
High Performance Computing

Lecture 10

Matthew Jacob

Indian Institute of Science

We will assume that ...

1. Activity is overlapped in time where possible
 - PC increment and instruction fetch from memory?
 - Instruction decode and effective address calculation
2. Load-store ISA: the only instructions that take operands from memory are loads & stores
3. Main memory delays are not typically seen by the processor
 - Otherwise the timeline is dominated by them
 - There is some hardware mechanism through which most memory access requests can be satisfied at processor speeds ([cache memory](#))

Term: Processor Cycle Time

- Unit of timescale of processor; time required to do a basic operation

Steps in Instruction Execution

- Fetch instruction from memory to processor
 - $IR = \text{Memory}[PC]$ cache memory access
 - Increment PC (simple) ALU operation
- Decode instruction and get its operands
 - Decode (simple) logic circuitry
 - Get Operands from registers register access
- Execute the operation
 - Trigger appropriate functional hardware ALU operation
 - Load/store: get data from main memory cache access
- Write back the result
 - Write result to destination register register access

Term: Processor Cycle Time

- Unit of timescale of processor; time required to do a basic operation
 - Cache memory access
 - Register access + some logic (like decode)
 - ALU operation

Steps in Instruction Execution

Fetch instruction from memory to processor

- IR = Memory[PC]; Increment PC 1 cycle

Decode instruction and get its operands

- Decode; Get Operands from registers 1 cycle

Execute the operation

- Trigger appropriate functional hardware 1 cycle
- Load/store: get data from main memory 1 cycle

Write back the result

- Write result to destination register 1 cycle

Steps in Execution: JR R6

Fetch instruction from memory to processor

- IR = Memory[PC]; Increment PC 1 cycle

Decode instruction and get its operands

- Decode; Get Operands from registers 1 cycle

Execute the operation

- Trigger appropriate functional hardware 1 cycle
- Load/store: get data from main memory 1 cycle

Write back the result

- Write result to destination register 1 cycle

Steps in Execution: ADD R1, R2, R3

Fetch instruction from memory to processor

- IR = Memory[PC]; Increment PC 1 cycle

Decode instruction and get its operands

- Decode; Get Operands from registers 1 cycle

Execute the operation

- Trigger appropriate functional hardware 1 cycle
- Load/store: get data from main memory 1 cycle

Write back the result

- Write result to destination register 1 cycle

Steps in Execution: LW R1, -8(R29)

Fetch instruction from memory to processor

- IR = Memory[PC]; Increment PC 1 cycle

Decode instruction and get its operands

- Decode; Get Operands from registers 1 cycle

Execute the operation

- Trigger appropriate functional hardware 1 cycle
- Load/store: get data from main memory 1 cycle

Write back the result

- Write result to destination register 1 cycle

Term: Processor Cycle Time

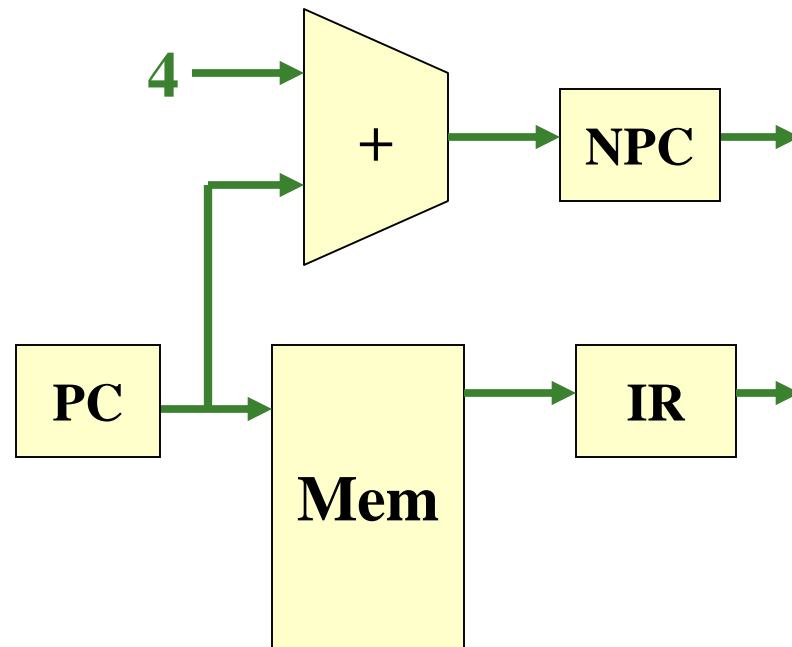
- Unit of timescale of processor; time required to do a basic operation
 - Cache memory access
 - Register access + some logic (like decode)
 - ALU operation
- A MIPS 1 instruction can be processed in 3-5 cycles
 - Jump: IFetch, Decode/OpFetch, DoOp (3)
 - ALU: IFetch, Decode/OpFetch, DoOp, WriteReg (4)
 - Load: IFetch, Decode, EffAddr, Cache, WriteReg (5)
- Addressing modes: (R) vs d(R)

Instruction Execution

Instruction Fetch (IF)

from program memory
to instruction register

$IR \leftarrow Mem[PC]$
Increment PC

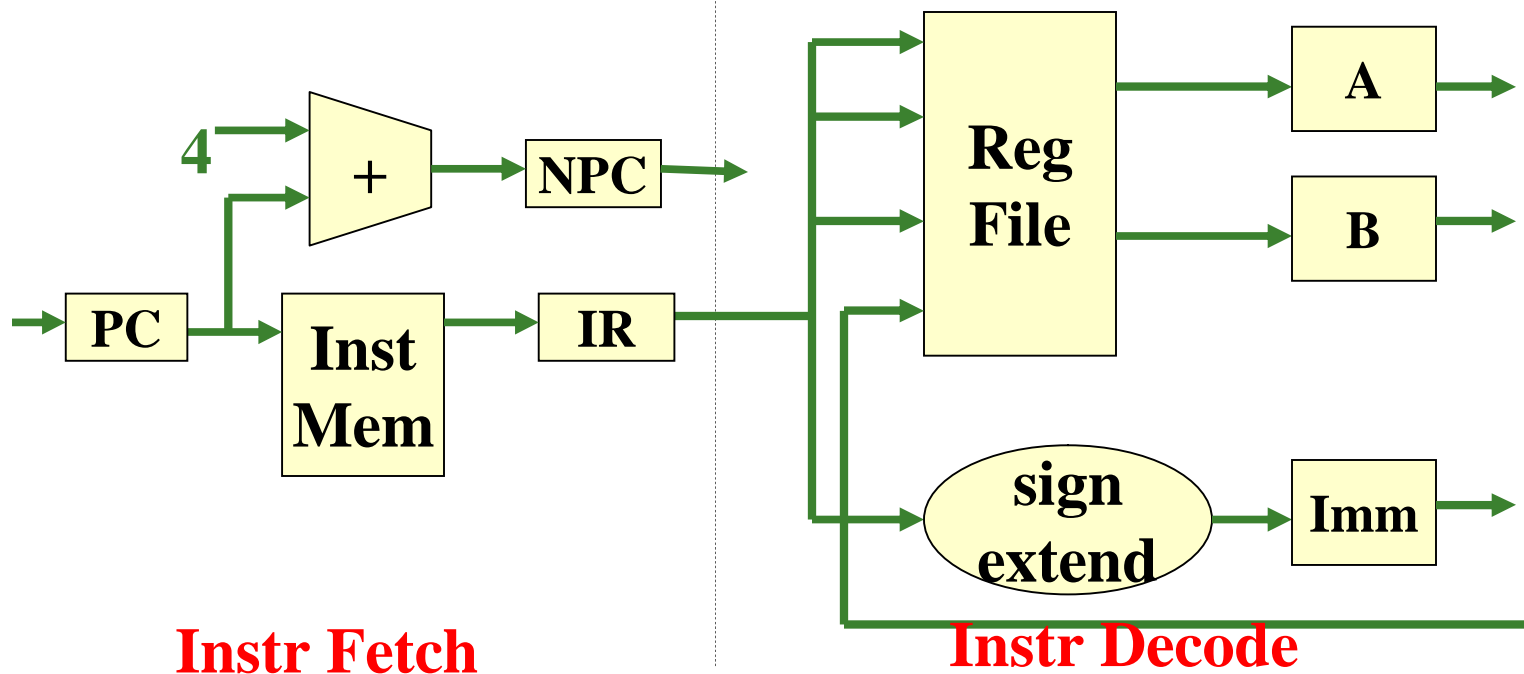


Instr Fetch

Instruction Execution.

Instruction Decode & Operand Fetch (ID)

$A \leftarrow \text{RegisterFile}[IR_{rs}]$
 $B \leftarrow \text{RegisterFile}[IR_{rt}]$
 $\text{Imm} \leftarrow \text{sign extend}(IR_{15-0})$



Instruction Execution..

Execution (EX)

Arithmetic Inst:

ALU-Out \leftarrow A op B

ALU-Out \leftarrow A op Imm

Load/Store Inst:

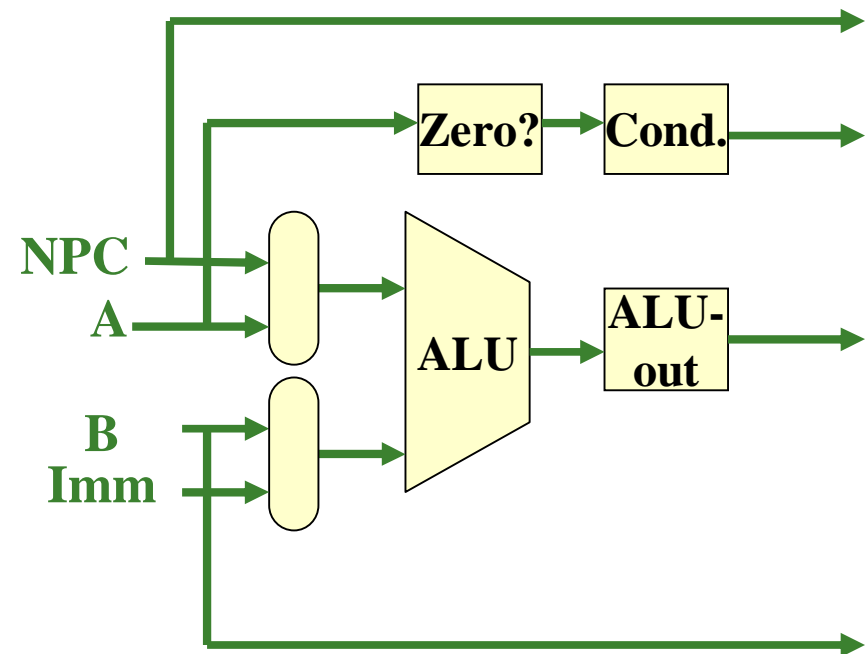
ALU-Out \leftarrow A + Imm

Branch Inst:

ALU-Out \leftarrow NPC + Imm

Jump Inst:

PC \leftarrow NPC₃₁₋₂₈ || IR₂₅₋₀ || 00



Execution

Instruction Execution...

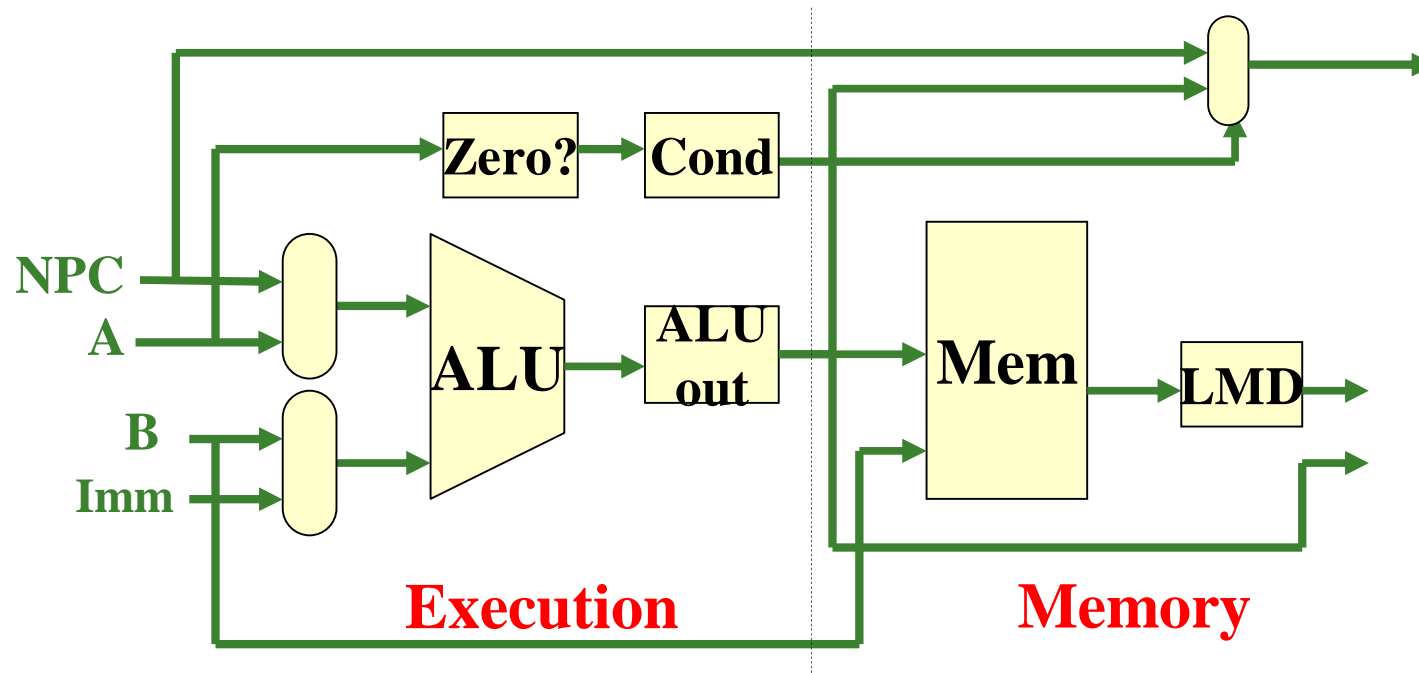
Memory (MEM)

Load Instr

$LMD \leftarrow Mem[ALUout]$

Store Instr

$Mem[ALUout] \leftarrow B$



Instruction Execution....

Write Back (WB)

ALU Inst

RegisterFile[rd] \leftarrow ALUout

Load Inst

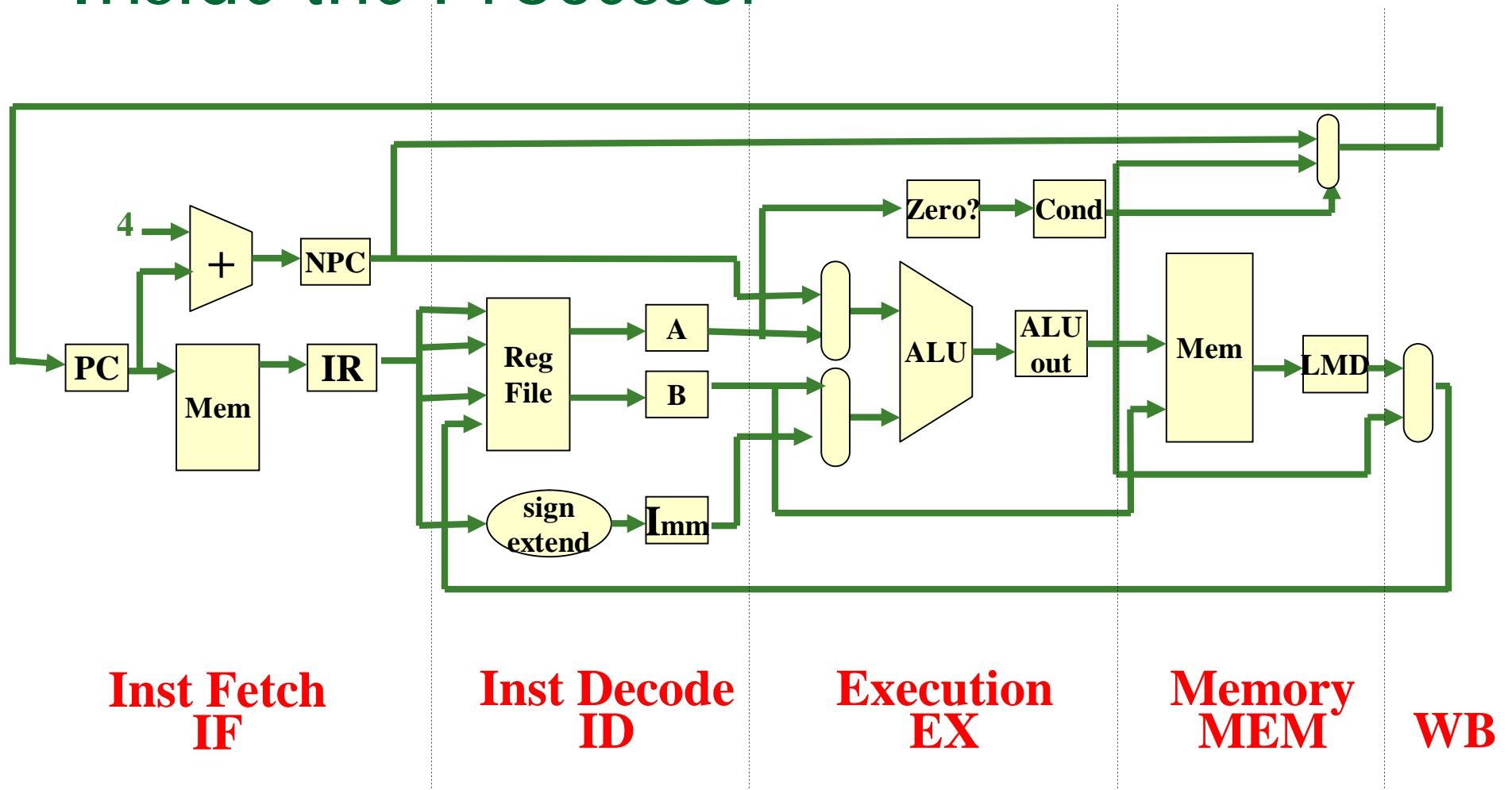
RegisterFile[rt] \leftarrow LMD

Conditional Branch Inst

PC \leftarrow ALU-out if Cond

PC \leftarrow NPC otherwise

Inside the Processor

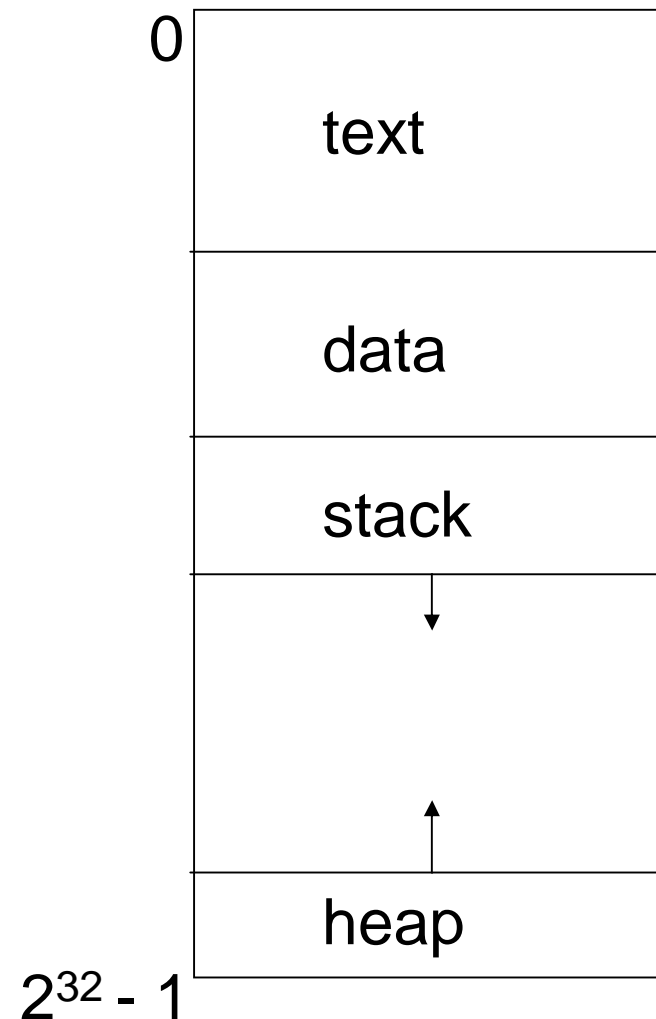


Reality Check

- Problem: There could be many programs running on a machine concurrently
- Sharing the resources of the computer
 - Processor time
 - Main memory
- They must be protected from each other
 - One program should not be able to access the variables of another
 - This is typically done through [Address Translation](#)

Use of Main Memory by a Program

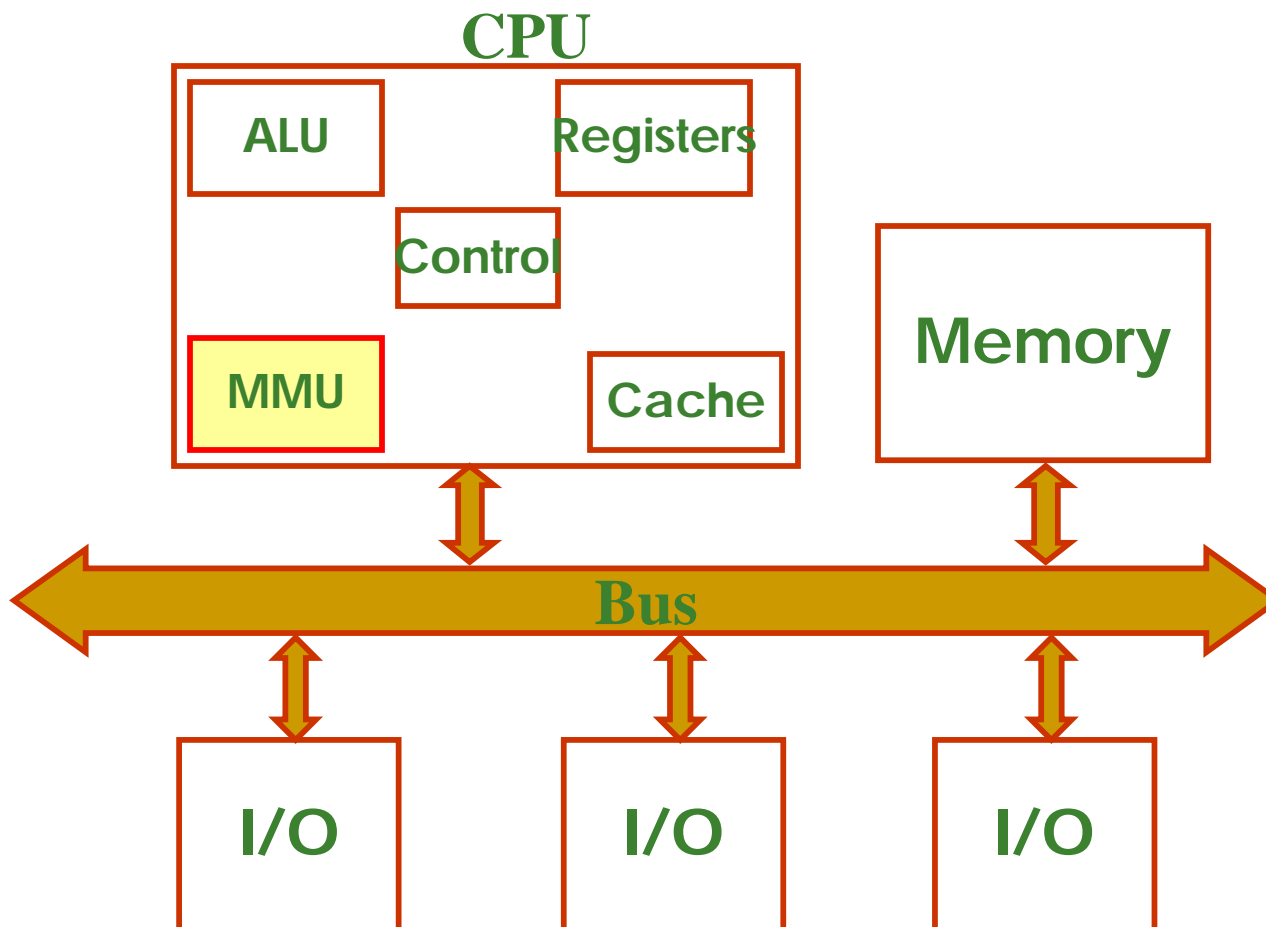
- Instructions (code, text)
- Data
 - Statically allocated
 - Stack allocated
 - Heap allocated



Idea of Address Translation

- Each program is compiled to use addresses in the range 0 .. MaxAddress (e.g., 0 .. $2^{32} - 1$)
- These addresses are not real, but only Virtual Addresses
- They have to be translated into actual main memory addresses
- The translation can be done to ensure that one program can not access variables of another program
- Many programs in execution can then safely share main memory
- Terminology: **virtual address**, **physical address**
- **Memory Management Unit (MMU)**: The hardware that does the address translation

Recall: Basic Computer Organization



Agenda

1. **Program execution:** Compilation, Object files, Function call and return, Address space, Data & its representation (4)
2. **Computer organization:** Memory, Registers, Instruction set architecture, Instruction processing (6)
3. **Virtual memory:** Address translation, Paging (4)
4. **Operating system:** Processes, System calls, Process management (6)
5. **Pipelined processors:** Structural, data and control hazards, impact on programming (4)
6. **Cache memory:** Organization, impact on programming (5)
7. **Program profiling** (2)
8. **File systems:** Disk management, Name management, Protection (4)
9. **Parallel programming:** Inter-process communication, Synchronization, Mutual exclusion, Parallel architecture, Programming with message passing using MPI (5)