NWEN 242

Lecture Notes

# Performance

* Throughput is the total work done in a given time, while response time is the time between the start and end of a task.
* To maximize performance we need to minimize execution time. If computer X is *n*-times faster than computer *Y* , that means that:

Performance*X*

= *n*

Performance*Y*

* Time is the best measure of performance. CPU time is the time the CPU spends computing solely for a task, and does not include time spent waiting for I/O or computing other tasks.
* Clock cycles determine how fast the hardware can perform basic functions. The clock period is the time for a complete clock cycle, the clock rate is the inverse of the clock period.
* The execution time for a program is given by:

*T* = Clock cycles for program × Clock cycle time

CPU time:

CPU time = clock cycles × clock cycle time

* Amdahl’s law says that the performance enhacement possible with a given improvement is limited by how much the enhanced feature is used:

Execution time affected

Execution time after improvement = +Execution time unaffected

Amount of improvement

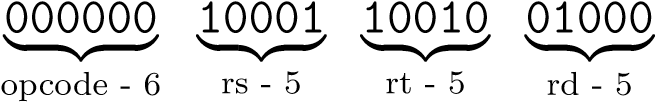
# MIPS

* MIPS stands for **M**icroprocessors without **I**nterlocked **P**ipeline **S**tages. It has 32 registers, and 230 memory words. MIPS has a few different types of instructions:
  + - Arithmetic
    - Data transfer
    - Logic
    - Conditional branching
    - Unconditional jumping
* Registers $v0 and $v1 are for return values, $a0 - $a3 are for arguments to functions, $t0 - $t9 are temporary registers (not preserved across function calls), $s0 - $s7 are saved registers (preserved across function calls), $gp is the global pointer to the middle of the static data segment, $sp is the stack pointer to the last location on the stack, $fp is the frame pointer (preserved across function calls), and $ra holds the return address.
* MIPS design principles:
  + 1. Simplicity favours regularity
    2. Smaller is faster
    3. Make the common case fast
    4. Good design requires good compromises
* To convert a binary number to two’s complement, we simply add 1 to the one’s complement of the number:

0000 1101

1111 0010

1111 0011 • R-format:

 shift amount - 5 function code - 6



I-format:

000000

|

{

z

}

opcode-6

00000

|

{

z

}

rs-5

00000

|

{

z

}

rt-5

0000000000000000

|

{

z

}

constantoraddress-16

J-format:

00

|{z}

opcode-6

00000000000000000000000000

|

{

z

}

constantoraddress-26

* Compilers often create branches/labels for their own purposes even if they’re not in the original code.
* A basic block is a sequence of instructions without any branching, or branch targets, or branch labels (except perhaps at the end). Breaking a program into basic blocks is one of the first stage of compilation.
* MIPS does not include a blt instruction, but it is provided as a pseudoinstruction (note that $at is reserved by the assembler in order to implement pseudo-instructions).
* To fulfill a function call:
  + Put parameters in a place where the function can access them (store in stack)
  + Transfer control to the function (jal functionLabel)
  + Acquire storage resource needed for the procedure (put existing data in stack, and reuse registers)
  + Perform the task
  + Store the result value in $v0 or $v1 so the caller can access it
  + Return control to the caller (jr $ra)
* When jumping to a label with j, the new address to put in the PC will be:

*Address* = *PC* + 4 + 4 ×*steps*

Where ’steps’ is the number of instructions the target is away from the current instruction. Note that with jr, the address is stored in a register instead of being given as a constant.

MIPS addressing modes:

* + - Immediate addressing, the operand is a constant within the instruction (i.e. addi $t0, $zero, 42).
    - Register addressing, the operand is a register (i.e. add $t0, $zero, $t1).
    - Base or displacement addressing, the operand is at the memory location whose address is the sum of a register and a constant in the instruction (i.e. sw $t0, 42($t1)).
    - PC-relative addressing, the branch address is the sum of the PC and a constant in the instruction (i.e. beq $t0, $zero, label).
    - Pseudo-direct addressing, jump address is the 26 bits of the instruction concatenated with the upper bits of the PC (i.e. j label).
* A procedure frame is the segment of the stack containing a procedure’s saved registers and local variables.
* A frame pointer is a value denoting the location of the head of a procedure frame.
* A race condition is when the result of an operation on one thread depends on the sequence/timing of other events.
* MIPS has the ll and sc instructions for handling synchronization. When the two instructions are used in sequence, if the contents of the memory location specified by ll are changed before the sc to the same address occurs, then the sc fails.

# Combinational and Sequential Logic

* With combinational logic, the output is purely a function of the inputs. With sequential logic, the output depends both on the inputs and the history of the inputs. In other words, sequential logic has memory while combinational logic does not.
* The register file in the CPU is made out of flip-flops, decoders, and multiplexors.
* Sequential logic blocks usually update their output on the clock edge, which is called edge-triggered clocking.

A set-reset (SR) latch is composed of two cross-coupled NOR gates, and stores

data. The only invalid inputs are 1 and 1, for which both Q and Q are 0.

* A D-latch is an SR latch preceded by two AND gates that ensure that the invalid inputs of 1 and 1 are never passed to the SR latch.
* A D flipflop is a level triggered non-transparent latch that holds data, and takes input from a data bus and the CPU clock.
* The register file holds 32 registers. There are three register address inputs, one data input port, one write control signal, and two register output ports.

# Datapaths

* The datapath is path along which data flows, and is the hardware that connects instruction memory, the register file, the ALU, and data memory.
* Instruction memory holds program instructions, and the program counter (PC) holds the address of the *next* instruction to be executed.

# Pipelines

* Single-cycle processors are easy to design, but slow. Multi-cycle processors have shorter cycle times and take more cycles per instruction. Pipelining supports multi-cycle design and achieves instruction-level parallelism.
* Pipelining is all about having multiple instructions in the pipeline at once to increase throughput. This can provide a large speed up, even though the individual instructions still take about the same time (some possibly even longer). This requires that the pipeline be kept full.
* The total number of clock cycles required by a program with *n* instructions is

5 + (*n*− 1) = *n* + 4.

* Data is written in the first half of a cycles, and read in the second half.
* Pipelined processors are generally faster than single-cycle processors, as long as the pipeline is kept full. Pipeline registers isolate stages and hold information produced in previous cycles. They also get written on each cycle and control hand-over of partially completed instructions.

Control signals are determined in the instruction decode (ID) stage and pass through the pipeline registers as the instruction moves down the pipeline.

## Hazards

* There are 3 types of hazards: structural, data, and control hazards.
* A structural hazard is a conflict in resources (i.e. a load occuring at the same time as a store), a data hazard is a dependency between instructions, and control hazards occur on branches when the CPU has to choose which instructions to load into the pipeline.
* When the CPU detects a hazard (such as with the hazard detection unit), it will prevent the PC from being updated, freeze the pipeline registers, and insert nop’s as necessary.
* Forwarding is another way to avoid hazards. It involves passing data back to previous stages. This usually happens in the MEM and WB stages. Forwarding requires hazard detection, and then feeding ALU output from the MEM or WB stage to ALU inputs. This removes the need for nop’s.
* Hazard conditions:
  + if (EX/MEM.RegWrite and

(EX/MEM.RegisterRd 6= 0) and

(EX/MEM.RegisterRd = ID/EX.RegisterRs))

* + if (EX/MEM.RegWrite and

(EX/MEM.RegisterRd 6= 0) and

(EX/MEM.RegisterRd = ID/EX.RegisterRt))

* + if (MEM/WB.RegWrite and (MEM/WB.RegisterRd 6= 0) and not

(EX/MEM.RegWrite and (EX/MEM.RegisterRd 6= 0) and (EX/MEM.RegisterRd = ID/EX.RegisterRs)) and

(MEM/WB.RegisterRd = ID/EX.RegisterRs))

* + if (MEM/WB.RegWrite and (MEM/WB.RegisterRd 6= 0) and not

(EX/MEM.RegWrite and (EX/MEM.RegisterRd 6= 0) and (EX/MEM.RegisterRd = ID/EX.RegisterRt)) and

(MEM/WB.RegisterRd = ID/EX.RegisterRt))

* A load-use hazard is a form of data hazard where data loaded from memory using lw is used immediately after. Because the data is only fetched in the MEM stage, one nop must be inserted (even with Forwarding). The condition for this is: if ID/EX.MemRead and ((ID/EX.RegisterRt = IF/ID.RegisterRs) or

(ID/EX.RegisterRt = IF/ID.RegisterRt))