        - takes in `rs` and `rt` from the ID/EX latch
            - as well as `EX/MEM.RegisterDest` and `MEM/WB.RegisterDest`
        - controls the two ALU input MUXs with control signals `ForwardA` and `ForwardB`
- the **double data hazard**:
    - in the sequence `add r1, r1, r2` , `add r1, r1, r3` , and `add r1, r1 ,r4` , *both* previous instructions are hazards for the third instruction
        - could receive the forwarded register `r1` from *either* previous instruction as an MEM or WB hazard
    - need to make sure to only use the most *recent* hazard
        - revise the WB hazard condition to only forward if the MEM hazard forward condition is not true

Forwarding conditions:

```
// EX hazard ie. forwarding instruction in MEM stage
if (EX/MEM.RegWrite && (EX/MEM.RegisterDest != 0)
        && (EX/MEM.RegisterDest == ID/EX.RegisterRs))
    ForwardA = 0x10
if (EX/MEM.RegWrite && (EX/MEM.RegisterDest != 0)
        && (EX/MEM.RegisterDest == ID/EX.RegisterRt))
    ForwardB = 0x10

// MEM hazard ie. forwarding instruction in WB stage
if (MEM/WB.RegWrite && (MEM/WB.RegisterDest != 0)
        && (MEM/WB.RegisterDest == ID/EX.RegisterRs))
    ForwardA = 0x01
if (MEM/WB.RegWrite && (MEM/WB.RegisterDest != 0)
        && (MEM/WB.RegisterDest == ID/EX.RegisterRt))
    ForwardB = 0x01

// WB hazard with double data hazard fix
if (MEM/WB.RegWrite && (MEM/WB.RegisterDest != 0)
        && !(EX.MEM.RegWrite && (EX/MEM.RegisterDest != 0)
            && (EX/MEM.RegisterDest = ID/EX.RegisterRs))
        && (MEM/WB.RegisterDest == ID/EX.RegisterRs))
    ForwardA = 0x01
if (MEM/WB.RegWrite && (MEM/WB.RegisterDest != 0)
        && !(EX.MEM.RegWrite && (EX/MEM.RegisterDest != 0)
            && (EX/MEM.RegisterDest = ID/EX.RegisterRt))
        && (MEM/WB.RegisterDest == ID/EX.RegisterRt))
    ForwardB = 0x01
```

- however, we cannot always avoid stalls by forwarding:
  - for load instructions, the value read out of memory is not ready until the end of the MEM stage
    - if we have a load-use instruction sequence, the load will thus not be ready in time for the EX stage of the next instruction
  - if the value is not computed when needed, cannot forward back in time
  - impossible to avoid adding a bubble to stall the pipeline
- load-use hazard detection:
  - need to check when the "using" instruction is decoded in the ID stage:
    - in ID stage, need to pass control signals of all zero in order to perform the no-op and stall
    - check `IF/ID.RegisterRs` and `IF/ID.RegisterRt` against the load destination
      - destination register for a load is always `rt`
  - note that the mechanism to forward the data from an instruction in the WB stage to the EX stage is already in place with the WB hazard condition
  - same process as before for performing a stall on the pipeline:
    - set all control values in ID/EX latch to 0
    - prevent update of PC and IF/ID latch
    - note that the "using" instruction will be decoded again and the following instruction will also be fetched again

Load-use hazard detection:

```
if (ID/EX.MemRead &&
        ((ID/EX.RegisterRt == IF/ID.RegisterRs)) ||
         (ID/EX.RegisterRt == IF/ID.RegisterRt))
```

```
    ... stall ...
```

- add a new control unit ie. the hazard detection unit:
  - takes in `ID/EX.MemRead` and `ID/EX.RegisterRt`
    - as well as `IF/ID.RegisterRs` and `IF/ID.RegisterRt`
  - controls a big MUX over the control signals that can set them all to zeros as a no-op
  - controls the `PCWrite` and `IF/IDWrite` signals to initiate a stall
  - Figure 13 shows the complete pipeline with data hazard handling
    - does not include all control signals and ignores `ALUSrc`

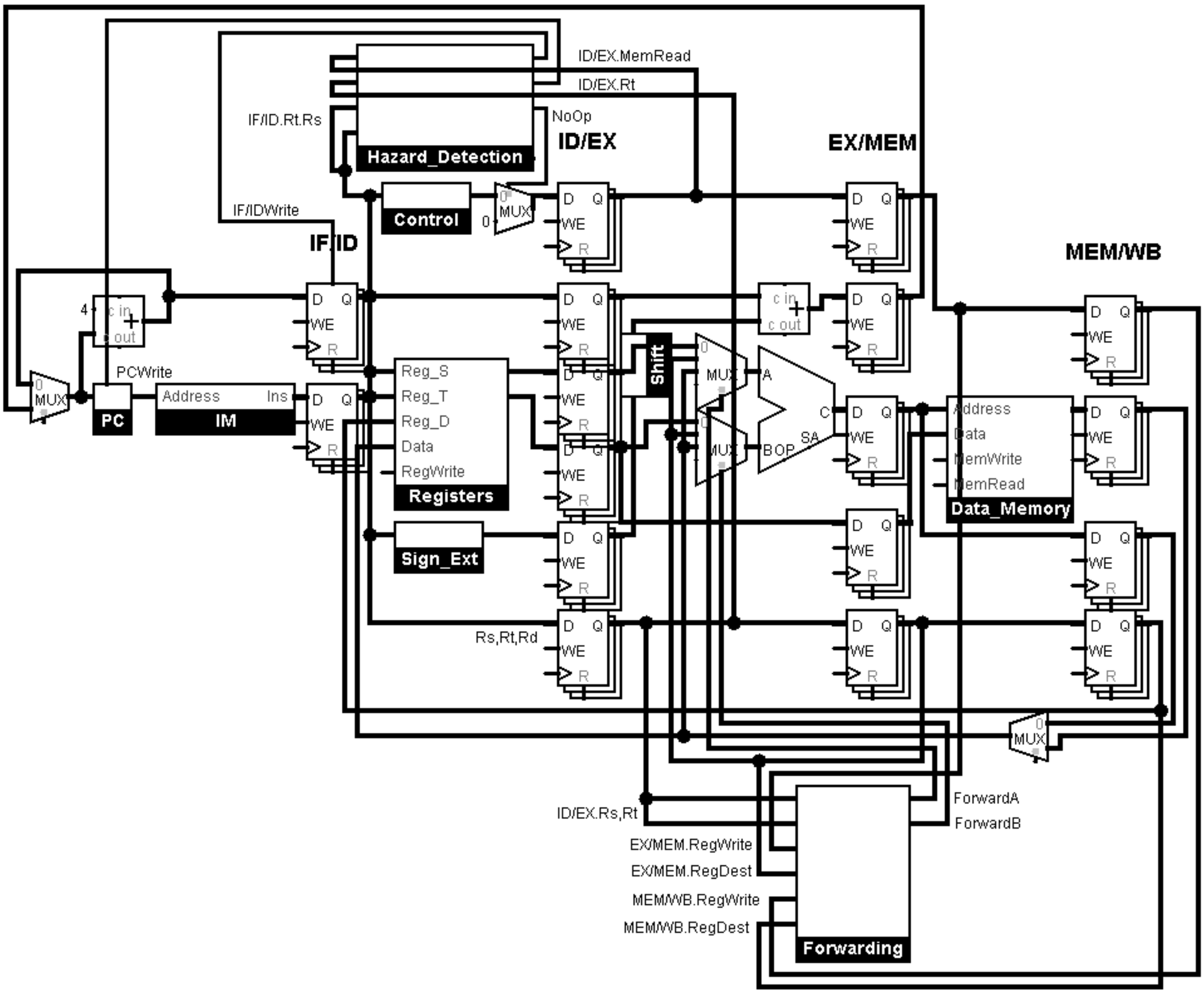

Figure 13: Pipeline with Data Hazard Handling

Control Hazards

- branches determine flow of control:
  - the fetching of the next instruction depends on the branch outcome
  - pipeline can not always fetch the correct instruction
- handling control hazards in the software:

  - if software is aware of the underlying pipeline ISA, software can insert instructions that are *not* guarded by a branch directly after it, to fill in the branch resolution delay
  - ie. have 2 free **branch delay slots** for instructions after each branch
    - can fill with no-ops as well if there are no available independent instructions
  - could take an instruction from *before* the branch, or from *one* of the paths after the branch:
    - above the branch instructions will always be executed vs. instructions after the branch may not always get taken
      - gives more options for instructions to pull in
    - restrictions on the liveness of these chosen instructions as well
  - *pros*:
    - never have to flush incorrectly predicted instructions
    - compiler will select independent instructions to pull in that do not affect correctness either path the branch takes
  - *cons*:
    - software is tightly coupled with specific machine ISA
    - adds control complexity
- handling control hazards in the hardware:
  1. just stall until we know which direction the branch will take
  2. *guess* which direction and start executing a chosen path:
     - must be prepared to undo any mistakes
       - how often and how much time is lost when the prediction is incorrect
     - with *static* branch prediction, base the guess on instruction type
     - with *dynamic* branch prediction, base the guess on execution history
  3. reduce the branch delay
     - difficult in reality to achieve
- stall on branch:
  - simply wait until the branch outcome is determined before fetching the next instruction
  - need to stall the instruction *fetch* rather than the instruction decode stage
  - forces a stall on every branch
    - stall penalty becomes unacceptable with more deeper, longer pipelines
- branch prediction:
  - predict outcome of branches and only have a penalty if prediction is incorrect
  - note that prediction was not used with data hazards since there is a large range of values to consider

        - with branch prediction, only have to predict between two options
    - in the MIPS pipeline:
        - can more easily predict branches *not* taken, just take `PC + 4`
        - fetch instruction after the branch, with no delay
    - more realistic branch predictions:
        - in static branch prediction, we predict based on typical branch behavior:
            - predict backward branches are taken eg. in loops
            - note that branch-not-taken is an example of static branch prediction
        - in dynamic branch prediction, we measure actual branch behavior:
            - eg. record recent history of each branch
            - assumes future behavior will continue trends
            - when incorrect, need to also update the history
    - have two consider two drawbacks impacting the CPI:
        - how often a branch is incorrect
            - impacted by software and hardware
        - how expensive is the penalty to correct erroneous predictions
            - only impacted by the hardware implementation
- when a prediction is incorrect, we need to perform a **flush**:
    - assume that the branch outcome is determined in the MEM stage
    - in order to flush, we set the control signals to 0 of instructions that should have been *skipped*:
        - want to perform no-ops for the remaining stages in these instructions
        - but what about the operations that were *already* performed for these invalid instructions?
            - could not have possibly reached the data memory or writeback stages yet
            - thus these operations have not done anything to the pipeline that cannot be removed by setting the control bits to 0
        - after the flush, instructions will not write to memory or the register file
    - the instruction we should have jumped to will execute normally, after the PC is updated when the branch is determined
        - not affected by the flush
    - flush essentially acts the same as a stall in the pipeline
        - but when the prediction is correct, we do not see any penalty at all
    - to check if the branch outcome was correct:
        - check `Branch AND 0` is equal to whether the branch was taken or not
        - implemented as a comparator on the branch outcome, and only write the PC if the prediction was incorrect

- reducing branch delay penalty:
    - there is no reason to wait until the MEM stage to determine the branch outcome
    - can we move the hardware to determine a branch outcome to an earlier stage?
    - eg. can move to the EX stage and save an instruction flush
    - eg. could even move the hardware all the way to the ID stage:
        - add a register comparator and a target address adder in this earlier stage
        - reduces the branch delay to just a single cycle
            - only have to pay a single cycle penalty when incorrect
- issues with branch prediction using a single cycle delay:
    - if a comparison register for the branch is a destination of a 2nd or 3rd preceding ALU instruction:
        - there is no current forwarding of data values to the ID stage, only the EX stage
        - thus need additional forwarding logic
    - if a comparison register for the branch is a destination of a preceding ALU instruction or 2nd preceding load instruction
        - cannot use forwarding alone, need an additional stall cycle
    - if a comparison register for the branch is a destination of a preceding load instruction
        - need two additional stall cycles
- in more advanced pipelines, branch penalty is much more significant:
    - thus, we want to rely on **dynamic branch prediction**
    - uses a **branch prediction buffer (BPB)** AKA branch / prediction history table:
        - indexed by recent branch instruction addresses
            - may only index by part of the PC for space reasons eg. lower 12 bits with 2 bottom zeros for a word addresses granularity
        - stores outcome ie. taken or not taken branch
    - to execute a branch, check table and expect the same outcome:
        - start fetching from the fall-through or target
            - need additional mechanism to keep track of target address
        - if wrong, flush pipeline and flip prediction
    - with a 1-bit predictor:
        - simply stores whether this branch was taken the last time it was taken
        - not enough **hysteresis** ie. inertia to track branch outcomes
        - eg. with an if-statement conditioned on whether a loop iteration is even, the 1-bit predictor would never be correct
        - eg. with inner and outer loops, mispredictions will occur when both leaving and reentering the inner loop

    - instead, use a 2-bit predictor:
        - each entry in the table has 2 bits, and the most significant bit is used to predict the branch outcome
        - adds hysteresis such that the prediction will only change on two *successive* mispredictions
        - fixes issues with 1-bit predictor
    - can also reference global history stored in a shift buffer:
        - hashing this with the PC allows us to distribute it along multiple branch locations
        - solves the even-odd looping branch
- even with a direction predictor, still need to *calculate* the target address:
    - the earliest we can have the effective address calculation done is the ID stage, *after* IF
        - no easy way to extract out the address calculation earlier
    - thus, use another table called the **branch target buffer (BTB)** that caches target addresses:
        - indexed by PC when instruction is fetched
        - if hit and instruction is branch predicted taken, can fetch target immediately
        - the first time we hit a branch, it will not be in the BTB
            - afterwards, the branch target addresses will be recorded into this cache
        - for indirect branches, we can save the target address taken, but this may not always be correct because the address is held in a register that could have been updated
- note that this simple directional branch prediction ignores more complex problems that must be solved in reality
    - eg. predicting indirect branches like jumps that go to an address in a register or function returns

Exceptions and Interrupts

- exceptions and interrupts are more forms of control hazards:
    - ie. *unexpected* events requiring changes in flow of control
    - **exceptions** are said to generally arise within the CPU
        - eg. undefined opcode, overflow, system call errors
    - **interrupts** are said to generally arise instead from external components:
        - are usually asynchronous
        - eg. external I/O controllers
    - difficult to handle these without sacrificing performance
- handling exceptions:
    - in MIPS, exceptions are managed by a **system control coprocessor**
    1. need to save the PC of the offending or interrupted instruction

    - in MIPS AKA the **exception program counter (EPC)**
  2. save indication of the problem:
     - in MIPS a code is stored in the **cause register**
     - eg. 1 may indicate overflow, 0 may indicate some undefined opcode
       - generally more than a single bit
  3. jump to the handler at a strict handler address
     - alternatively, with **vectored interrupts**, can have different specific handlers whose addresses are determined by the cause by indexing into a handler table
  - thus, exceptions are another form of a control hazard
    - eg. need to consider overflow in the EX stage and change the PC accordingly
- handler responsibilities:
  - need to read the cause and possibly transfer to a relevant second-level handler
  - if restartable, take corrective action and use the EPC to return to the program
    - technically, need to adjust EPC since `PC + 4` is saved
  - otherwise, terminate the program and report error with the EPC
- ex. Handling overflow in the EX stage from the add instruction `add r1, r2, r1`:
  1. need to prevent `r1` from being clobbered
  2. complete previous instructions
  3. flush the add and any subsequent instructions still in the pipeline
  4. set `Cause` and EPC register values
  5. transfer control to handler by setting the PC
  - very similar to mispredicted branch, except for writing to extra registers:
    - uses much of the same hardware
    - also need to stop the current instruction and allow it resume later after the exception
- since the pipeline overlaps multiple instructions, could have multiple exceptions at once:
  - eg. overflow along with undefined opcode in another pipeline stage
  - in the simplest approach, just deal with the exception from the earliest instruction ie. issue **precise exceptions**
    - flush subsequent instructions
  - in more complex pipelines, multiple instructions are issued per cycle or out-of-order completion is utilized
    - maintaining precise exceptions is very difficult
  - can alternative issue **imprecise exceptions**:
    1. stop the pipeline completely and save its state, including exception causes

        2. let the handler work out:
            * which instructions had exceptions
            * which to complete or flush
        * simplifies hardware, but much more complex handler software
        * not feasible for complex multiple-issue out-of-order pipelines

Extending the Pipelines

- there are several possibilities for extending the pipeline in an attempt to decrease cycle time:
    - pipelining more deeply may drop the cycle time, but additional forwarding logic could increase cycle time as well if the critical path becomes slower
    - in addition, the fundamental tradeoff between TC and CPI comes into play, since a lower TC results in increased CPI due to data and control hazards

1. split a single add into two staggered adds, separated by latches, eg. 32-bit into two 16-bit adders:
    - want to decrease TC while deepening the pipeline
    - most functionality in the split adder will still work:
        - pass carry-out from lower 16 bits to carry-in of upper 16 bits
        - need to pass both parts of the immediate to reach both ALU stages
        - SLT complicates things, since the MSB is fed backwards back to the `less` passthrough of the LSB
            * need to use an additional MUX at the second ALU to *overwrite* the least significant bit in the case of an SLT
    - in the case of a load-add dependency, now have to stall two cycles instead of just one
        - memory results become delayed with the extra ALU stage
    - there is a new dependency hazard in the case of an SLT with a dependent following instruction:
        - need additional stall cycle since lower bits of an SLT result are not finalized til the MUX at the *second* adder
        - unlike any other arithmetic instruction that can correctly forward the calculated lower bits
    - branch hazard recovery must flush an additional stage
    - data must be forwarded from multiple latches to both the 16-bit lower and upper ALUs:
        - to lower 16-bit ALU, need 3 additional forwarding paths for the lower 16 bits from E1/E2, E2/MEM, MEM/WB after `RegDst`
        - to upper 16-bit ALU, need 2 additional forwarding paths for the

        upper 16 bits from E2/MEM, MEM/WB:
          - but there is no way to access the *previous* instruction that *used* to be in MEM/WB
          - value from MEM/WB after `RegDst` needs to be latched into E1/E2 so that it is accessible even after the *original* instruction leaves the pipeline
          - additional 3rd forwarding path to upper ALU
          - not an issue anywhere else because those forwarding paths all still exist

2. split the IM stage into two parts:
    - data hazards are *not* affected since the loop between EX and WB has not been extended
    - have to flush extra stage during branch hazards
3. split the MEM into two parts:
    - load use dependency now requires two cycle delay
        - there is also an *additional* load use dependency on the *other* MEM component
    - additional data forwarding required

Instruction Level Parallelism

- in **instruction level parallelism (ILP)**, execute independent instructions at the same time:
    - in pipelining, already executing multiple instructions in parallel
        - some level of ILP
    - to increase the ILP:
        1. can make the pipeline deeper:
            - less work per stage and a shorter clock cycle
            - but increased hazards
        2. multiple-issue:
            - replicate pipeline stages or add multiple pipelines
            - start multiple instructions per clock cycle
            - CPI should ideally become less than 1, so **instructions per cycle (IPC)** may be a better metric
            - in reality, dependencies and hazards reduce this CPI
    - how effective is multiple-issue?
        - works, but has a certain limit and starts to give diminishing returns:
            - programs will have real dependencies that limit ILP
            - certain dependencies are very hard to eliminate eg. pointer aliasing
            - some parallelism is hard to expose eg. limited window size

during instruction issue

- difficult to keep pipelines full due to memory delays and limited bandwidth
- in general, seeing a turn towards multicore and mulithread CPUs and applications to avoid the diminishing return of optimizing ILP
    - however, most machines are still multiple-issue and dynamically scheduled

- for multiple-issue, how do we choose which instructions to issue together?
    1. in static multiple-issue, compiler groups independent instructions together:
        - compiler packages them into **issue slots**
        - compiler detects and avoids hazards
        - drawbacks of ISA specificity and non-portability
        - compiler may have to use no-ops, which increases IC
    2. in dynamic multiple-issue, the hardware will examine the instruction stream and chooses instructions to issue in each cycle:
        - the more common approach
        - CPU has to resolve hazards using complex techniques
- similarly to branch prediction, in **speculation**, we can guess what to do with an instruction:
    - start an operation as soon as possible
    - later, check whether guess was correct:
        - if so, complete the operation
        - otherwise, roll-back and re-do the correct operation
    - unlike in branch prediction, we allow the instructions to actually impact the state of the processor
        - more complex roll-back operation
    - compiler *can* perform speculation and reorder instructions:
        - eg. move load before branch
        - would need "fix-up" instructions to recover from incorrect guesses
    - hardware speculation:
        - look ahead for instructions to execute
        - buffer results until hardware determines they area actually needed
        - flush buffers on incorrect speculation
    - if an exception occurs during a speculatively executed instruction:
        - in software, needs ISA support for deferring exceptions
        - in hardware, needs to buffer exceptions until instruction completion

Static Multiple-Issue

- compiler groups instructions into **issue packets** ie. a group of instructions that can be issued in a single cycle:

  - essentially creates one very long instruction that specifies multiple concurrent operations:
    - AKA **very long instruction word (VLIW)**
    - instructions *must* be completely independent since they travel down the pipeline together and cannot forward
  - determined by pipeline resources required
  - *pros*:
    - avoids complex hardware design and implementation
  - *cons*:
    - very difficult to predict all stalls in software
      - eg. may depend on dynamic input data besides just the static program alone
    - can't always schedule around branches since branch outcome is dynamically determined
      - eg. limitations on reordering distance
    - limited portability, requires very ISA and even machine-specific compiler implementations
- compiler must also remove some or all of the hazards:
  - reorder instructions in issue packets
  - no dependencies *within* a packet
  - possibly some dependencies between packets
  - pad with no-ops if necessary
- ex. MIPS with static dual-issue:
  - allow for two-issue packets:
    - 1 ALU / branch instruction and 1 load / store instruction
    - 64-bit aligned, back to back instructions
      - pad unused instruction with a no-op
  - note that the paths are specialized:
    - only one instruction can access memory
    - the memory instruction will make use of an auxiliary adder to perform its address calculations during EX
    - specialization reduces the duplicated resources in the pipeline
  - most of the MIPS pipeline is the same in the dual-issue implementation:
    - register file functionalities, MUXs, etc. are all extended to support an additional instruction
      - heavy cost on extended register file ports
    - additional auxiliary adder
    - still only one memory component
      - note that in the ALU / branch instruction, the ALU output always goes to the register file, while in the load / store the memory output always goes to the register file, so there is some simplification in control complexity
- hazards with dual-issue MIPS:

  - more instructions execute in parallel
  - forwarding was used to avoid some stalls in single-issue:
    - now, cannot use ALU result in load / store in the same packet
    - need to split these into two packets, effectively a stall
  - more aggressive scheduling required

Ex. Schedule the following for dual-issue MIPS:

```
Loop: lw   t0, 0(s1)
      addu t0, t0, s2
      sw   t0, 0(s1)
      addi s1, s1, -4
      bne  s1, $0, Loop
```

Table 9: Scheduling Instructions into Packets

Cycle	ALU / branch	load / store
1	`nop`	`lw t0, 0(s1)`
2	`addi s1, s1, -4`	`nop`
3	`addu t0, t0, s2`	`nop`
4	`bne s1, $0, Loop`	`sw t0, 4(s1)`

- scheduling for dual-issue MIPS as seen in Table 9:
  - note that we will always have a load-use dependency between the `lw` and `addu` , so they must be separated by a full cycle
  - note that it is possible with our duplicated pipeline to have the `bne` and `sw` in the same cycle
  - note that the `addi` can be moved *anywhere* before the `bne` , because we can adjust the load and store offsets accordingly:
    - ie. the offset is known at compiler time because an add with an immediate was performed to `s1`
    - in our table, the `sw` has an anti-dependency that has to be fixed with the extra offset `4(s1)`
  - the IPC is $\frac{5}{4} = 1.25$ with a peak IPC of 2
- loop unrolling is one possible advantage that the static ILP approach can benefit from:
  - in loop unrolling, the loop body is replicated in order to expose more parallelism:
    - reduces loop-control overhead
    - allows more instructions to be packed together
  - need to use different registers per iteration through **register renaming**
  - determining how much to unroll a loop:
    1. how many registers are being used so that renaming is possible

2. stop after desired parallelism has been extracted
3. have an idea of how many times the loop should run, or if it should run a multiple of a number times
   * sometimes reordering an unrolled loop is not possible due to memory aliasing eg. in C

Ex. Unrolling the previous loop example 3 times:

```
Loop: lw   t0, 0(s1)
      addu t0, t0, s2
      sw   t0, 0(s1)
      addi s1, s1, -4
      lw   t0, 0(s1)
      addu t0, t0, s2
      sw   t0, 0(s1)
      addi s1, s1, -4
      lw   t0, 0(s1)
      addu t0, t0, s2
      sw   t0, 0(s1)
      addi s1, s1, -4
      lw   t0, 0(s1)
      addu t0, t0, s2
      sw   t0, 0(s1)
      addi s1, s1, -4
      bne  s1, $0, Loop ; gives 17 instructions

; however, all the addi's can be condensed to reduce the number of instructions
Loop: lw   t0, 0(s1)
      addu t0, t0, s2
      sw   t0, 0(s1)
      lw   t0, -4(s1)
      addu t0, t0, s2
      sw   t0, -4(s1)
      lw   t0, -8(s1)
      addu t0, t0, s2
      sw   t0, -8(s1)
      lw   t0, -12(s1)
      addu t0, t0, s2
      sw   t0, -12(s1)
      addi s1, s1, -16
      bne  s1, $0, Loop ; gives 14 instructions
```

Ex. Performing register renaming to reduce dependencies:

```
Loop: lw   t0, 0(s1)
      addu t0, t0, s2
      sw   t0, 0(s1)
      lw   t1, -4(s1)
      addu t1, t1, s2
      sw   t1, -4(s1)
      lw   t2, -8(s1)
      addu t2, t2, s2
      sw   t2, -8(s1)
      lw   t3, -12(s1)
      addu t3, t3, s2
      sw   t3, -12(s1)
      addi s1, s1, -16
      bne  s1, $0, Loop
```

Table 10: Scheduling Unrolled Instructions

Cycle	ALU / branch	load / store
1	`addi s1, s1, -16`	`lw t0, 0(s1)`
2	`nop`	`lw t1, 12(s1)`
3	`addu t0, t0, s2`	`lw t2, 8(s1)`
4	`addu t1, t1, s2`	`lw t3, 4(s1)`
5	`addu t2, t2, s2`	`sw t0, 16(s1)`
6	`addu t3, t3, s2`	`sw t1, 12(s1)`
7	`nop`	`sw t2, 8(s1)`
8	`bne s1, $0, Loop`	`sw t3, 4(s1)`

- in Table 10:
  - by putting the `addi` at the earliest cycle, which forces all the `lw` and `sw` offsets to be adjusted
    - *except* the very first load, which occurs in the same cycle of the `addi`
  - note that it would also have been possible to put the `addi` in cycle 7
  - *pros*:
    - the IPC is $\frac{14}{8} = 1.75$
      - closer to ideal IPC of 2
  - *cons*:
    - more register usage, which can lead to more spills and more loads and stores
    - in addition, the *static* code size is increased, which may require more instruction cache space in memory

Ex. Schedule the following for dual-issue MIPS, using compiler branch delay slots:

```
Here: lw   t0, 0(s0)
      add  t0, t0, s1
      lw   t0, 0(t0)
      add  t0, t0, s2
      sw   t0, 4(s0)
      addi s0, s0, 8
      bne  s0, s3, Here
```

Table 11: Scheduling Instructions with Branch Delay

Cycle	ALU / branch	load / store
1	`nop`	`lw t0, 0(s0)`
2	`nop`	`nop`
3	`add t0, t0, s1`	`nop`
4	`addi s0, s0, 8`	`lw t0, 0(t0)`
5	`bne s0, s3, Here`	`nop`
6	`add t0, t0, s2`	`nop`
7	`nop`	`sw t0, -4(s0)`

- in Table 11:
  - 2 branch delay slots are used
  - performance analysis:
    - to run 100 iterations on the loop, 704 total cycles are needed:
      - 4 cycles for fill time
      - 7 cycles for each iteration in the dual-issue pipeline
    - 14 instructions, including no-ops, are issued per iteration
      - CPI approaches $\frac{704}{1400} \approx 0.5$
    - note IC of program had to increase

Ex. Unrolling and renaming registers from the previous loop:

```
Here: lw   t0, 0(s0)
      add  t0, t0, s1
      lw   t0, 0(t0)
      add  t0, t0, s2
      sw   t0, 4(s0)
      lw   t1, 8(s0)
      add  t1, t1, s1
      lw   t1, 0(t1)
      add  t1, t1, s2
```

```
    sw   t1, 12(s0)
    addi s0, s0, 16
    bne  s0, s3, Here
```

Table 12: Scheduling Unrolled Instructions with Branch Delay

Cycle	ALU / branch	load / store
1	`nop`	`lw t0, 0(s0)`
2	`nop`	`lw t1, 8(s0)`
3	`add t0, t0, s1`	`nop`
4	`add t1, t1, s1`	`lw t0, 0(t0)`
5	`addi s0, s0, 16`	`lw t1, 0(t1)`
6	`add t0, t0, s2`	`nop`
7	`bne s0, s3, Here`	`sw t0, -12(s0)`
8	`add t1, t1, s2`	`nop`
9	`nop`	`sw t1, -4(s0)`

- in Table 12:
  - 2 branch delay slots are again used
  - loop unrolling allowed a `bne` and `addi` to be removed ie. consolidated
    - registers were renamed in order to reduce dependency chains
  - note the structural hazard that forces the `bne` to have two delay slots after it
    - `add, sw` chain has to be offset by one cycle to make space for the `bne`
  - performance analysis:
    - to run 100 iterations on the loop, $50 \times 9 + 4 = 454$ total cycles are needed:
      - still 4 cycles for fill time
      - 9 cycles to run *two* iterations
    - 18 instructions are issued per iteration
      - CPI approaches $\frac{454}{1800} \approx 0.25$

Dynamic Multiple-Issue

- dynamic multiple-issue processors are known as **superscalar** processors:
  - CPU decides whether to issue multiple instructions each cycle
    - responsible for avoiding structural and data hazards
  - avoids the need for compiler scheduling, though it may still be beneficial
    - compiler scheduling can solve longer distance optimization prob-

lems

Ex. Dynamically schedule the following:

```
lw   t0, 20(s2)
addu t1, t0, t2
sub  s4, s4, s3
slti t5, s4, 2
```

- note that the subtraction can be performed alongside the load:
  - independent instructions
  - thus allow the CPU to execute instructions out of order to avoid stalls
    - but still *commit* results to register *in order*
- dynamically scheduled CPU overview:
  1. a phase where we fetch and decode instructions *in-order*
     - do not yet read from registers yet
  2. a resource allocation phase:
     - renaming *logical* registers to *physical* registers through a **register allocation table (RAT)**:
       - though the number of logical registers is bounded by the ISA eg. register number bits, the number of physical registers is only bounded by hardware limitations
       - modern machines use around 100 physical registers for renaming
     - note that since this is an alternative to static instruction reordering in the compiler, the hardware is essentially performing limited liveness analysis based on the stream of writes it is receiving
     - overlapping instructions uses the additional register space in order to free up the antidependencies
       - allows us to do more in parallel
     - set up an in-order structure called the **reorder buffer (ROB)**
       - provides the illusion to the system that instructions are executed in-order
  3. a phase of *out-of-order* execution:
     - a series of reservation stations connected to functional units
     - speculative execution that doesn't commit any changes to registers
  4. an *in-order* commit phase:
     - using the ROB to commit in-order
     - should handle any incorrect speculative executions and recover by flushing accordingly
- implementing out-of-order execution:
  - have several **reservation stations** for instructions to sit and wait before being used in a **functional unit** eg. an ALU:
    - ie. staging areas for an instruction to wait until its register

operands are ready

        - when an operation completes, it will notify all the reservation stations that its value is ready
            - instructions are done out-of-order, depending on when dependencies are ready
    - on instruction issue to reservation station:
        - if operand is available in register file or reorder buffer, copy it to the station
            - operand register is no longer required, can be overwritten
        - if operand is not yet available, it will be provided later to the reservation station by a function unit
            - register update may not be required
- dynamic speculation:
    - CPU can perform aggressive speculation due to the new commit phase
        - eg. predict branches, reorder load
    - avoid committing until the correct outcome is determined ie. the speculation is cleared
        - ie. rampant speculation in execution stage is *validated* in the commit phase

Memory Hierarchy

- the memory hierarchy is typically made up of the following tiers of memory:
    1. **static RAM (SRAM)**:
        - 0.5ns to 2.5ns delay, between $2000 and $5000 per GB
        - cache memory attached to the CPU
            - may have multiple cache sizes in the hierarchy, eg. L1, L2, L3 caches
    2. **dynamic RAM (DRAM)**:
        - 50ns to 70ns delay, between $20 and $75 per GB
        - typically stored as charge in a capacitor
            - memory reads are destructive, must rewrite after reading
        - volatile, loses data when power is lost
        - middling latency and capacity
    3. **magnetic disk**:
        - 5ms to 20ms delay, up to $2 per GB
        - disk is a nonvolatile, rotating magnetic storage
            - mechanical latency
    - flash storage is a nonvolatile semiconductor storage:
        - 100x to 1000x faster than disk and takes lower power, but more expensive per GB
        - replacing magnetic disks in some scenarios
    - difficult to achieve an *ideal* memory with access time of SRAM with the capacity and cost efficiency of disk
        - no perfect memory solution, so this motivates us to take exploit the memory hierarchy to its fullest
- memory developments over time:
    - DRAM cost has decreased significantly, but the latency not scaled as dramatically
        - phenomenon known as the **memory wall**
    - CPU transistors are scaling faster than memory development is
        - bottleneck of getting data to the processor fast enough
- in the **principle of locality**:
    - programs access a *small* proportion of their address space at any time
    - in *temporal* locality:
        - items accessed recently are likely to be accessed *again* soon
        - eg. instructions in a loop, induction variables
    - in *spatial* locality:
        - items *near* those accessed recently are likely to be accessed soon
        - eg. sequential instruction access, array data
    - thus to take advantage of locality:

    - when we access something, we can bring it in a certain granularity up a layer of the hierarchy
    - addresses spatial *and* temporal locality
    - want to maximize **hits** ie. accesses satisfied by upper levels and minimize **hits**
  - note that locality depends on memory access patterns:
    - greatly depends on the specific software, eg. algorithm behavior
    - in addition to compiler optimizations for memory access
  - **caching** takes advantage of locality to give the *illusion* of fast, large memories
    - higher levels of the memory hierarchy *cache* data from lower levels that would be dramatically slower to access
- types of misses and related cache design tradeoffs:
  - **compulsory misses** AKA cold start misses occur at the first access to a block
    - may be solved with prefetching
  - **capacity misses** occur due to finite cache size:
    - a replaced block is later accessed again
    - if a fully associative cache of the same size with the same unique accesses *would* have missed, it is a conflict miss
    - may be solved by increasing cache size
  - **conflict misses** AKA collision misses occur in a non-fully associative cache:
    - due to competition for entries in a set
    - if a fully associative cache of the same size with the same unique accesses *would* have hit, it is a conflict miss
    - may be solved by increasing associativity
  - cache design considerations:
    - increasing the cache size
      - decreases capacity misses, but may increase access time and power requirements
    - increasing the associativity
      - decreases conflict misses, but may increase access time and power even *more* dramatically
    - increasing block size
      - decreases compulsory misses, but increases the miss penalty and may also increase miss rate due to pollution
- common principles that apply at all levels of the memory hierarchy:
  1. block placement:
     - determined by associativity
     - higher associativity reduces miss rate, but increases complexity, cost, and access time
  2. finding a block:

   - different associativities require a different number of tag comparisons
   - reduce comparisons to reduce cost
   - note that in virtual memory, we have a fully associative cache that requires zero tag comparisons:
     - ie. the page table itself acts as a fully associative cache
     - mapping the *entire* virtual space to a physical address space, so no need for tag checks
       - no real miss rate, just a page fault that can occur to bring a block up into main memory
     - but requires an additional TLB cache on top of virtual memory, which does use tags
3. replacement on a miss:
   - eg. LRU, random
   - virtual memory typically uses an LRU approximation that requires much hardware support
4. write policies:
   - in write-through, update both levels
     - simpler replacement, but may require write buffer
   - in write-back, update upper level only:
     - update lower level when block is replaced
     - need to keep more state
   - with virtual memory, only write-back is feasible given disk write latency

Cache Memory

- **cache memory** is the level of the memory hierarchy closest to the CPU:
  - caches are implemented with SRAM
  - usually three levels of cache L1, L2, L3
    - in increasing size and delay
  - how do we know if the data is even present, and then where do we look?
    - cache is a *subset* of memory
  - on cache hits, the CPU can proceed normally
  - however, on a cache miss, the CPU pipeline has to be stalled while the block is fetched from the hierarchy
    - misses also occur in instruction cache accesses
  - note that advanced, out-of-order CPUs can execute instructions during cache miss:
    - much harder to analyze, since effect of miss depends on the program data store

        - typically use system simulation
- multilevel cache considerations:
    - in the primary cache, want to achieve the minimal hit time
    - for lower caches such as L2:
        - focus on low miss rate to avoid main memory access
        - hit time has less overall impact since cache capacity is larger than primary cache
    - thus L1 cache is usually smaller than other caches
        - L1 block size is also typically smaller than the L2 block size
- block size considerations:
    - the block size determines the size of the byte offset field
        - eg. for a 32 byte block, need $log_2 32 = 5$ bits for offsets
    - with a larger block size:
        - can exploit *more* spatial locality by bringing in more data on a miss
        - however, with a fixed-sized cache, larger blocks means fewer total blocks in the cache at a time:
            - leads to more competition and potentially an increased miss rate
            - ie. causes pollution of the cache
        - larger miss penalty when hardware has to pull in more data into memory
            - critical-word-first ie. moving the requested memory words first can help mitigate this
- example address subdivision of a 32-bit address:
    - lower 2 bits are the **byte offset**:
        - indicates which particular offset of the block we are referring to in memory we want to read out from
        - *not* used by the cache access, since the cache holds data in a block granularity
        - used later when actually reading the block
    - next 10 bits are the **index**:
        - number of bits used in the index is based on the size of the cache
        - need 10 bits to index into a cache with $2^{10} = 1024$ elements
    - last 20 bits are the **tag**:
        - uses the remaining bits
        - compares against the cache tag to see if desired block is already in cache
            - also need to AND with the valid bit to check cache hits
- how do we handle stores to data held in cache?
    - ie. handling write *hits*
    - the block is held in both cache and somewhere else in the memory hierarchy
    - if we only update the block in cache, cache and memory become incon-

sistent

  - we can perform **write-through** ie. update memory as well:
    - writes take much longer, so the CPI tanks
    - but no delay on evictions
  - alternatively, we can use a **write buffer** AKA **write-back**:
    - hold data *waiting* to be written to memory:
      - set a **dirty bit** that indicates whether the block has been updated
      - ie. accumulate modifications over time
    - actually write the block when it is evicted from memory
    - CPU continues immediately, and only stalls on write if the write buffer is already full
  - different layers of the hierarchy may use different write policies
    - typically higher levels like the L1 use write-through, while lower levels like the L2 use write-back
- what happens on a write miss?
  - ie. writing to something that is not in cache
  - may want to bring data up into the cache, since there is not necessarily locality to exploit between loads and stores
    - may unnecessarily pollute the cache
  - in **write-allocate**:
    - allocate a block in the cache on miss, bring in the data, and modify the data
      - data modification will follow the chosen write hit policy for the cache ie. write-through vs. write-back
    - useful when there is high storage locality between loads and stores
  - in **write-around**:
    - don't bring the data into the cache and instead just write to the next level of the memory hierarchy
    - does not pollute the cache
  - typically, write-through needs to use both write-allocate and write-around depending if the data is in cache
    - while write-back will use write-allocate since it needs to mark the dirty bit in the cache

Cache Performance

- ex. Measuring cache performance:
  - given the following:
    - instruction cache miss rate of 2%
    - data cache miss rate of 4%

    - miss penalty of 100 cycles
    - base CPI of 2
    - loads and stores make up 36% of instructions
  - we can can find the miss cycles per instruction:

$$\textit{memory stall cycles} = \frac{\textit{instruction}}{\textit{program}} \times \frac{\textit{misses}}{\textit{instruction}} \times \textit{miss penalty}$$

    - for the instruction cache, $0.02 \times 100 = 2$
    - for the data cache, $0.36 \times 0.04 \times 100 = 1.44$
  - actual CPI is $2 + 2 + 1.44 = 5.44$
    - ideal CPU is $\frac{5.44}{2} = 2.72$ times faster
  - we can see that as CPU performance increases, the miss penalty becomes much more significant
    - needs to be evaluated when determining system performance
- ex. Measuring the **average memory access time (AMAT)**:
  - hit time is another important metric
  - to calculate AMAT:

$$\textit{AMAT} = \textit{hit time} + \textit{miss rate} \times \textit{miss penalty}$$

  - given the following:
    - CPU with 1ns clock
    - hit time of 1 cycle
    - miss penalty of 20 cycles
    - instruction cache miss rate of 5%
  - $\textit{AMAT} = 1 + 0.05 \times 20 = 2$ nanoseconds
    - an average access time of 2 cycles per instruction
- ex. Measuring AMAT for multilevel caches:
  - given the following:
    - block size of 32 bytes
    - L1 cache with 20% miss rate, latency of 2 cycles
    - L2 cache with 10% miss rate, latency of 12 cycles
    - L3 cache with 5% miss rate, latency of 30 cycles
    - memory with latency of 100 cycles
  - $\textit{AMAT} = 2 + 0.2 \times (12 + 0.1 \times (30 + 0.05 \times 100)) = 5.1$ cycles
- ex. Measuring AMAT for non-unified caches:
  - given the following contraints:
    - L1 instruction cache with 10% miss rate, latency of 1 cycle
    - L1 data cache with 15% miss rate, latency of 1 cycle
    - L2 cache with 20% miss rate, latency of 15 cycles
    - memory with latency of 150 cycles
    - 20% of instructions are loads and stores
  - $\textit{AMAT} = (1 + 0.1 \times (15 + 0.2 \times 100) + 0.2 \times (1 + 0.15 \times (15 + 0.2 \times 100))) \times \frac{1}{1.2}$ cycles

    - weighted average calculation
- ex. Measuring the impact of multilevel caches:
  - given the following:
    - CPU with base CPI of 1 and a clock rate of 4GHz
    - miss rate per instruction of 2%
    - main memory access time of 100ns
  - with just a primary L1 cache:

$$\textit{miss penalty} = \frac{100ns}{0.25ns} = 400 \textit{ cycles}$$
$$CPI = 1 + 0.02 \times 400 = 9$$

  - with an additional L2 cache with 5ns access time and 10% miss rate:

$$\textit{miss penalty} = \frac{5ns}{0.25ns} = 20 \textit{ cycles}$$
$$CPI = 1 + 0.02 \times (20 + 0.01 \times 400) = 2.2$$

    - performance increase of $\frac{9}{2.2} = 4.1$ times
- ex. Full CPI calculation:
  - pipeline and instruction mix details:
    - 5 stage pipeline with full forwarding
    - branches resolved in the EX stage, branch predictor predicts not-taken
    - 50% R-type, 20% loads, 10% stores, 20% branches
    - dependencies follow a load for 20% of loads
    - 25% of branches are taken
    - store misses do not stall the pipeline
  - memory details:
    - unified L1 cache for both instruction and data with 1 cycle delay
    - L1-I misses 10%, L1-D misses 5%
    - L2 misses 5%, has 20 cycle delay
    - memory has 100 cycle delay
  - need to calculate the base CPI from the pipeline and the CPI incurred

by the memory hierarchy to find the *total* CPI:

$$\begin{aligned} BCPI &= CPI_{peak} + T_{load\text{-}use\ hazard\ penalty} + T_{branch\ hazard\ penalty} \\ &= 1 + 0.2 \times 0.2 \times 1 + 0.2 \times 0.25 \times 2 \\ &= 1.14 \end{aligned}$$

$$\begin{aligned} MCPI &= MCPI_{inst} + MCPI_{data} \\ &= 1 \times 0.1 \times (20 + 0.05 \times 100) + 0.2 \times 0.05 \times (20 + 0.05 \times 100) \\ &= 2.75 \end{aligned}$$

$$CPI = BCPI + MCPI = 3.89$$

        * note that the 1 cycle delay of the L1 cache defines the pipeline stages used
            - if the L1 instruction cache required more cycles, IF would have to be pipelined across multiple stages
        * in addition, the percentage of loads in the instruction mix is necessary to factor into the MCPI
        * importantly, the AMAT calculation is *not* the same as the MCPI calculation
            - MCPI *ignores* the time taken on a hit

- ex. Full CPI calculation where the L1 cache now has a 2 cycle delay:
    - the IF stage is now broken into two stages
    - increased size of the cache and decreased miss rate from 10% to 5%:
        * there are now 3 stages to flush for incorrect branch predictions
        * load-use hazards are *not* affected
    - given the same statistics from previous example:

$$\begin{aligned} BCPI &= CPI_{peak} + T_{load\text{-}use\ hazard\ penalty} + T_{branch\ hazard\ penalty} \\ &= 1 + 0.2 \times 0.2 \times 1 + 0.2 \times 0.25 \times 3 \\ &= 1.19 \end{aligned}$$

$$\begin{aligned} MCPI &= MCPI_{inst} + MCPI_{data} \\ &= 1 \times 0.05 \times (20 + 0.05 \times 100) + 0.2 \times 0.05 \times (20 + 0.05 \times 100) \\ &= 1.5 \end{aligned}$$

$$CPI = BCPI + MCPI = 2.69$$

- ex. Full CPI calculation where the MEM stage is broken into two stages:
    - reduces miss rate for L1 data cache from 5% to 2%
    - the load-use hazard now has a two cycle flush penalty:

        - introduces a new load-independent-use hazard with a single cycle penalty
        - need to know this new hazard percentage eg. 10%
    - given the same statistics from earlier example:

$$\begin{aligned} \mathit{BCPI} &= \mathit{CPI}_{\mathit{peak}} + T_{\textit{load-use hazards penalty}} + T_{\textit{branch hazard penalty}} \\ &= 1 + 0.2 \times 0.1 \times 1 + 0.2 \times 0.2 \times 2 + 0.2 \times 0.25 \times 2 \\ &= 1.2 \end{aligned}$$

$$\begin{aligned} \mathit{MCPI} &= \mathit{MCPI}_{\mathit{inst}} + \mathit{MCPI}_{\mathit{data}} \\ &= 1 \times 0.1 \times (20 + 0.05 \times 100) + 0.2 \times 0.02 \times (20 + 0.05 \times 100) \\ &= 2.6 \end{aligned}$$

$$\mathit{CPI} = \mathit{BCPI} + \mathit{MCPI} = 3.8$$

Associativity

- in a **direct mapped cache**:
    - has the lowest overhead to implement
    - the location is determined by the address:
        - for a particular address, it is direct mapped to only one choice
        - `block_address % blocks_in_cache` ie. similar to a hash table
            - here, multiple memory addresses may map to the *same* cache block
    - how to know *which* particular block is stored in a cache location?
        - store part of the block address as well as the data in the **tag**
        - the **valid bit** determines if there is no data in a location, initially 0
    - on a cache miss, we bring in the data from the memory hierarchy and update the metadata in the cache
        - on a cache hit after checking the valid bit and tag, we can take the data directly from the cache
    - the key limitation of a direct mapped cache:
        - we may have a cache miss that evicts ie. *overwrites* a mapping in the cache because it shares the same mapping
        - even though there may still be *unused* mapping entries in the cache
- **associativity** determines how many entries in the cache can be used for a particular block in memory:
    - in a direct map, there is only 1 location in cache for a memory cache
        - less flexibility, but only have to check a single location
    - in a **fully associative** cache:

        - a given block can go in any *cache* entry
        - total flexibility, but requires all entries to be searched at once
            - requires an expensive comparator per entry as well
        - typically only used with very specific, small caches
        - thus the index field is not needed, since every entry will be checked
    - in an n-**way set associative** cache:
        - the cache is organized into sets:
            - each set contains n entries
            - a particular address can go into any one of these entries
        - fully associative and direct maps lie at either end of the n-way set associativity
        - similar to a hash table with a fixed number of buckets
        - search all entries in a given set at once
            - only need n comparators
        - thus the index maps to a particular set rather than a cache
    - increased associativity decreases miss rate
        - but with diminishing returns
- associativity also determines the replacement policy used:
    - the **replacement policy** chooses which entry to evict from the cache
    - in a direct map, there is no choice
        - must evict out the current cache mapping
    - in set associative caches, there are at least two choices for what to evict:
        - first and foremost, prefer non-valid entries, if there is one
        - in **least-recently used (LRU)**:
            - choose the entry unused for the longest time
            - simple for 2-way, manageable for 4-way, much more difficult beyond that
            - can use the clock algorithm as a more efficient heuristic
        - in a **random** replacement policy
            - approximates the same performance as LRU for high associativity
        - FIFO, LIFO, etc.

Virtual Memory

- main memory can act as a cache for secondary ie. disk storage:
    - managed jointly by CPU hardware and the OS
    - programs are able to share main memory through a technique called **virtualization**:
        - each has a private *virtual* address space holding its frequently used code and data

            - an additional level of indirection
        - protected from other programs
    - CPU and OS is responsible for translating virtual addresses to physical addresses:
        - a **virtual memory (VM)** block is called a **page**
            - distinct from the smaller block granularity
        - a VM translation miss is called a **page fault**
    - in caching, hardware moved blocks between L1, L2 caches and memory
        - in VM translation, the OS explicitly moves pages between memory and disk
- responsibilities of VM:
    - protection between programs
    - possible sharing between programs
    - independent compilation
    - hardware independence
- through **address translation** the virtual addresses assigned to a program by the linker are translated into physical addresses during program execution:
    - addresses are made up of a **virtual page number** and a **virtual page offset**
    - need to convert the virtual page number into a physical page number:
        - page offset remains the same after address translation
        - may map between different memory sizes eg. from a 32-bit virtual address to a 30-bit physical address
- on a page fault, the page must be fetched from disk:
    - takes millions of clock cycles
    - handled by the OS vs. caches misses are handled by hardware
    - minimizing the page fault rate is done in software:
        - fully associative placements
        - smart replacement algorithms
    - on the other hand, hardware is responsible for making page *hits* faster
- the **page table (PT)** stores placement information for virtual memory:
    - is an array of **page table entries (PTE)**, indexed by virtual page number
    - is a *memory*-resident structure that is pointed to by the **page table base register (PTBR)** in the CPU
        - the PTBR is a *process*-specific pointer that is different for each unique *running* process
    - if the page is present in memory:
        - PTE stores the physical page number
        - alongside other status bits eg. valid, present, referenced, dirty, etc.
    - to access the page table and find a specific entry (when pages are not in the TLB), a hardware structure called a **page table walker** is used:
        - scans PT based on PTBR
        - walker may find PTE in cache, or go all the way to memory

        - some systems do not cache PTEs outside of the TLB, but caching PTEs in the L1 or L2 cache will cache the page table based on a block granularity, which can benefit from locality
            - rather than the TLB caching on a *translation* level granularity
    - if the page is not present, a page fault occurs
        - PTE refers to location in swap space on disk
    - to reduce the page fault rate, LRU replacement is typically used:
        - the referenced bit AKA use bit is set to 1 on page access
        - a page with a zero reference bit has not been used recently
    - however, the page table *itself* may not always fit in memory:
        - would have to page the page table
        - ie. incur multiple loads for a single load operation
- disk writes take millions of cycles:
    - write-through is impractical, and write-back is usually used instead
    - amortize writes by collecting them in a write buffer
- how to we make translation fast?
    - address translation appears to require *extra* memory references, since the PT is held in memory
        - one to access the PTE, and then another to access the actual memory access
    - but page table accesses have good locality:
        - due to the fact that pages are huge
        - use a fast cache of PTEs *within* the CPU in a **translation lookaside buffer (TLB)**
    - TLBs will be typically smaller than regular caches, since they only have to store translations
        - typical systems use multiple levels of TLBs
    - TLB is *process*-specific, and will need to be flushed on context switches
        - can sometimes be shared to keep the cache hot without clearing it
    - TLB is like any other hardware cache, has certain associativity and size
        - use index and tag to find entries
    - typical TLB statistics:
        - 16-512 PTEs
        - 0.5-1 cycle on a hit
        - 10-100 cycles on misses
        - 0.01-1% miss rate
    - handling TLB misses:
        - if page is in memory, the page table walker needs to load the PTE from memory and retry
        - a **page fault** will occur if page is not in memory:
            - OS handles fetching the page and updating the page table
            - then, restart the faulting instruction
    - note that TLB accesses, page table accesses, and address translations are

handled completely by hardware except for page faults

- TLB miss handler:
  - TLB miss needs to indicate to the pipeline that the PTE will come from memory
  - ie. need to recognize a TLB miss *before* the destination register is overwritten
    - stop the request coming from the TLB cache from actually writing in the case of a load, etc.
  - handler then copies PTE from memory to TLB, then restarts instruction
    - if page is not present, a page fault would occur
- there are several ways to use tags with virtual and physical addresses in the TLB and regular caches:
  - regular caches cache *physical* memory values
  1. regular cache tag uses a physical address ie. **physically indexed physically tagged (PIPT)**:
     - need to translate before regular cache lookup occurs
     - pay latency of TLB first
  2. regular cache tag uses a virtual address ie. **virtually indexed virtually tagged (VIVT)**:
     - can directly lookup regular cache with virtual address and access the TLB in parallel
       - need physical address in case cache misses and to see protection bits from PTE etc.
     - complications due to aliasing where there are *different* virtual addresses for shared physical addresses
     - need to augment regular cache with a process specific ID
  3. in a hybrid approach, regular cache uses a virtual address for the index, but the physical address after translation for the tag:
     - ie. **virtually index physically tagged (VIPT)**
     - works if the regular cache index and offset field widths are subsumed within the page offset field
       - page offset remains consistent across translation, so regular cache index and offset will not change between the physical and virtual address
     - allows us to immediately access the cache while accessing the TLB in parallel
       - essentially PIPT, but performing accesses in *parallel*
     - when TLB completes the translation, we can compare the tags in the cache
       - tag does change after translation
- ex. Control flow on a load instruction using VIPT address translation:
  1. can immediately index into the L1 cache from the virtual address
  2. in parallel, check the TLB to get the physical address tag to compare

with the L1 cache tag:
  - on a TLB hit, we succesfully get the tag, and can check the L1 cache tag
    - ⋆ the L1 cache can hit or miss
      - · on a hit we are done, while a miss requires going down the memory hierarchy
  - on a TLB miss, the PTE is not in cache, and we need to go to the page table

3. the page table walker will check the L1 cache for the desired page table entry:
   - on a hit, we get the PTE, add it to the TLB, and get the tag to compare with the L1 cache tag
   - on a miss, we need to go down the memory hierarchy, to memory
4. if not in L1, the walker will check memory for the desired page table entry:
   - on a hit, we get the PTE, add it to the L1 cache and then to the TLB, and get the tag to compare with the L1 cache tag
   - on a miss, we have a page fault, and the OS has to handle

- ex. Examining address field views from paging vs. TLB vs. cache perspectives:
  - given the following:
    - ⋆ 2^{64}-byte virtual address space, 2^{34}-byte physical address space, 2^{16}-byte pages
    - ⋆ 8-way associative VIPT TLB with 64 entries
    - ⋆ 4-way L1 cache has 2^{12} bytes of space and a block size of 2^5 bytes
  - we can infer:
    - ⋆ virtual memory thus has $\frac{2^{64}}{2^{16}} = 2^{48}$ pages, and physical memory has $\frac{2^{34}}{2^{16}} = 2^{18}$ pages
    - ⋆ the TLB has $\frac{64}{8} = 8$ indices
    - ⋆ the L1 caches has $\frac{2^{12}}{2^5 \times 2^2} = 2^5$ indices
  - addresses from paging perspective:
    - ⋆ virtual addresses have 16 bits to specify the page offset and 48 bits for a virtual page number
    - ⋆ physical addresses also have 16 bits to specify the page offset and 18 bits for a physical page number
  - each page table entry will need to hold at least 18 bits for the PPN, along with 6 bits for protection information for 24 bits or 3 bytes total:
    - ⋆ PT overall has 2^{48} entries and size $2^{48} \times 3$
      - · very large, would have to be paged
  - addresses from the TLB perspective:
    - ⋆ virtual addresses still have the lower 16 bits for the page offset
      - · next 3 bits are the index, and last 45 bits are the tag
    - ⋆ on TLB hit, TLB delivers the 18-bit PPN and the propagates the

16-bit offset to form the physical address

- addresses from the L1 cache perspective:
    - virtual addresses have the lower 5 bits for the block offset
        - next 5 bits are the index, remaining $48 + (16 - 10) = 54$ bits are left unused since the cache is VIPT
    - physical addresses have the lower 5 bits for the block offset
        - next 5 bits are the index, remaining $18 + (16 - 10) = 24$ bits are used as the tag
    - the cache is accessed with either 5-bit index from physical or virtual address (the same value, since the page offset is the same) together with the 24-bit tag from the physical address

Cache Coherence

- in multiprocessor or multicore shared caches:
    - data is migrated to many local caches in a multiprocessor design
        - reduces bandwidth for shared memory
    - read-shared data is replicated
        - reduces contention for access
    - these patterns cause cache coherency issues
- **cache coherence** deals with the multiple CPU case where multiple cores share a physical address space:
    - each has a private cache, while *sharing* a lower level cache and/or memory
        - note that the private caches may have different architectures, but block size will usually be the same
    - the changes made by one CPU must be seen by the other, ie. trickle up levels of the memory hierarchy to cores and their private caches:
        - ie. the caches must be kept consistent with one another
        - eg. if the private cache is write back, it will not flush its changes on write to the lower level of the shared hierarchy
        - eg. if the private cache is write through, the shared hierarchy will be updated, but if there is a copy of the value in the cache of the other core, it will not reflect the updated value
        - in contrast to cache *consistency* which requires trickling down levels of the memory hierarchy
    - thus in coherence, we want reads to always return the most recently written value:
        - if P writes X and then P reads X with no intervening writes, the read should return the written value

    - if P_1 writes X and then P_2 reads X sufficiently later, the read should return the written value
    - if P_1 writes X and then P_2 writes X all processors see writes in the same order
      - end up with the same final value for X

- cache coherence protocols include snooping, snarping, or directory-based

- in a **snooping protocol**, each cache monitors reads and writes on a **snoop bus**, thus placing the burden on the caches:

  - caches get exclusive access to a block when it is written
  - each core cache monitors the snoop bus:
    - bus listening does not increase latency, we can add additonal ports into the cache to perform writes and modifications depending on inputs from the snoop bus
    - has additional power and area requirements for the cache, but no increased latency
  - the cache broadcasts an *invalidate* message for the specific written block and its block address on the snoop bus
    - other caches detect the snoop bus message, and set the valid bits of that block to 0
  - subsequent read in another cache misses:
    - owning cache supplies updated value
    - typically the private caches will be write through with snooping so that we end up writing to a shared cache
      - on cache miss, can take value from a consistent shared cache level
  - to prevent issues with simultaneous writes, an arbiter in the hardware is used to ensure only one core writes at a time
  - *pros*:
    - best suited for a few number of cores and frequent sharing between cores
  - *cons*:
    - major issues with scalability
    - with very many cores, may need multiple snoop buses

- in another technique called the **snarping protocol**, invalidate *and* update the overwritten block

  - same scaling issues as snooping protocol

- in a **directory-based protocols**, caches and memory record sharing status of blocks in a directory:

  - the **directory** maps cache blocks to a state

        - eg. each cache block has an MSI state of modified, shared, or invalid
    - handshaking protocol occurs between the directory and its participating core:
        - core can send messages like `GetM, GetR` to the directory to request to modify or read a certain address
            - directory must modify the corresponding cache block state accordingly, based on a state diagram
        - directory can send messages like `WB, WBI, INV, ACK`
            - ie. write back, write back invalidate, invalidate, acknowledgement
    - key difference with snooping is that in snooping, each core is responsible for their own cache to maintain their coherency:
        - with directories, we have an intermediate service that has the responsibility of maintaining all cores' coherency
        - the directory approach adds additional latency for shared variables compared to snooping, but makes the unshared cases faster

- ex. Cache consistency and coherence example:
    - given the following:
        - two L1 WT, WAR caches for two different cores
            - initially both hold address `4096` and a value of `42`
        - a L2 WB, WAL cache shared between the cores
            - snooping done on the write to L2

    1. after running `sw t0, 4(s0)` on core 0, if `t0 = 30` and `s0 = 4092` :
        - first L1 sets value in cache to `30`
        - L1 is write through, so L2 cache is also updated:
            - L2 is write back, so we update the corresponding dirty bit
            - memory is not updated
            - writing to L2 is done over the snoop bus
        - on the snoop bus, we send an invalidate method for address `4096`
            - the second L1 sets the valid bit to 0 for `4096`
    2. after running `sw t0, 8(s1)` on core 1, if `t0 = 60` and `s0 = 4088` :
        - L1 is write around, so we write into the L2
            - writing to L2 is done over the snoop bus
        - on the snoop bus, we send an invalidate method again for address `4096`
            - the first L1 sets the valid bit to 0 for `4096`
        - now, neither core has the correct address value in cache
            - only updated in the L2

Multicore and Multiprocessor Design

- the goal is to connect multiple computers to get higher performance:
  - go beyond the limits of parallelism in ILP:
    - instead use *process*-level parallelism
    - high throughput for independent jobs as well as for single programs running on multiple processors
  - *pros*:
    - scalability
    - availability
    - power efficiency
    - reliability
- in a **multiprocessor** system, there are separate packaged chips that can communicate via a bus through their pinouts, etc.
  - in a **multicore** system, there is a tighter integration onto the same chip so that multiple cores share parts of the same cache hierarchy
    - ie. closer communication with other cores through their caches or through a shared cache
- with multiprocessor design comes parallel programming:
  - pushes the responsibility of extracting parallelism to the programmer
  - difficulties:
    - partitioning
    - coordination
    - communications overhead
  - note that the sequential portion of the program can limit the speedup, as seen in the following example using Amdahl's law
  - ex. With 100 processors, can we achieve 90x speedup?

$$\begin{aligned} T_{new} &= \frac{T_{parallel}}{100} + T_{sequential} \\ 90 &= \frac{1}{1 - F_{parallel} + \frac{F_{parallel}}{100}} \\ F_{parallel} &= 0.999 \end{aligned}$$

    - only if the sequential part of the program takes up 0.1% of the original time
- ex. Calculating scaling speedup given the following workload, assuming the processors are load balanced:
  - sum of 10 scalars (sequential) and a 10×10 matrix sum (parallelizable):
    - with a single processor, $T = (10 + 100) \times t_{add} = 110 t_{add}$
    - with 10 processors:
      - $T = 10 \times t_{add} + \frac{100}{10} \times t_{add} = 20 t_{add}$

        - this gives a speedup of $\frac{110}{20} = 5.5$, or 55% of potential 10x improvement
    - with 100 processors:
        - $T = 10 \times t_{add} + \frac{100}{100} \times t_{add} = 11t_{add}$
        - this gives a speedup of $\frac{110}{11} = 10$, or 10% of potential 100x improvement
- alternatively, with a 100×100 matrix sum instead:
    - with a single processor, $T = (10 + 10000) \times t_{add} = 10010t_{add}$
    - with 10 processors:
        - $T = 10 \times t_{add} + \frac{10000}{10} \times t_{add} = 1010t_{add}$
        - this gives a speedup of $\frac{10010}{1010} = 9.9$, or 99% of potential 10x improvement
    - with 100 processors:
        - $T = 10 \times t_{add} + \frac{10000}{100} \times t_{add} = 110t_{add}$
        - this gives a speedup of $\frac{10010}{110} = 91$, or 91% of potential 100x improvement
- in **strong scaling**, the problem size is fixed across different numbers of processors
    - in **weak scaling**, the problem size is proportional to number of processors

Multiprocessors

- in a **shared memory multiprocessor**:
    - hardware provides a single physical address space for all processors
        - ie. all processors are connected to memory via an interconnection network
    - shared variables are synchronized using locks
    - memory access time may be **uniform (UMA)** or **nonuniform (NUMA)**
        - with nonuniform access time, load balancing becomes more difficult

Ex. Summing 100000 numbers on 100 processors with UMA:

```
// partitioning step:
sum[Pn] = 0; // each processor has ID: 0 ≤ Pn ≤ 99
for (i = 1000*Pn; i < 1000*(Pn+1); i++)
    sum[Pn] += A[i];

// reduction step:
half = 100;
while (half ≠ 1) {
    synch();
    if (half%2 != 0 && Pn == 0)
        sum[0] += sum[half-1]; // conditional sum when half is odd

    half /= 2;
    if (Pn < half)
        sum[Pn] += sum[Pn+half];
}
```

- to perform a parallel sum:
  - partition numbers per processor and perform an initial summation
  - then, in the reduction step, divide and conquer to add together the partial sums
    - need to synchronize between reduction steps
- in a **message passing** approach:
  - each processor has private physical address space
  - hardware sends and receives messages between processors
  - *pros*:
    - more scaling of memory capacity
      - no possible monolithic memory bottleneck
  - *cons*:
    - no more shared variables
  - to perform the same parallel summing of many numbers:
    - have to firstly distribute the partitions of the numbers to each computer
    - then after summing, have to send messages *while* some machines receive
      - eg. half send while half receive and add, etc.
- an example of message passing are loosely coupled **clusters**:
  - network of independent computers, each with private memory and OS
    - connected with I/O system eg. internet
  - suitable for applications with independent tasks eg. web servers, databases, etc.
  - *pros*:
    - high availability
    - scalable
    - affordable
  - *cons*:
    - administration cost
    - low interconnect bandwidth

Multithreading

- instead of running separate individual threads in as in multiprocessing, in **multithreading** we increase the number of threads that can run on a single processor:
  - requires replicating program state such as registers, PC, etc.
    - as well as fast switching between threads
  - note that this is different from the context switches done between time slices of traditional OS concurrency:
    - in OS, we are selecting between multiple threads for execution
    - in multithreading, utilizing a *faster* switch among threads *currently* in the processor to avoid expensive context switch overhead
      - ie. more than one thread executing at once on the same processor
  - may decrease cache performance with increased resource utilization, but may also increase overall processor utilization
- in **fine-grain multithreading**:
  - switch threads after each cycle
  - interleave instruction execution between multiple threads
  - if one thread stalls, others are executed to exploit **thread-level parallelism (TLP)**
- in **course-grain multithreading**:
  - only switch on long stalls eg. L2 cache miss
  - simplifies hardware, but does not hide short stalls eg. data hazards
  - eg. to avoid out-of-order execution, branch prediction, prefetching, etc. just switch threads
- in **simultaneous multithreading** AKA **hyperthreading**:
  - use a multiple-issue dynamically scheduled processor
  - schedule instructions from *multiple* threads in the same cycle
    - even more fine-grained
  - instructions from independent threads execute when function units are available
  - within threads, dependencies are handled by scheduling and register renaming
  - may have issues where one of the threads can slow down considerably compared to the others
- we can also classify based on the number of instruction streams and data streams:
  - a **single instruction single data (SISD)** machine is the sequential, default model
    - one set of instruction and one stream of data
  - a **multi instruction multiple data (MIMD)** machine is a multiprocessor

    machine
        * can run completely different programs or cooperating threads for a single program (SMPD)
    - in a **single instruction multiple data (SIMD)** machine, can perform the same instructions on multiple pieces of data
- SIMD operates elementwise on *vectors* of data:
    - all processors execute the *same* instruction at the same time
    - simplifies synchronization
    - reduced instruction control hardware
    - works best for highly data-parallel applications
- **vector processors** are an example of SIMD that have highly pipelined functional units:
    - stream data from or to vector registers to units
    - simplifies data-parallel programming
    - significantly reduces instruction-fetch bandwidth
        * absence of loop-carried dependencies
    - eg. in the vector extension for MIPS:
        * vectors are 32 64-bit registers
        * `lv, sv` load and store vectors
        * `addv.d` add vectors of double
        * `addvs.d` add scalar to each element of vector of double

Graphics Processing Units

- **graphics processing units (GPUs)** are specialized processors oriented to 3D graphics tasks
    - 3D graphics processing was originally only used in high-end computers
        * along with Moore's law, transistor density increased greatly, and 3D graphics processing became the norm
    - eg. vertex and pixel processing, shading, texture mapping, etc.
    - as a trend, CPU tends to be used more for sequential code, while GPU is used for parallel code
    - programming languages and APIs include DirectX, OpenGL, CUDA, etc.
- GPU architectures:
    - processing is highly data-parallel
    - GPUs are thus highly multithreaded
        * use thread switching to hide memory latency
    - graphics memory is wide and high-bandwidth, amenable to SIMD vectorization
    - generally made up of multiple **streaming multiprocessors (SM)** each composed of **streaming processors (SP)**

- streaming processors:
    - each is fine-grained multithreaded
    - a **warp** is Nvidia's group of 32 threads executed in parallel, SIMD style
        - eg. used in their Tesla GPUs
    - however GPUs do not fit exactly into a SIMD or MIMD model:
        - GPUs allow for *conditional* execution in a thread ie. not every thread in a warp affects register state
        - thus, similar to how a superscalar processor is a *dynamic* implementation of ILP when compared to static VLIW
            - the GPU multiprocessor is a *dynamic* implementation of data-level parallelism when compared to static SIMD or vector processors