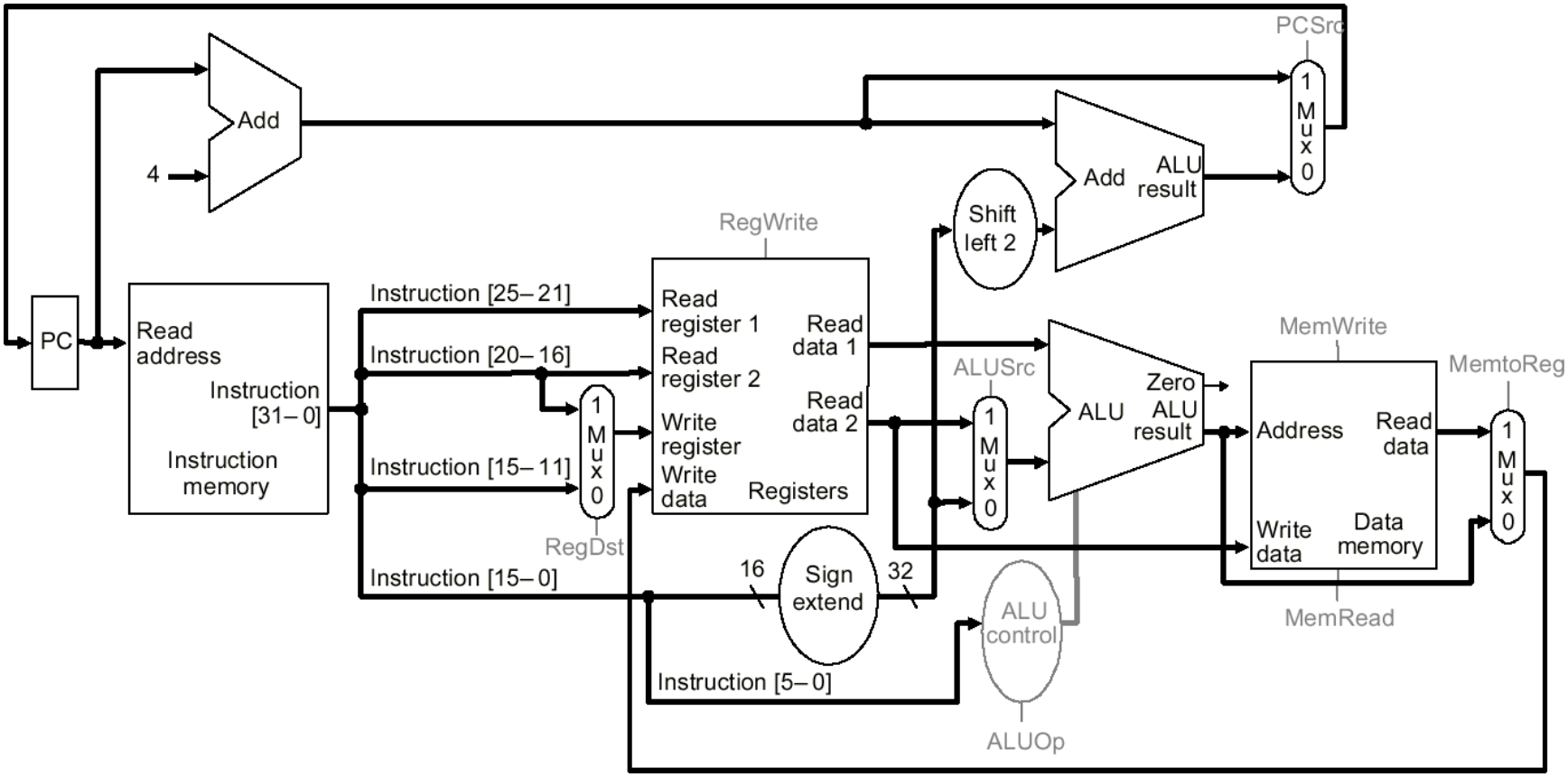
Overview of Processor Techniques

A brief look at CDA 3101
and CDA 5155

Single Cycle Machine

- What we learned over the course of this semester
- Each instruction is executed fully before the next instruction can start
- Requires the clk to be set to the slowest instruction
- Whilst executing instructions, each component will be unused for most of the execution and some may never be used.

Single Cycle Machine



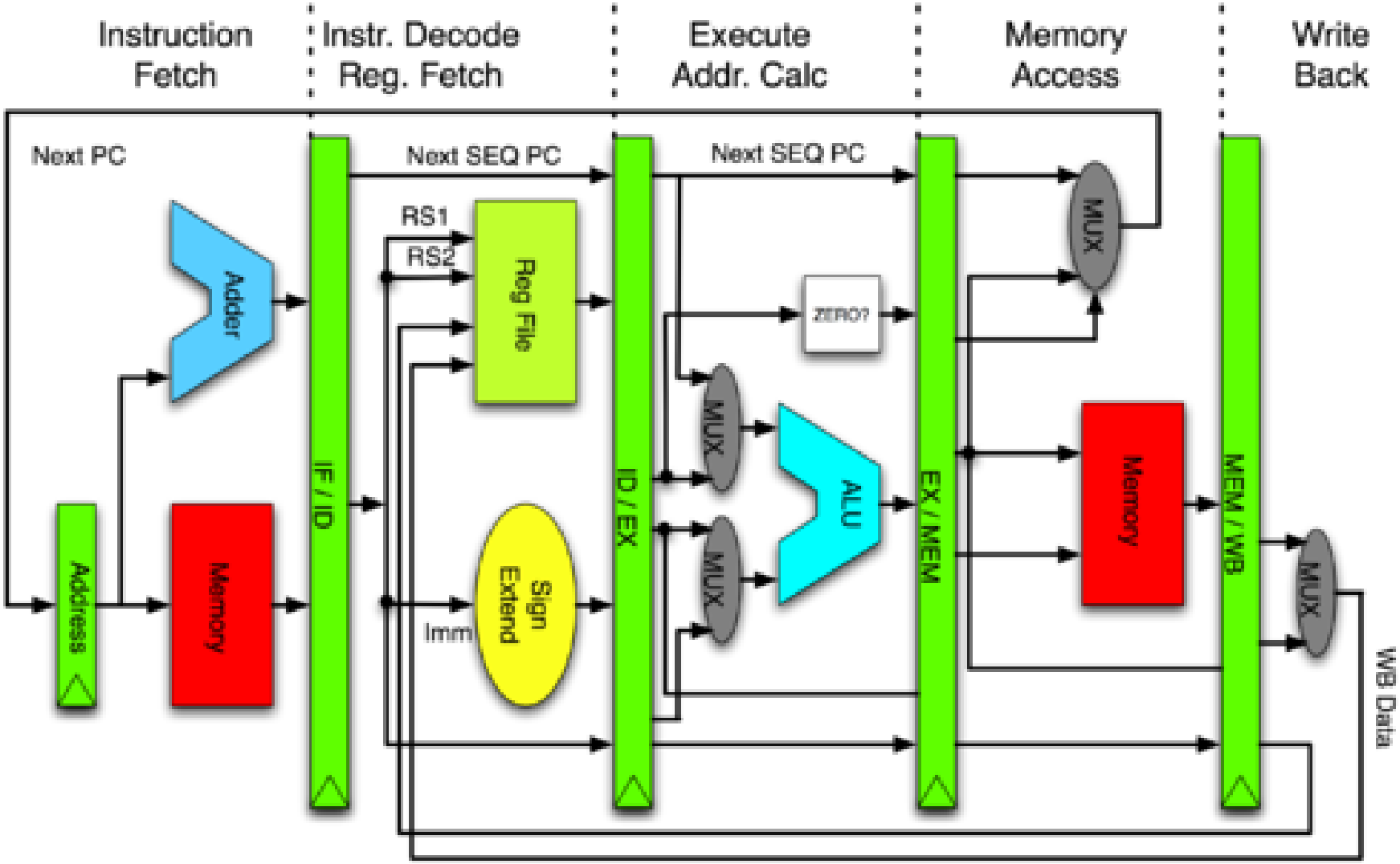
Multi Cycle Machine

- Breaks up the data path into stages and saves the results of each stage into special registers
- Typical stages:
  - IF (Instruction Fetch)
  - ID (Instruction Decode)
  - EX (Execute)
  - MEM (Memory Access)
  - WB (Write Back)
- Clk is set to be the longest stage
- Each instruction still executes to completion but now takes an amount of clk cycles equal to the number of stages
- Not really any faster than Single Cycle as you still have to wait for the instruction to finish; in fact, it's a little slower because now you have to save results to registers and the slow stages stall the faster ones
- Not ever really used; the ideas quickly turned into ….

Pipelining

- Found in most modern processors
- Takes the stages idea from multi cycle, but executes a different instruction within each stage
  - This, in theory, makes pipelining ‘number of stages’ faster than multi cycle or single cycle
  - In practice, however, various problems crop up

Pipelining


Instruction Fetch
Instr. Decode Reg. Fetch
Execute Addr. Calc
Memory Access
Write Back
Next PC
Next SEQ PC
Next SEQ PC
RS1
RS2
Adder
Reg File
IF / ID
ID / EX
EX / MEM
MEM / WB
MUX
ZERO?
ALU
Memory
Address
Memory
Sign Extend
Imm
WB Data

Pipelining

- Problems you have to worry about:
  - Data hazards: Later instructions reading from register/memory values not yet stored
  - Control hazards: Branch instructions might need to go to a different section of code, yet there will be several instructions in the pipe that are executing that shouldn't be there
- Solutions:
  - Data hazards:
    - Do nothing (not a solution)
    - Stall
    - Forward
  - Control hazards:
    - Do nothing (not a solution)
    - Flush
    - Predict (and Flush)

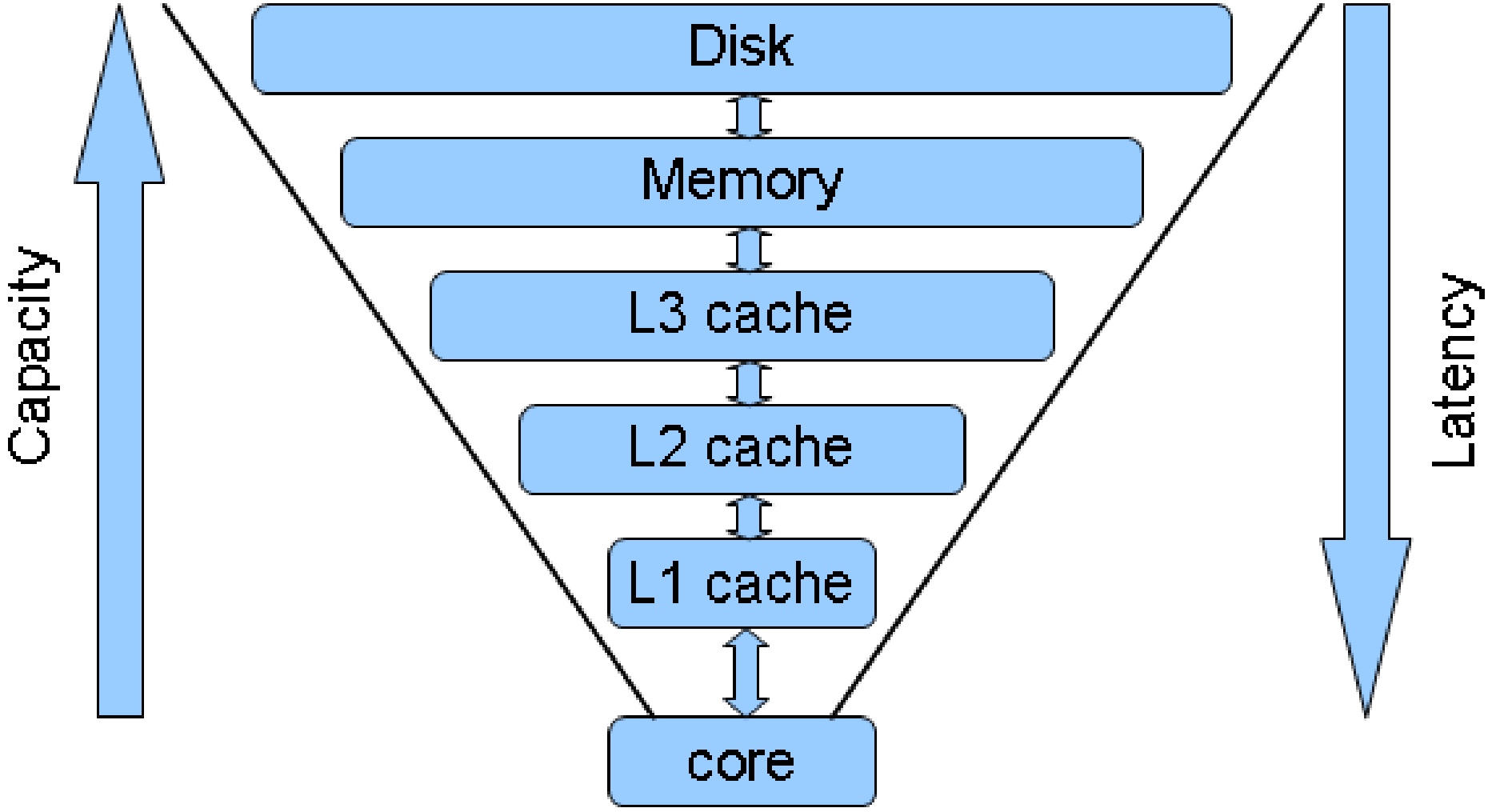
Branch Prediction

- Used to lower the amount of flushes the processor has to due in event of a branch
- The simplest is to always predict not taken and just fetch the next instruction which is what the processor does in the normal case
- More advanced implementations keep a state counter remembering what the result of the previous occurrence of the branch was
- The results are stored in a buffer mapping the pc of the branch to the value of taken/not taken and the offset to where the branch instruction branches to
- If you want to predict not taken, just fetch the next instruction like always
- If you want to predict take, branch to the offset stored within the branch buffer
- In the event of a mispredict, just flush, load the proper pc, and update the branch prediction buffer

Cache

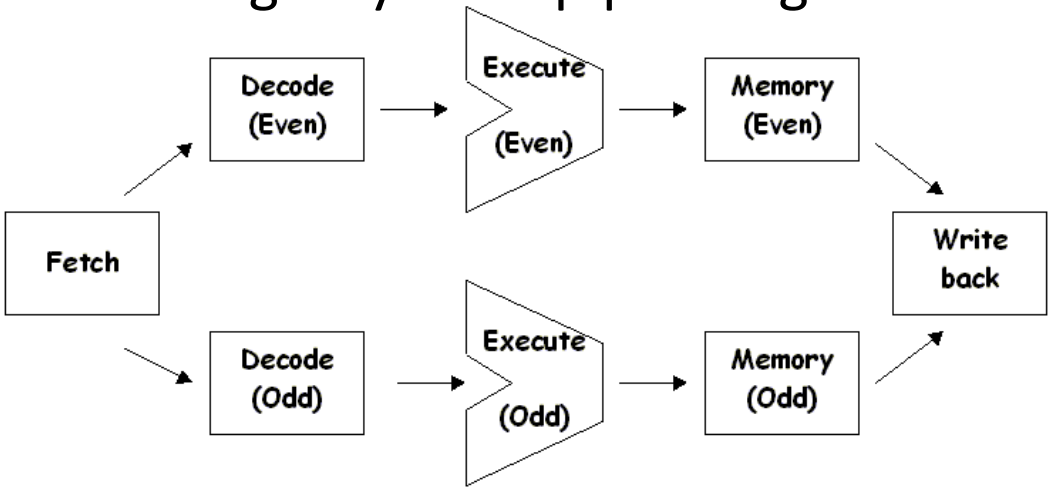
- Like the branch prediction buffer, but used to increase memory accesses speeds
- Stores a copy of a memory value within a smaller memory unit
- Since the unit is smaller (and closer) it takes less time to get the value allowing you to decrease the clk even further
- Typically, there are multiple layers (e.g. have a 4MB L2 cache and a 512 KB L1 cache), as well as specialized caches (e.g. instruction cache)
- Not perfect, if there are lots of misses it will be slower as you have to access L1 then L2 and then memory

Cache


Disk
Memory
L3 cache
L2 cache
L1 cache
core
Capacity
Latency

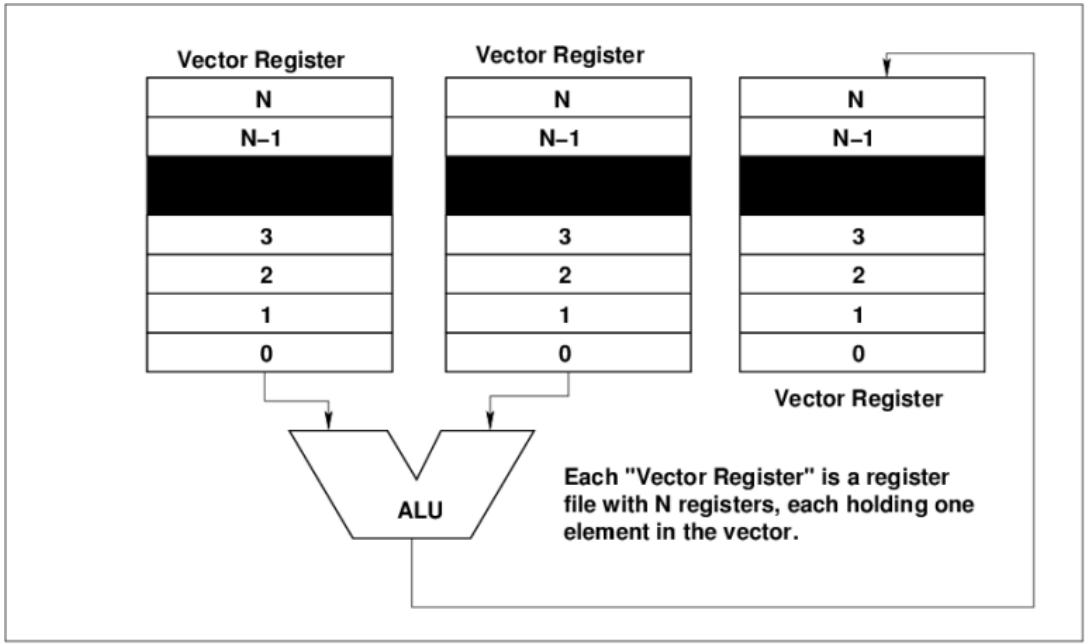
More Advanced Techniques

- Super scalar
  - Executes multiple instructions at a time by adding more components
  - Diminishing returns after 2 (that is the amount of hardware used and heat generated goes up faster than the speed increase)
  - Commonly, processors use 2, 4, or 8 way scaling
  - Add on to other designs like single cycle or pipelining



More Advanced Techniques

- Vector processors
  - Used in scientific computing
  - Executes on arrays of data (e.g. add one array with another and store the results in a third array)
  - Only refers to register storage and ALU processing, so it can be used with other designs too

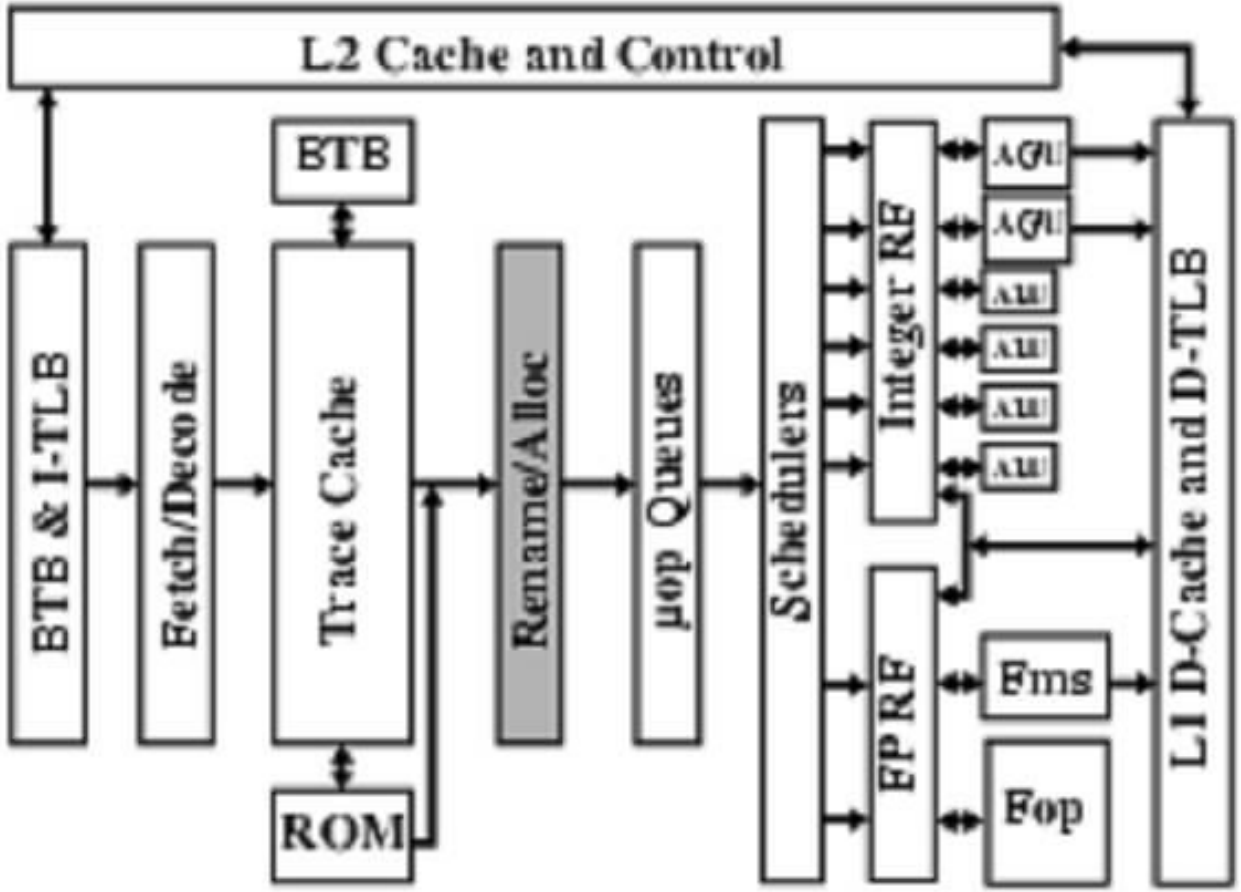


More Advanced Techniques

- Out of order execution (in order arrival / completion)
  - Very complicated and not used in processors that care about power consumption / heat
  - Groups instructions into certain types (e.g. add group, memory group)
  - Places instructions into wait queues until the operands have been calculated
  - Executes ready instructions in parallel (adds more dedicated hardware)
  - Places the results in a larger queue where the instructions come in at
  - When the lead instruction has fully executed it is removed and another instruction is fetched

More Advanced Techniques

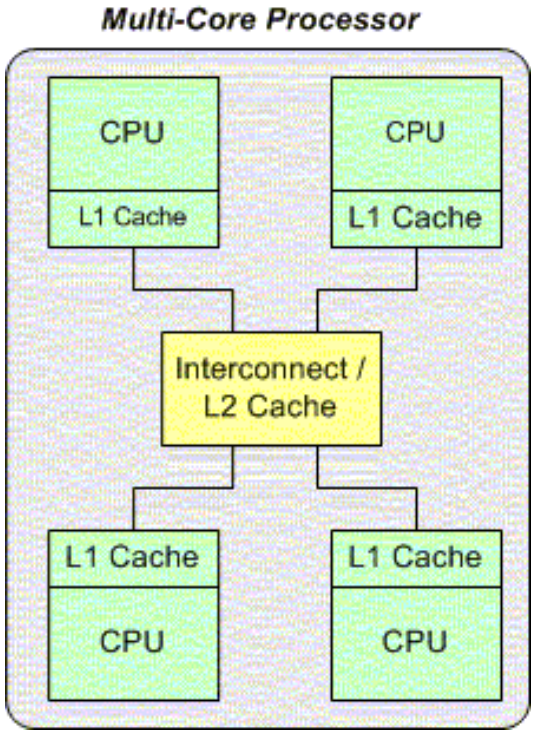
- Out of order execution (in order arrival / completion)



(e) Allocate; Register renaming

More Advanced Techniques

- Multiprocessing / multicore
  - Adds small, simple, independent processors within a single physical processor
  - Allows communication through specialized networks and through the larger cache
  - Each processor also has it's own smaller cache levels



Other Processors

Other Processors

- Having learnt MIPS, we can learn other major processors.
- This is only going to be a cursory glance at some interesting differences of the major processors compared to MIPS

ARM

- **Advanced RISC Machine**
- The major processor for mobile and embedded electronics, such as phones and tablets
- Features are: simple design, low power, low cost

ARM

- One of the most interesting features is the **conditional execution**.
- That is, an instruction will execute if some condition is true, otherwise it will not do anything (turned into a nop).

ARM

- A set of flags, showing the relation of two numbers : gt, equal, lt.
  - cmp Ri, Rj # set the flags depending on the values in Ri and Rj
  - subgt Ri, Ri, Rj # i = i – j if flag is gt
  - sublt Ri, Ri, Rj # i = i – j if flag is lt
  - bne Label # goto Label if flag is not equal

ARM

- How to implement

```
while (i != j) {
  if (i > j)
        i -= j;
  else
        j -= i;
}
```

ARM

- In MIPS, assume i is in $s0, j in $s1:

```
Loop:  beq $s0, $s1, Done
       slt $t0, $s0, $s1
       beq $t0, $0, L1
       sub $s1, $s1, $s0
       j Loop
L1:    sub $s0, $s0, $s1
L2:    j Loop
```

ARM

- In ARM,

```
Loop: cmp Ri, Rj
      subgt Ri, Ri, Rj
      sublt Rj, Rj, Ri
      bne Loop
```

ARM

- Discussion: Given the MIPS hardware setup, can we support conditional execution?

ARM

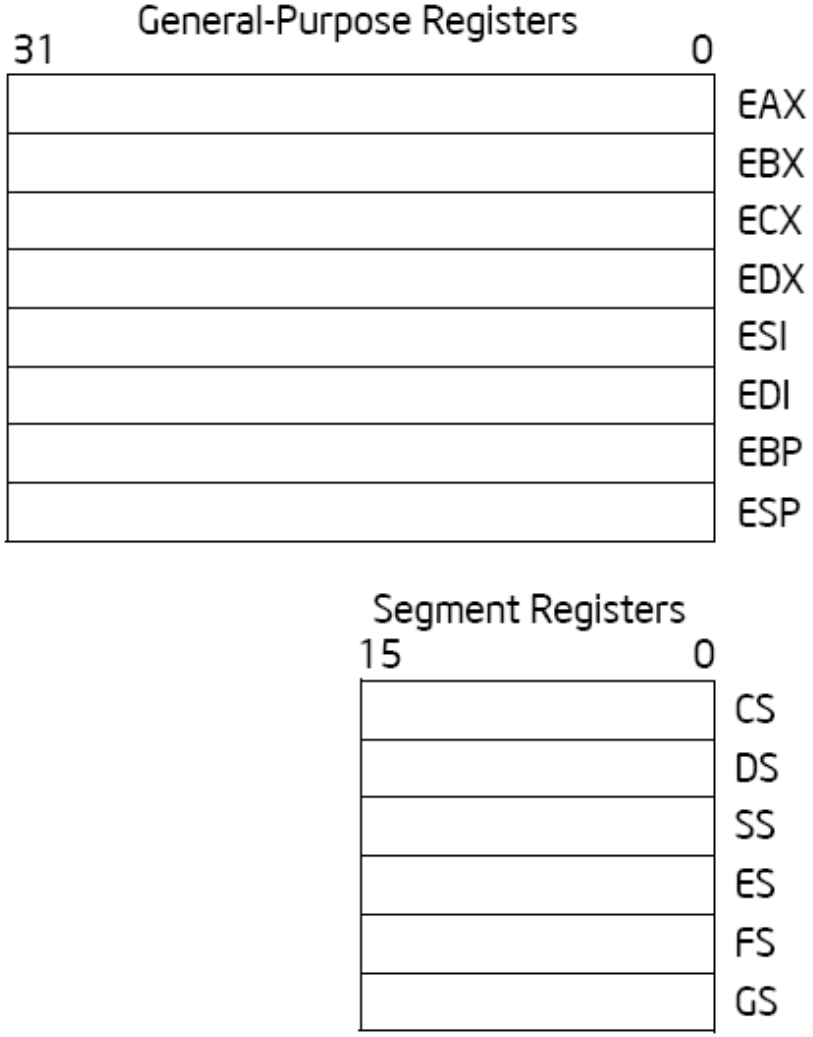
- Another interesting feature is its ability to fold shift instructions into data processing instructions (add, sub, move, etc)
- In C:
  - a += (j << 2);
- In MIPS (assuming a in $s0 and j in $s1):
  - sll $t0, $s1, 2
  - add $s0, $s0, $t0
- In ARM:
  - ADD Ra, Ra, Rj, LSL #2

Intel Processor

Basic Program Execution Registers

- General purpose registers
  - There are eight registers (note that they are not quite general purpose as some instructions assume certain registers)
- Segment registers
  - They define up to six segment selectors
- EIP register – Effective instruction pointer
- EFLAGS – Program status and control register

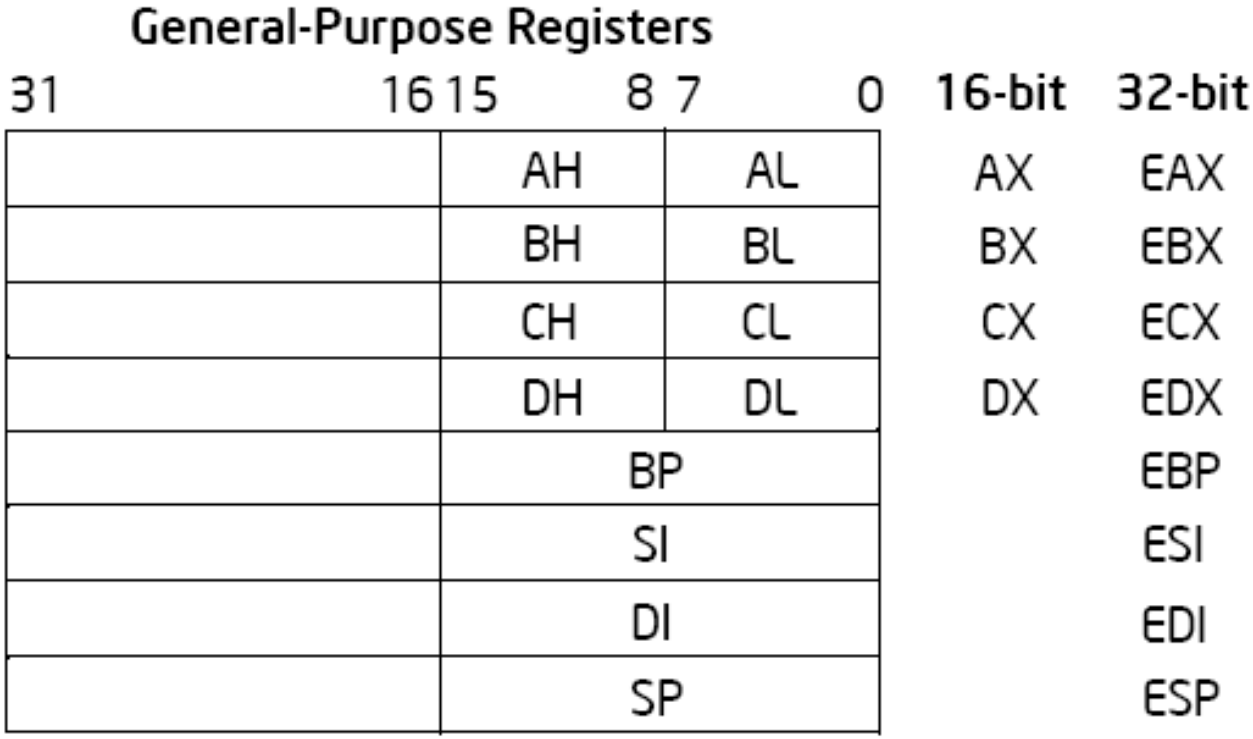
General Purpose and Segment Registers



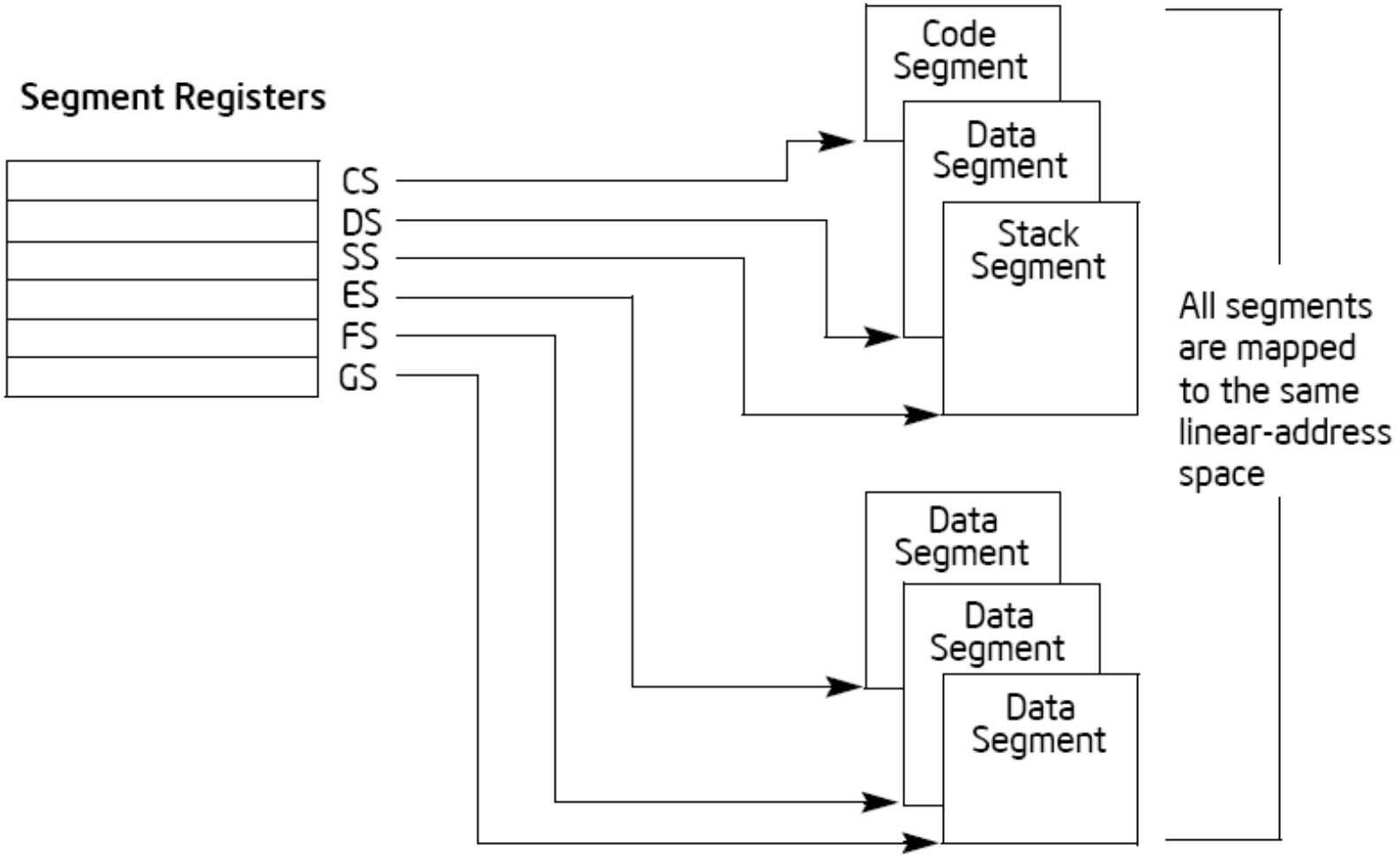
General Purpose Registers

- **EAX** — Accumulator for operands and results data
- **EBX** — Pointer to data in the DS segment
- **ECX** — Counter for string and loop operations
- **EDX** — I/O pointer
- **ESI** — Pointer to data in the segment pointed to by the DS register; source pointer for string operations
- **EDI** — Pointer to data (or destination) in the segment pointed to by the ES register; destination pointer for string operations
- **ESP** — Stack pointer (in the SS segment)
- **EBP** — Pointer to data on the stack (in the SS segment)

Alternative General Purpose Register Names

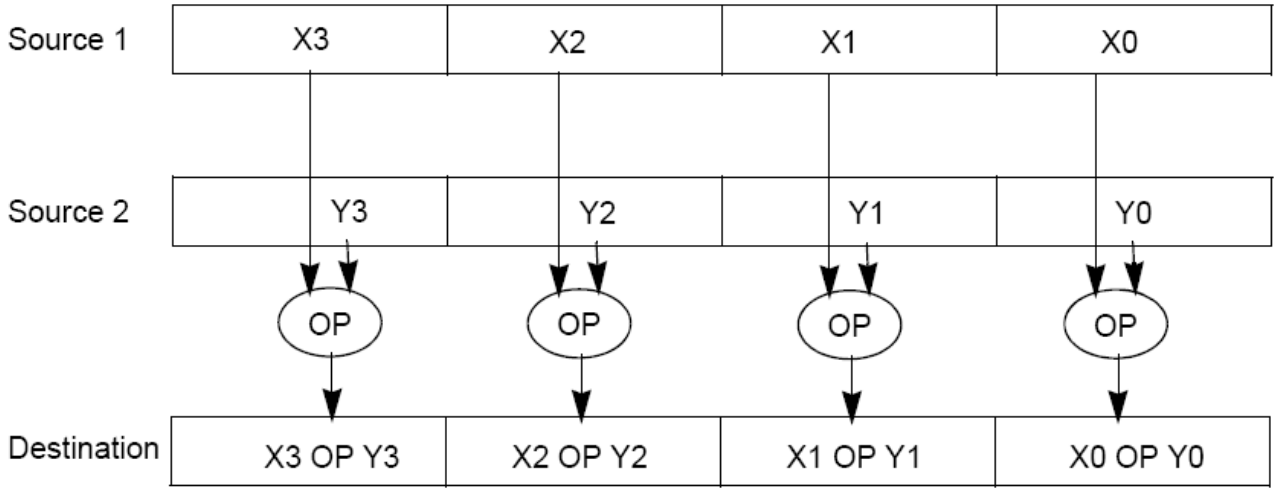


Segment Registers



SIMD

- To improve performance, Intel adopted SIMD (single instruction multiple data) instructions.
- Streaming SIMD Extensions (SSE) introduced eight 128-bit data registers (called XMM registers)
  - In 64-bit modes, they are available as 16 64-bit registers
  - The 128-bit packed single-precision floating-point data type, which allows four single-precision operations to be performed simultaneously



GCC Inline Assembly

- GCC inline assembly allows us to insert inline functions written in assembly
  - http://www.ibiblio.org/gferg/ldp/GCC-Inline-Assembly-HOWTO.html
  - GCC provides the utility to specify input and output operands as C variables
  - Basic inline

```
asm("movl %ecx %eax"); /* moves the contents of ecx to eax */
__asm__("movb %bh (%eax)"); /*moves the byte from bh
                              to the memory pointed by eax */
```

  - Extended inline assembly

```
asm ( assembler template
    : output operands                   /* optional */
    : input operands                    /* optional */
    : list of clobbered registers       /* optional */
    );
```

GCC Inline Assembly

- A simple example:

```
int main(void)
{
        int foo = 10, bar = 15;
        __asm__ __volatile__("addl  %%ebx,%%eax"
                             :"=a"(foo)
                             :"a"(foo), "b"(bar)
                             );
        printf("foo+bar=%d\n", foo);
        return 0;
}
```

Using SSE

```
#define mulfvec4_SSE(a, b, c) \
{ \
__asm__ __volatile__ ("movups %1, %%xmm0 \n\t" \
                      "movups %2, %%xmm1 \n\t" \
                      "mulps %%xmm0, %%xmm1 \n\t" \
                      "movups %%xmm1, %0 \n\t" \
                      :"=m" (c) \
                      :"m" (a), \
                      "m" (b)); \
}
```

BCM4306 Wireless Card Processor

A Part of the Code from http://www.ing.unibs.it/~openfwwf/

```
// ************************************************************************************************
// HANDLER:     state_machine_start
// PURPOSE:     Checks conditions looking for something to do. If there is no coming job firmware sleeps for a while or suspends device.
//
        state_machine_idle:;
                mov     0, WATCHDOG
                jnzx    0, 3, GLOBAL_FLAGS_REG3, 0x000, state_machine_start;    /* This bit was set and reset in bg_noise_sample */
                jnzx    0, 9, [SHM_HF_MI], 0x000, state_machine_start;
                mov     0xFFFF, SPR_MAC_MAX_NAP;                                /* Sleep for a while.. */
                nap;

        state_machine_start:;
                jnext   EOI(COND_RADAR), no_radar_workaround;
                jzx     0, 13, [SHM_HF_LO], 0x000, no_radar_workaround;         /* if (!(shm_host_flags_1 & MHF_RADARWAR)) */
                mov     0x00C8, GP_REG5;                                        /* GP_REG5 = APHY_RADAR_THRESH1 */
                or      [SHM_RADAR], 0x000, GP_REG6;                            /* write [SHM_RADAR] into GP_REG5 */
                call    lr0, write_phy_reg;
        no_radar_workaround:;
                extcond_eoi_only(COND_PHY0);
                extcond_eoi_only(COND_PHY1);
                orx     1, 3, 0x000, GLOBAL_FLAGS_REG2, GLOBAL_FLAGS_REG2;      /* clear bits 0x18 */
                jzx     0, 3, SPR_IFS_STAT, 0x000, check_mac_status;            /* if (!(SPR_IFS_STAT & 0x08)) */
                orx     1, 1, 0x000, GLOBAL_FLAGS_REG2, GLOBAL_FLAGS_REG2;      /* GLOBAL_FLAGS_REG2 & ~AFTERBURNER_TX|AFTERBURNER_RX */
                or      [SHM_GCLASSCTL], 0x000, GP_REG6;
                call    lr1, gphy_classify_control_with_arg;                    /* Classify control from SHM to PHY */
        check_mac_status:;
                jnext   COND_MACEN, mac_suspend_check;                          /* Check if we can sleep */
                jext    COND_TX_FLUSH, check_conditions;
```