Lecture Notes of Computer Architecture

(Lecturer A-Lei Liang)

Zhou Fan
ACM Class, Shanghai Jiao Tong University

Contents

1 Pipelining

1.1 Introduction of Pipelining

Pipelining is an implementation technique whereby multiple instructions are overlapped in execution; it takes advantage of parallelism that exists among the actions needed to execute an instruction.[1]

1.1.1 Laundry Example

Suppose we have many loads of clothes to wash, dry and fold.

- Each step (washing, drying and folding) is called a *pipe stage* or a *pipe segment*.
- *Latency* is the total time spent on a single task, which is not improved by pipelining.Unbalanced lengths of pipe stages reduces speedup.
- *Throughput* is defined as the number of loads of clothes per minute. It shows how often a load of clothes exits the pipeline.
- The time required between moving an instruction one step down the pipeline is a *processor cycle*. In a computer, this processor cycle is usually 1 clock cycle.[1]

1.1.2 Speedup from Pipelining

Throughput is what matters. Pipelining helps *throughput* of whole workload, while it doesn't help *latency* of single task.

- Unbalanced lengths of pipe stages reduces speedup.
- Handover time between pipe stages reduces speedup.

To improve the efficiency of a pipeline, one should balance the length of each pipeline stage. If the stages are perfectly balanced, then the time per instruction on the pipeline processor is equal to (under ideal conditions)

$$\frac{\text{Time per instruction on unpipelined machine}}{\text{Number of pipe stages}}$$

and the throughput of the pipeline is equal to

$$\text{Number of pipe stages} \times \text{Throughput on unpipelined machine}$$

1.2 The Basics of a RISC Instruction Set

1.2.1 Introduction of RISC

A RISC[1] is a computer whose instruction set architecture has lower *cycles per instruction* (CPI) than a CISC[2].

MIPS is a RISC instruction set architecture.

[1] reduced instruction set computer

[2] complex instruction set computer

Key Properties of RISC Architectures[1]

- All operations on data apply to data in registers and typically change the entire register (32 or 64 bits per register).
- Only load and store operations can affect memory. Load and store operations that load or store less than a full register are often available.
- The instruction formats are few in number, with all instructions typically being one size.

These properties make the implementation of pipelining simple.
Most RISC architectures like MIPS have three classes of instructions:

1. ALU[3] instructions
2. Load and store instructions
3. Branches and jumps

1.2.2 Implementation of a RISC Instruction Set

The implementation here will focus only on a pipeline for an integer subsetof a RISC architecture that consists of load-store word, branch, and integer ALU operations.

Implementation Without Pipelining[1] Every instruction in the RISC subset can be implemented in at most 5 clock cycles as follows:

1. *Instruction fetch cycle* (IF):

   Send the program counter (PC) to memory and fetch the current instruction from memory. Update the PC to the next sequential PC by adding 4 (since each instruction is 4 bytes) to the PC[4].

2. *Instruction decode / register fetch cycle* (ID):

   Decode the instruction and read the registers corresponding to register source specifiers from the register file. Do the equality test on the registers as they are read, for a possible branch. Sign-extend (introduced later) the offset field of the instruction in case it is needed. Compute the possible branch target address by adding the sign-extended offset to the incremented PC.

   Decoding is done in parallel with reading registers, which is possible because the register specifiers are at a fixed location in a RISC instruction. This technique is known as *fixed-field decoding.* It may read a register that we don't use, and it doesn't help but also doesn't hurt performance.

3. *Execution / effective address cycle* (EX):

   The ALU operates on the operands prepared in the prior cycle, performing one of three functions depending on the instruction type.

[3] Arithmetic logic unit

[4] program counter, indicates where a computer is in its program sequence.

   - Memory reference — The ALU adds the base register and the offset to form the effective address.
   - Register-Register ALU instruction — The ALU performs the operation specified by the ALU opcode on the values read from the register file.
   - Register-Immediate ALU instruction — The ALU performs the operation specified by the ALU opcode on the first value read from the register file and the sign-extended immediate.

   In a load-store architecture the effective address and execution cycles can be combined into a single clock cycle, since no instruction needs to do both of them.

4. *Memory Access* (MEM):

   The memory does a read using the effective address computed in the previous cycle or writes the data read from the register file using the effective address, if the instruction is a load or a store.

5. *Write-back cycle* (WB):

   For a Register-Register ALU instruction or a load instruction, write the result into the register file.

Sign Extension In computer arithmetic, sigh extension is the operation of increasing the number of bits of a binary number while preserving the number's sign and value. This is done by appending digits (same as the sign bit) to the most significant side of the number, following a procedure dependent on the particular signed number representation used.

For example:

- "00 1010" (decimal positive 10) → "0000 0000 0000 1010"
- "11 1111 0001" (decimal negative 15) → "1111 1111 1111 0001"

Pipelined Implementation[1] Under ideal conditions, we can easily get a pipelined implementation from the execution described above, doing one of the five steps for five instructions in parallel in each pipeline stage. This procedure is showed in the following table.

	Clock Number								
Instruction Number	1	2	3	4	5	6	7	8	9
Instruction i	IF	ID	EX	MEM	WB				
Instruction $i+1$		IF	ID	EX	MEM	WB			
Instruction $i+2$			IF	ID	EX	MEM	WB		
Instruction $i+3$				IF	ID	EX	MEM	WB	
Instruction $i+4$					IF	ID	EX	MEM	WB

1.3 Pipeline Hazards

1.3.1 Three Classes of Hazards

Hazards prevent the next instruction in the instruction stream from executing during its designated clock cycle. There are three classes of hazards[1]:

1. *Structural hazards* arise from resource conflicts when the hardware cannot support all possible combinations of instructions simultaneously in overlapped execution.
2. *Data hazards* arise when an instruction depends on the results of a previous instrucion still in the pipeline.
3. *Control hazards* is caused by delay between the fetching of instruction and decisions about changes in control flow (branches and jumps).

These hazards can make it necessary to *stall* the pipeline.
There are three generic data hazards[2]:

- Read After Write (RAW):
  $Instr_J$ tries to read oprand before $Instr_I$ writes it.
  - I: add **r1**, r2, r3
  - J: sub r4, **r1**, r3
- Write After Read (WAR):
  $Instr_J$ writes oprand before $Instr_I$ reads it.
  - I: sub r4, **r1**, r3
  - J: add **r1**, r2, r3
  - K: mul r6, r1, r7

  This cannot happen in MIPS 5 stage pipeline.
- Write After Write (WAW):
  $Instr_J$ writes oprand before $Instr_I$ writes it.
  - I: sub **r1**, r4, r3
  - J: add **r1**, r2, r3
  - K: mul r6, r1, r7

  This cannot happen in MIPS 5 stage pipeline.

1.3.2 Solutions to Hazards

Forwarding Some case of data hazards can be avoided using a technique called *forwarding*. Forwarding can be generalized to include passing a result directly to the functional unit that requires it: A result is forwarded from the pipeline register corresponding to the output of one unit to the input of another, or from the result of a unit to the input of the same unit.[1]
However, some data hazards still exists even with forwarding.

Software Scheduling Using software to rearrange instructions is a way to avoid load hazards. In the following example, rearrange instruction order, increase the distance between $Instr_I$ and $Instr_J$ to avoid the hazard:

- I: lw **r1**, addr
- J: add r3, r2, **r1**

Control Hazards Alternatives for control hazard[1]:

- Stall until branch direction is clear
- Predict Branch Not Taken / Branch Taken
- Static Branch Prediction: use profile information collected from ealier runsm, because an individual branch is often highly biased toward taken or untaken.
- Dynamic Branch Prediction: predict branches dynamically base on program behaviour.

2 Placeholder

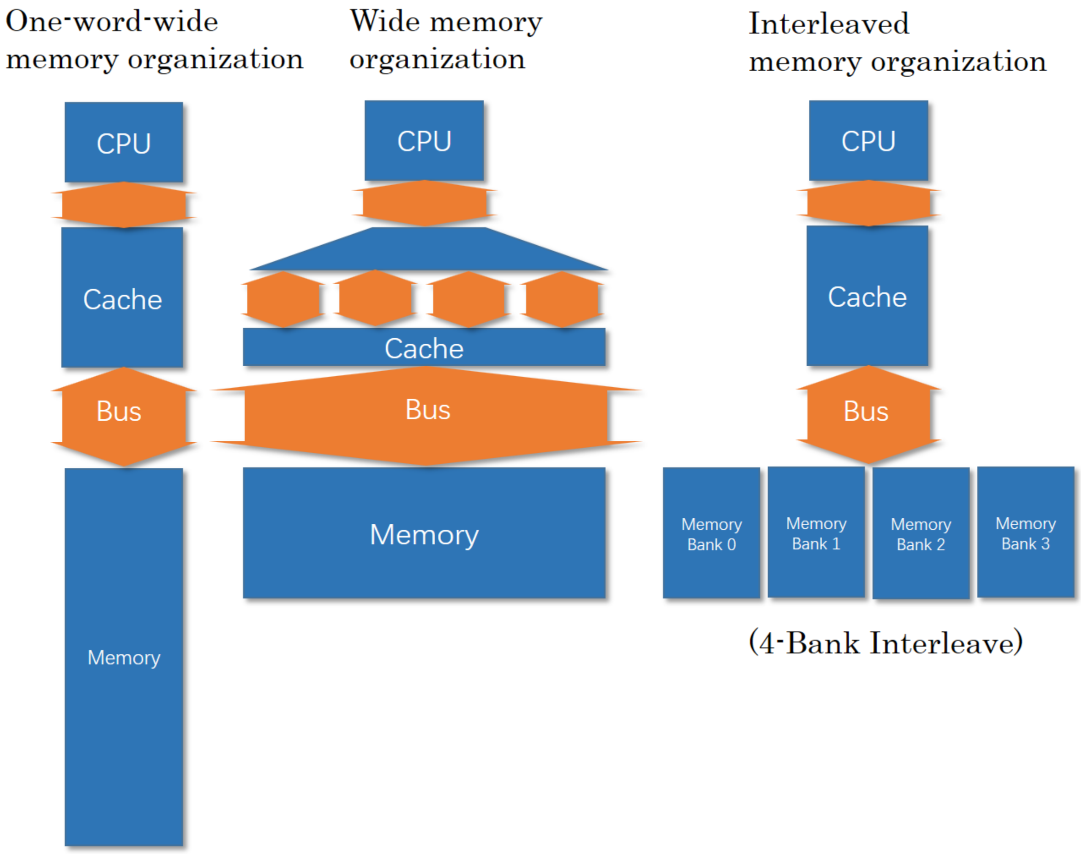
Sorry for the blank here :-(
To be filled ...

3 Memory Hierarchy Design

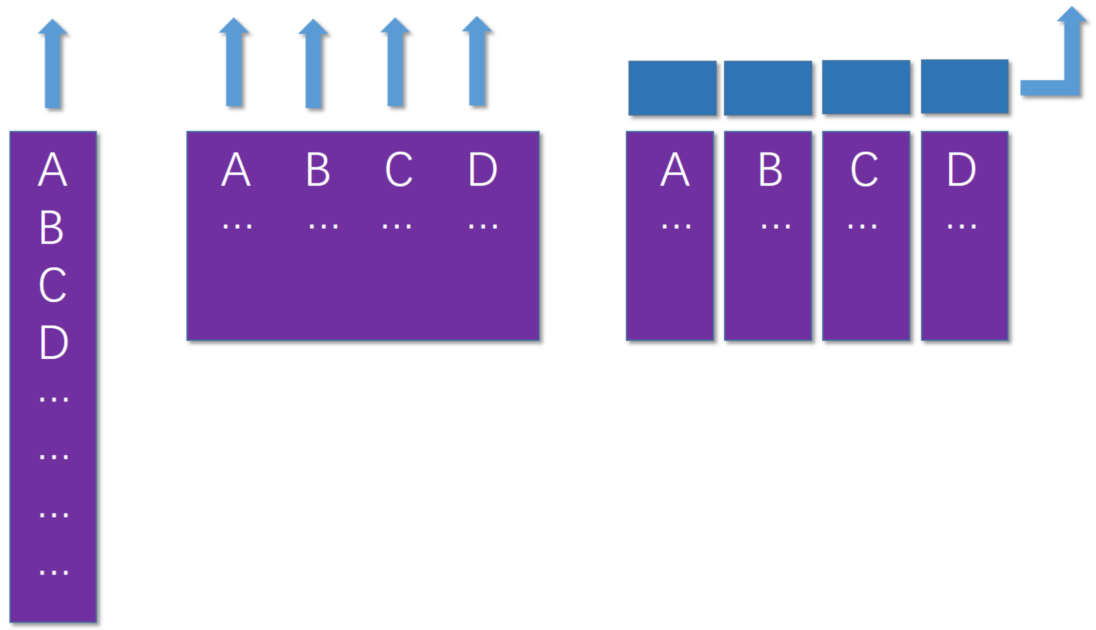
3.1 Memory Technology and Optimizations

3.1.1 Main Memory Organizations

THIS PART[2]



- Simple: CPU, Cache, Bus, Memory same width (32 or 64 bits)
- Wide: CPU/Mux 1 word; Mux/Cache, Bus, Memory N words
- Interleaved: CPU, Cache, Bus 1 word; Memory N Banks, and these independent memory banks can access data parallelly



Memory Organization Performance Timing model (word size is 32 bits, Cache Block is 4 words):

- 1 to send address
- 6 for access time
- 1 to send data

Performance of these three memory organizations under the timing model above:

- Simple: $T = 4 \times (1 + 6 + 1) = 32$
- Wide: $T = 1 + 6 + 1 = 8$
- Interleaved: $T = 1 + 6 + 4 \times 1 = 11$

Wide memory organization has the best time efficiency, but it causes bad compatibility of memory hardwares for its unusual width of Data Bus. Therefore, Interleaved memory organization is often a better choice for PCs.

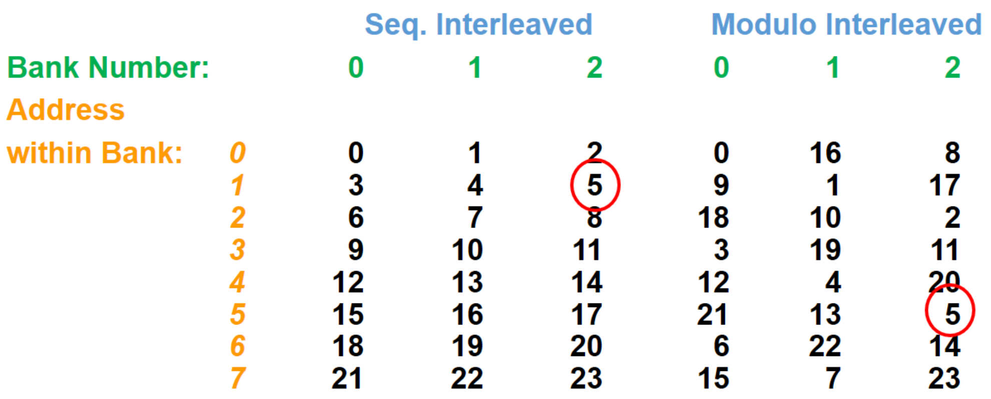
Number of Banks Note that: number banks $\leq$ number clocks to access word in bank, for sequential accesses, otherwise a bank will access a new word before the last accessed one being sent to Bus.

Avoiding Bank Conflicts For the following code, independent banks would have no use, for sequential memory accesses come to the same bank since 512 mod number bank = 0 if number bank = 2^k.

```c
int x[256][512];
for (j = 0; j < 512; j = j+1)
    for (i = 0; i < 256; i = i+1)
        x[i][j] = 2 * x[i][j];
```

Solutions to bank conflicts:

- Software: loop interchange or declaring array not power of 2 (”array padding”)
- Hardware: prime number of banks



		Seq. Interleaved			Modulo Interleaved		
Bank Number:		0	1	2	0	1	2
Address within Bank:	0	0	1	2	0	16	8
	1	3	4	5	9	1	17
	2	6	7	8	18	10	2
	3	9	10	11	3	19	11
	4	12	13	14	12	4	20
	5	15	16	17	21	13	5
	6	18	19	20	6	22	14
	7	21	22	23	15	7	23

Fast Bank Number Prime number of banks would cause efficiency problem in memory addressing. Assume the address x would be located at (R_x, C_x) of memory. If we set number of banks as a prime p,

$$R_x = x/p,\ C_x = x \bmod p$$

as shown in the *Seq. Interleaved* part of the picture above, and calculation of x/p is pretty time consuming. If We arrange data as the *Modulo Interleaved* part of the picture,

$$R_x = x \bmod 2^k,\ C_x = x \bmod p$$

and the calculation of $x \bmod 2^k$ is very fast.

For the calculation of $C_x = x \bmod p$, if we set $p = 2^t - 1$, we have

$$\begin{aligned} x &= kp + r \\ &= k(2^t - 1) + r \\ &= k2^t + r - k \\ r &\equiv x + k \mod 2^k \end{aligned}$$

So we can calculate C_x with bit operations, which is very quick.

And it holds that

$$(R_i, C_i) \neq (R_j, C_j) \text{ if } i \neq j$$

It can be proved with the Chinese remainder theorem. So this arrangement of data does work.

References

[1] John L. Hennessy, David A. Patterson, et al. *Computer Architecture: A Quantitative Approach*, Fifth Edition, 2012.

[2] David A. Patterson. PPT of *CS252 Graduate Computer Architecture*, 2001