Programmstruktur

```
#include <stdio.h>


#define L_LIMIT 0
#define U_LIMIT 300

int pepe:            /* globale Variable */

main()
{
  int fahr, celsius, lower, upper, step:

  lower = L_LIMIT:   /* untere Grenze */
  upper = U_LIMIT:   /* obere Grenze */
  step = 20:         /* Schrittbreite */

  fahr=lower:
  while(fahr <= upper) {
    celsius = 5 * (fahr-32) / 9:
    printf(" %d    %d\n", fahr, celsius):
    fahr=fahr+step:
  }
}

Ausgabe:
 0          -17
 20          -6
```

Includes & Defines

globale Variablen
globale Deklarationen
(structs, typedefs)

Funktionsdeklarationen

Funktionen

Datatypes:

- Char: 1 Zeichen oder 8bit zahl -> 1Byte
- Float
- Double
- Int
- Short: mindestens 16 bit = 2 byte
- Long: mindestens 32 bit = 4 byte

Prioritäten von Operatoren:

- != starker als =
- ➔ klammern

- *,/,% Multiplikation, Division, Modulo
- +,- Addition, Subtraktion
- <<,>> bitshift links und rechts
- <, >, <=, >= Vergleich
- ==, != Gleichheit, Ungleichheit
- & bitweise AND
- | bitweise OR
- ~ Bitkomplement
- && logisches AND
- || logisches OR
- =,+=,-=,*=,/=,%=,<<=, >>=,|=,&= Zuweisung
- ++,-- Inkrement, Dekrement
- 'c' liefert ASCII Wert des Zeichens c
- ?: bedingt „a=b?1:2"
- sizeof(Vartyp) Speicherbedarf einer Variable in Byte

Printf:

Specifier	Effect	Default precision
`d`	signed decimal	1
`i`	signed decimal	1
`u`	unsigned decimal	1
`o`	unsigned octal	1
`x`	unsigned hexadecimal (`0`–`f`)	1
`X`	unsigned hexadecimal (`0`–`F`)	1
	Precision specifies minimum number of digits, expanded with leading zeros if necessary. Printing a value of zero with zero precision outputs no characters.	
`f`	Print a `double` with *precision* digits (rounded) after the decimal point. To suppress the decimal point use a *precision* of explicitly zero. Otherwise, at least one digit appears in front of the point.	6
`e`, `E`	Print a `double` in exponential format, rounded, with one digit before the decimal point, *precision* after it. A *precision* of zero suppresses the decimal point. There will be at least two digits in the exponent, which is printed as `1.23e15` in `e` format, or `1.23E15` in `E` format.	6
`g`,`G`	Use style `f`, or `e` (`E` with `G`) depending on the exponent. If the exponent is less than –4 or ≥ *precision*, `f` is not used. Trailing zeros are suppressed, a decimal point is only printed if there is a following digit.	unspecified
`c`	The `int` argument is converted to an unsigned char and the resultant character printed.	
`s`	Print a string up to *precision* digits long. If *precision* is not specified, or is greater than the length of the string, the string must be `NUL` terminated.	infinite
`p`	Display the value of a (`void *`) pointer in a system-dependent way.	
`n`	The argument must be a pointer to an integer. The number of characters output so far by this call will be written into the integer.	
`%`	A `%`	—

Struct: Zusammengesetzter Datentyp. Z.B. struct myStruct{int a; int b; char n[32];} ;
Unions: analog zu struct aber Elemente innerhalb teilen sich den selben speicherbereich.

Bitfeld: wie struct aber mit Angabe der Breite in bits z.B.
Struct Packet{

- Unsigned ihl: 4;
- Unsigned version: 4;
- Unsigned id: 16;};

Typedef: umbenennung z.b. typedef int Fritz

Funktionen:

- müssen vor ihrem ersten Aufruf bekannt d.h.
  definiert z.B. int sum(int a, int B){
  Return a+b;}
  oder deklariert z.B. int sum(int a, int b); werden.
- Haben Return value
- Values are NOT passed by reference. Funktion erhält nur wert der Parameter nicht Parameter selbst.

Pointer:
(*) dereferenzieren
(&) addressoperator
Für pointer auf structs, bitfields and union: -> statt .
z.B.

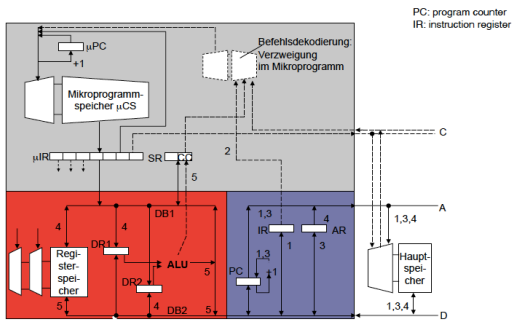
```
int f(struct Point *p)
{
    printf("(%d/%d)",p->x,p->y);
    return 0;
}

int f(struct Point p)
{
    printf("(%d/%d)",p.x.p.y);
    return 0;
}
```

Malloc/calloc/free -> Funktionen für dynamischen Speicher
Malloc: (grösse in bytes)
Calloc: (#Elemente, Grösse eines Elements)
Free: (Zeiger auf Speicherblock)

➔ Dynamisch allozierter Speicher wird nicht automatisch freigegeben, falls pointer auf allozierten Speicherblock verloren geht kann Speicher nicht mehr freigegeben werden.

Prozessorstruktur



Prozessor Struktur:
CPU Elements (Central Processing Unit):

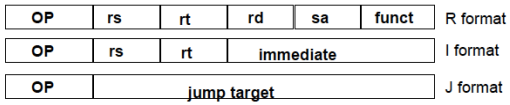
- Program Counter (PC) contains the address of the instruction to be executed next
- Stack: described by a special register (stack pointer), can be used explicitly to save/restore data, used implicitly by procedure call instructions.
- Instruction Register (IR), holds the current instruction being processed by the microprocessor.

RISC vs. CISC:
CISC – Compley Instruction Set Computers -> belief that better performance would be obtained by reducing the number of instructions required to implement a program, lead to the design f processors with very complex instructions.
RISC – Reduced Instruction Set Computers -> simpler instractions, fixed instruction length, faster execution speed per instruction, more instructions executed in same amount of time than CISC. Only load and store are used to access the external memory e.g. MIPS

Instruction Set Architecture (ISA):
Instruction Categories:

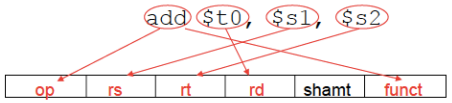
- Computational
- Load/Store
- Jump and Branch

3 Instruction Formats: all 32 bits wide



R: Add, sub, and, or, jr
Slt $t0, $s0, $s1 -> if $s0<$s1 $t0 = 1 else =0
I: Store, load, bne, beq, addi, slti
J: j, jal

Destination <- source1 op source2

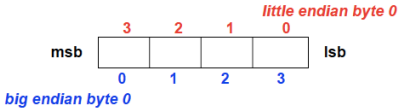


op	6-bits	opcode that specifies the operation
rs	5-bits	register file address of the first source operand
rt	5-bits	register file address of the second source operand
rd	5-bits	register file address of the result's destination
shamt	5-bits	shift amount (for shift instructions)
funct	6-bits	function code augmenting the opcode

Jump if address is to far away (address can't be captured in 16 bits): assembler inserts unconditional ump to branch target and inverts the condition

Endianness:

- Big Endian: leftmost byte is word address
  IBM 360/370, Motorola 68k, MIPS, Sparc, HP PA
- Little Endian: rightmost byte is word address
  Intel 80x86, DEC Vax, DEC Alpha



Test

```
#define LITTLE_ENDIAN 0
#define BIG_ENDIAN 1
int endian() {
  int i = 1;
  char *p = (char *)&i;
  if (p[0] == 1)
      return LITTLE_ENDIAN;
  else
      return BIG_ENDIAN;
}
```

Stacks/ Stack frames:

- Stack frame for a function call: space allocated for the local variables of the function.
- Frame Pointer(FP): indicates the location of the current frame, allows easy access to the local variables.
- on return the current stack frame is popped out and execution continues with the previous stack frame -> Store old instruction pointer(PC) in the stack frame

Load large constants:

- a new "load upper immediate" instruction

```
lui $t0, 1010101010101010
```

16	0	8	1010101010101010

- Then must get the lower order bits right, use

```
ori $t0, $t0, 1010101010101010
```

first 16 bits of immediate

1010101010101010	0000000000000000
0000000000000000	1010101010101010

lower 16 bits of immediate

1010101010101010	1010101010101010

Performance:

Performance$_x$ = 1/execution_time$_x$

$$\text{CPU execution time for a program} = \text{\# CPU clock cycles for a program} \times \text{clock cycle time}$$

or

$$\text{CPU execution time for a program} = \frac{\text{\# CPU clock cycles for a program}}{\text{clock rate}}$$

(= 1/clock cycle time)

10 nsec clock cycle	=>	100 MHz clock rate
5 nsec clock cycle	=>	200 MHz clock rate
2 nsec clock cycle	=>	500 MHz clock rate
1 nsec clock cycle	=>	1 GHz clock rate
500 psec clock cycle	=>	2 GHz clock rate
250 psec clock cycle	=>	4 GHz clock rate
200 psec clock cycle	=>	5 GHz clock rate

Can improve performance by reducing length of the clock cycle or the number of clock cycles required for a program

$$\text{\# CPU clock cycles for a program} = \text{\# Instructions for a program} \times \text{Average clock cycles per instruction}$$

CPI = Clock Cycles per Instruction – average number of clock cycles each instruction takes to execute.

$$\text{Overall effective CPI} = \sum_{i=1}^{n} (CPI_i \times IC_i)$$

- Where IC_i is the count (percentage) of the number of instructions of class i executed
- CPI_i is the (average) number of clock cycles per instruction for that instruction class
- n is the number of instruction classes

$$\text{CPU time} = \text{Instruction\_count} \times \text{CPI} \times \text{clock\_cycle}$$

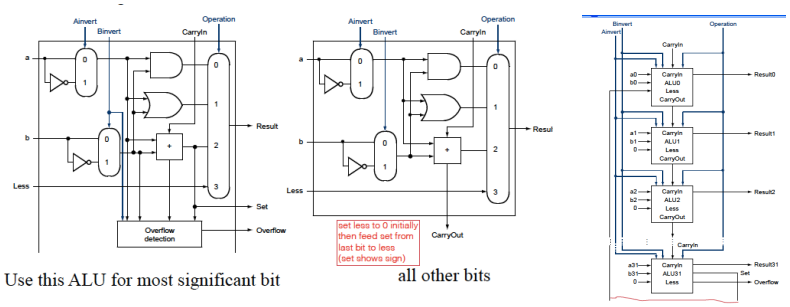
or

$$\text{CPU time} = \frac{\text{Instruction\_count} \times \text{CPI}}{\text{clock\_rate}}$$

Op	Freq	CPI_i	Freq x CPI_i			
ALU	50%	1	.5	.5	.5	.25
Load	20%	5	1.0	.4	1.0	1.0
Store	10%	3	.3	.3	.3	.3
Branch	20%	2	.4	.4	.2	.4
		CPI:	Σ = 2.2	1.6	2.0	1.95

- How much faster would the machine be if a better data cache reduced the average load time to 2 cycles?
  CPU time new = 1.6 x IC x CC so 2.2/1.6 means 37.5% faster
- How does this compare with using branch prediction to shave a cycle off the branch time?
  CPU time new = 2.0 x IC x CC so 2.2/2.0 means 10% faster
- What if two ALU instructions could be executed at once?
  CPU time new = 1.95 x IC x CC so 2.2/1.95 means 12.8% faster

ALU



Use this ALU for most significant bit

all other bits

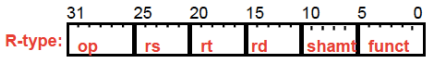
All results together OR then negate -> ZERO is 1 when the result is zero

MIPS = Microprocessor without interlocked pipeline stages
Fetching instructions:
- read instruction from the Instruction Memory
- update PC to hold address of the next instruction
- Pc is updated and instruction memory is read every cycle, no need for explicit write control / read control signal

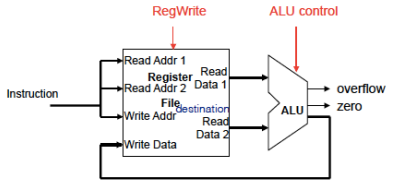
Decoding instructions:
- Send fetched instroction's opcode and function field bits to the control unit
- Read two values from the Register File -> register file addresses are contained in the instruction

Executing R-Type instructions

- R format operations (`add, sub, slt, and, or`)



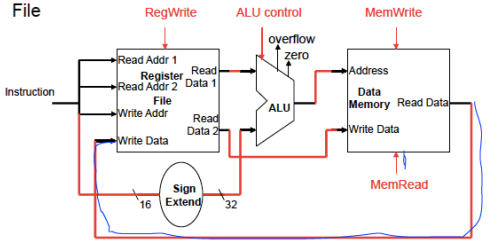
- perform the (op and funct) operation on values in rs and rt
- store the result back into the Register File (into location rd)



- The Register File is not written every cycle (e.g. `sw`), so we need an explicit write control signal for the Register File

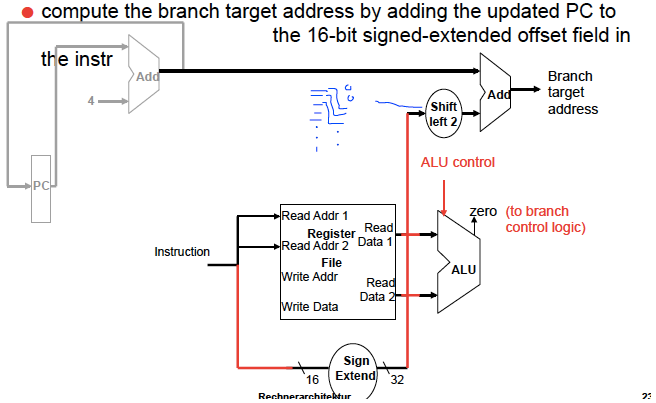
Executing Load and Store Operations

- Load and store operations involves
  - compute memory address by adding the base register (read from the Register File during decode) to the 16-bit signed-extended offset field in the instruction
  - store value (read from the Register File during decode) written to the Data Memory
  - load value, read from the Data Memory, written to the Register File



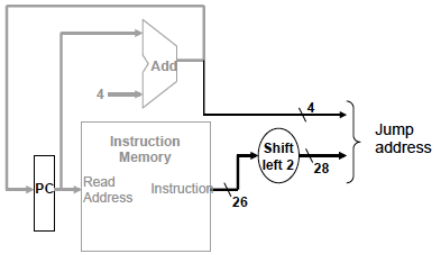
Executing Branch Operations

- Branch operations involves
  - compare the operands read from the Register File during decode for equality (`zero` ALU output)
  - compute the branch target address by adding the updated PC to the 16-bit signed-extended offset field in the instr



Executing Jump Operations

- Jump operation involves
  - replace the lower 28 bits of the PC with the lower 26 bits of the fetched instruction shifted left by 2 bits

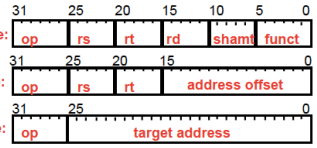


Single Cycle Design: fetch, decode and execute each instruction in one clock cycle

- No datapath resource can be used more than once per instruction, so some must be duplicated
- Multiplexors needed at the input of shared elements with control lines to do the selection
- Write signals to control writing to the Register File and Data Memory
  - ➔ Cycle time is determined by length of the longest path.

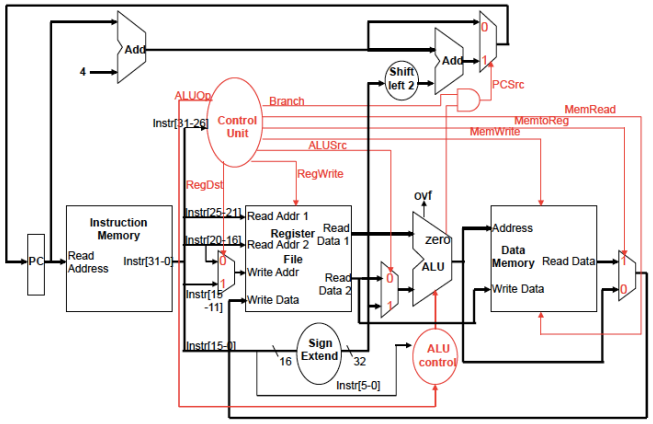
Adding the Control

- Selecting the operations to perform (ALU, Register File and Memory read/write)
- Controlling the flow of data (multiplexor inputs)



- Observations
  - op field always in bits 31-26
  - addr of registers to be read are always specified by the rs field (bits 25-21) and rt field (bits 20-16); for lw and sw rs is the base register
  - addr. of register to be written is in one of two places – in rt (bits 20-16) for lw; in rd (bits 15-11) for R-type instructions
  - offset for beq, lw, and sw always in bits 15-0

Single Cycle Datapath with Control Unit



Multicycle Datapath Approach:

- Let an instruction take more than 1 clock cycle to complete
- Break up instructions into steps where each step takes a cycle while trying to balance the amount of work to be done in each step and restrict each cycle to use only one major functional unit
- Not every instruction takes the same number of clock cycles
  - ➔ Functional units can be used more than once per instruction as long as they are used on different clock cycles -> only need one memory, need only one ALU

Five Execution Steps

- Instruction Fetch
  - Use PC to get instruction and put it in the Instruction Register
  - Increment PC by 4 and put result back in PC
- Instruction Decode and Register Fetch
  - Read registers rs and rt
  - Compute branch address
- Execution, Memory Address Computation, or Branch Completion

  ALU performs one of three functions based on instruction type:
  - Memory Reference: ALUOut <= A + sign-ext(IR[15:0]);
  - R-Type: ALUOut <= A op B
  - Branch: If(A==B) =C <=ALUOut
- Memory Access or R-type instruction completion
  - Loads and stores access memory
    MDR <= Memory[ALUOut];
    Or
    Memory[ALUOut] <= B
  - R-type instruction finish
    Reg[IR[15:11]]<=ALUOut;
- Write-back step

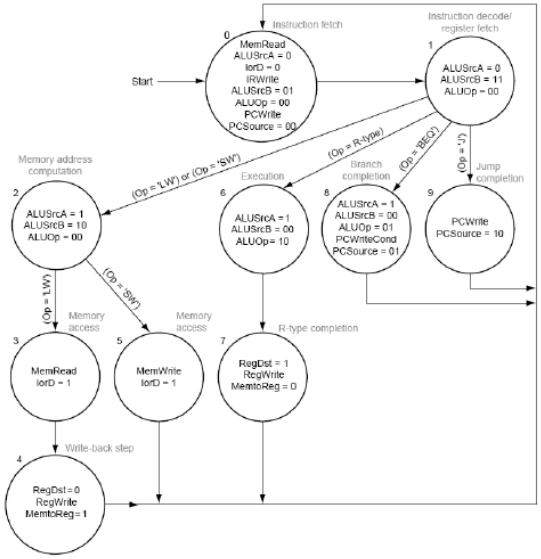
  Only for load instruction :
  Reg[IR[20:16]] <= MDR

INSTRUCTIONS TAKE FROM 3 - 5 CYCLES!

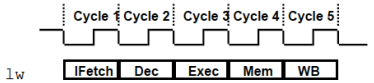
Step name	Action for R-type instructions	Action for memory-reference instructions	Action for branches	Action for jumps
Instruction fetch	IR <= Memory[PC] PC <= PC + 4			
Instruction decode/register fetch	A <= Reg [IR[25:21]] B <= Reg [IR[20:16]] ALUOut <= PC + (sign-extend (IR[15:0]) << 2)			
Execution, address computation, branch/jump completion	ALUOut <= A op B	ALUOut <= A + sign-extend (IR[15:0])	If (A == B) PC <= ALUOut	PC <= {PC [31:28], (IR[25:0]),2'b00)}
Memory access or R-type completion	Reg [IR[15:11]] <= ALUOut	Load: MDR <= Memory[ALUOut] or Store: Memory [ALUOut] <= B		
Memory read completion		Load: Reg[IR[20:16]] <= MDR		

FIGURE 5.30 Summary of the steps taken to execute any instruction class. Instructions take from three to five execution steps. The first two steps are independent of the instruction class. After these steps, an instruction takes from one to three more cycles to complete, depending on the instruction class. The empty entries for the Memory access step or the Memory read completion step indicate that the particular instruction class takes fewer cycles. In a multicycle implementation, a new instruction will be started as soon as the current instruction completes, so these cycles are not idle or wasted. As mentioned earlier, the register file actually reads every cycle, but as long as the IR does not change, the values read from the register file are identical. In particular, the value read into register B during the Instruction decode stage, for a branch or R-type instruction, is the same as the value stored into B during the Execution stage and then used in the Memory access stage for a store word instruction.

Multicycle Control Unit: Finite State Machine



The Five Steps of the Load Instruction



- IFetch: Instruction Fetch and Update PC
- Dec: Instruction Decode, Register Read, Sign Extend Offset
- Exec: Execute R-type; Calculate Memory Address; Branch Comparison; Branch and Jump Completion
- Mem: Memory Read; Memory Write Completion; R-type Completion (RegFile write)
- WB: Memory Read Completion (RegFile write)

INSTRUCTIONS TAKE FROM 3 - 5 CYCLES!

Multicycle Advantages & Disadvantages:

- Multicycle allow functional units to be used more than once per instruction as long as they are used in different clock cycles

BUT

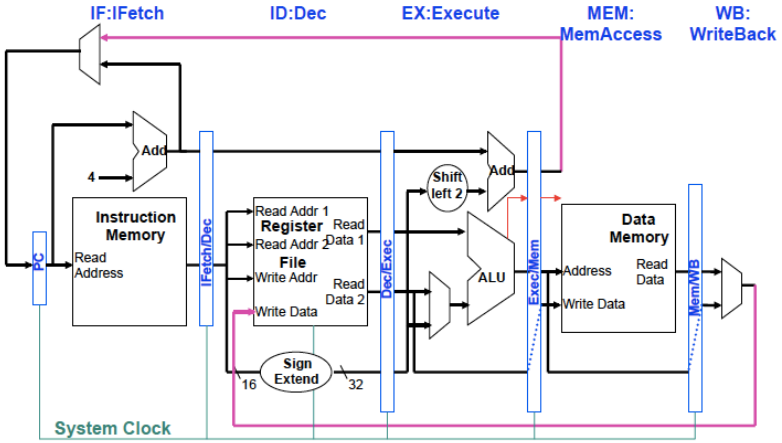
- Requires additional internal state registers, more muxes and more complicated (FSM) control

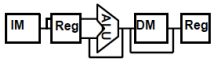
Pipelining

- Start fetching and executing the next instruction before the current one has completed
  - ➔ Pipeline rate is limited by slowest pipeline stage
  - ➔ Improves throughput – total amount of work done in a given time
  - ➔ Instruction latency (execution time, delay time, response time – time from the start of an instruction to its completion) is not reduced

MIPS Pipeline Datapath Modifications

- What do we need to add/modify in our MIPS datapath?
  - State registers between each pipeline stage to isolate them





State registers:

- needed to preserve the destination register address
- All control signals can be determined during decode and held in the state registers between pipeline stages

Pipelining the MIPS ISA

- What makes it easy
  - all instructions are the same length (32 bits)
    - can fetch in the 1st stage and decode in the 2nd stage
  - few instruction formats (three) with symmetry across formats
    - can begin reading register file in 2nd stage
  - memory operations can occur only in loads and stores
    - can use the execute stage to calculate memory addresses
  - each MIPS instruction writes at most one result (i.e., changes the machine state) and does so near the end of the pipeline (MEM and WB)
- What makes it hard
  - structural hazards: what if we had only one memory?
  - control hazards: what about branches?
  - data hazards: what if an instruction's input operands depend on the output of a previous instruction?

Pipeline Hazard:

- Structural hazards: attempt to use the same resource by two different instructions at the same time
- Data hazards: attempt to use data before it is ready
- Control hazards: attempt to make a decision about program control flow before the condition has been evaluated and the new PC target address calculated (branch)

Reading data and instruction from the same memory causes structural hazard

- ➔ Fix with separate instruction and data memories (i$ and D$)

Accessing the register to read and write write at the same time causes structural hazard

- ➔ Fix by always doing writes in the first half of the cycle and reads in the second half

Dependencies backward in time cause hazards

- Read before write data hazard
- Load-use data hazard
  - ➔ Can fix data hazards using stalls – impacts CPI
  - ➔ Fix data hazards by **forwarding** results as soon as they are available to where they are actually needed

- Branch instructions cause Control Hazards
  ➔ Use stalls

Data Forwarding Control Conditions

1. EX/MEM hazard:

```
if (EX/MEM.RegWrite
and (EX/MEM.RegisterRd != 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRs))
      ForwardA = 10
if (EX/MEM.RegWrite
and (EX/MEM.RegisterRd != 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRt))
      ForwardB = 10
```

Forwards the result from the previous instr. to either input of the ALU

Corrected Data Forwarding Control Conditions

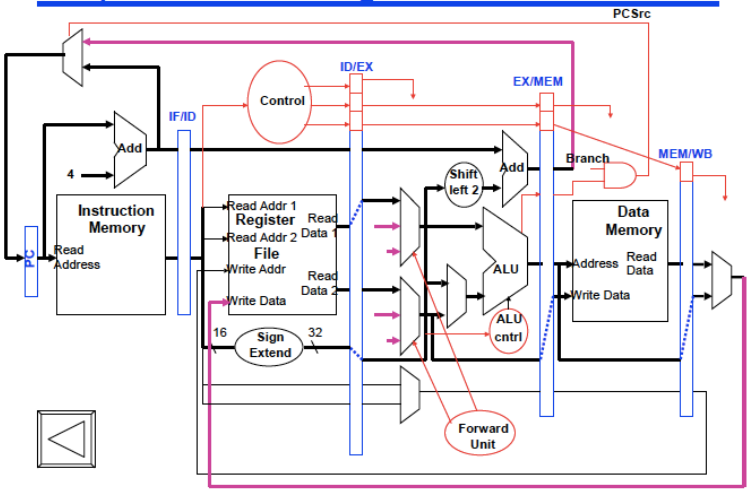
2. MEM/WB hazard:

```
if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd != 0)
and (EX/MEM.RegisterRd != ID/EX.RegisterRs)
and (MEM/WB.RegisterRd = ID/EX.RegisterRs))
      ForwardA = 01

if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd != 0)
and (EX/MEM.RegisterRd != ID/EX.RegisterRt)
and (MEM/WB.RegisterRd = ID/EX.RegisterRt))
      ForwardB = 01
```

Forwards the result from the second previous instruction to either input of the ALU

Datapath with Forwarding Hardware



BEFORE forward Unit:

How many bits wide is each pipeline register?

PC– 32 bits IF/ID – 64 bits ID/EX – 9 + 32x4 + 10 = 147

EX/MEM – 5 + 1 + 32x3 + 5 = 107 MEM/WB – 2 + 32x2+ 5 = 71

AFTER: Each pipeline register is now 157 bits wide: ID/EX – 9 + 32x4 + 10 = 147 + 10 = 157

Memory to Memory Copies:

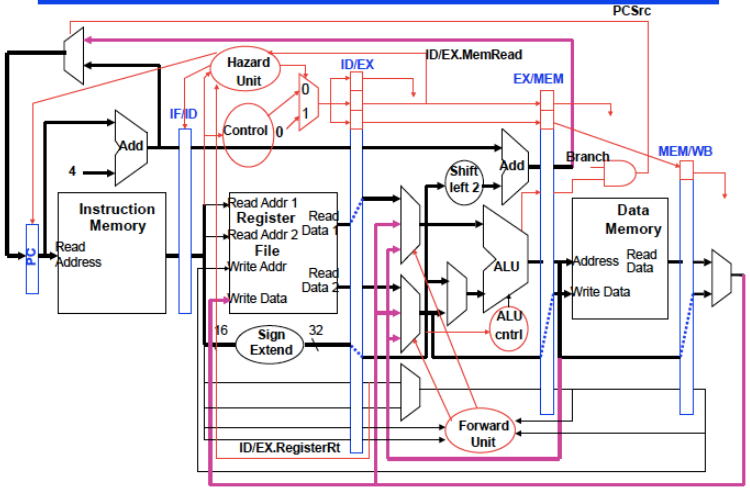
- Lw immediately followed by sw: can avoid stalls by adding forwarding hardware from the MEM/WB register to the data memory input (forward from end of DM in lw to beginning of DM in sw)

Load-use Hazard Detection Unit: inserts a stall between load and its use

2. ID Hazard Detection

```
if (ID/EX.MemRead
and ((ID/EX.RegisterRt = IF/ID.RegisterRs)
or  (ID/EX.RegisterRt = IF/ID.RegisterRt)))
stall the pipeline
```

Adding the Hazard Hardware (and Clock Gating)



Types of Stalls:

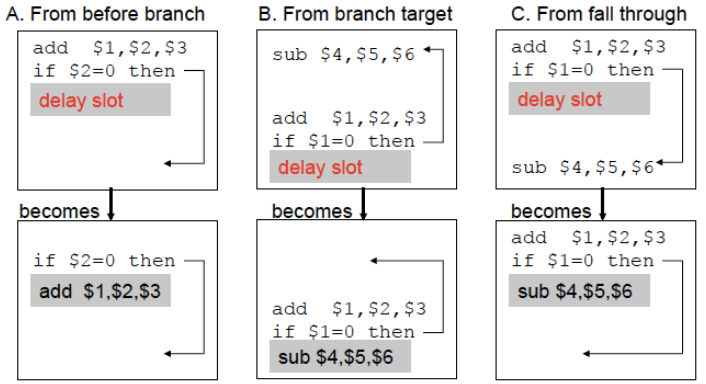
- Noop instruction (bubble) inserted between two instructions in the pipeline e.g. for load-use situations. Done by zeroing control bits in the pipeline register at the appropriate stage
- Flushes (instruction squashing) where an instruction in the pipeline is replaced with a noop instruction e.g for for instructions located sequentially after j instructions. Zero the control bits for the instruction to be flushed

Resolving control hazards:

- Moving branch decisions earlier in pipeline:
  - Move branch decision hardware back to the EX stage
  - Add hardware to compute the branch target address and evaluate the branch decision to the ID stage

  ➔ If the instruction immediately before the branch produces one of the branch source operands a stall is needed because the EX stage ALU operation occurs at the same time as the ID stage branch compare operation
  - If the branch hardware has been moved to the ID stage we can eliminate all branch stalls with **delayed branches** which are defined as always executing the next sequential instruction after the branch instruction – the branch takes effect after that next instruction

Scheduling Branch Delay Slots



- A is the best choice, fills delay slot and reduces IC
- In B and C, the `sub` instruction may need to be copied, increasing IC
- In B and C, must be okay to execute `sub` when branch fails

Static Branch predictions:

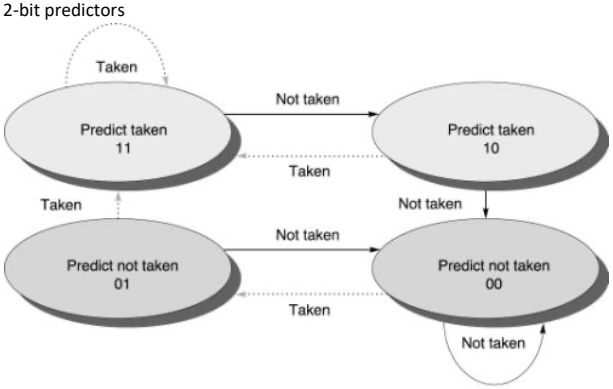
- Assume a given outcome and proceed without waiting to see the actual branch outcome.
  - Predict not taken : works well for "top of the loop" branching structures
  - Predict taken: always incurs one stall (if branch destination hardware has been moved to ID stage), works well for bottom of loop branching structures

Dynamic Branch Predictions

- Use a branch prediction buffer = branch history table (**BHT**). Contains a bit passed to the ID stage through the IF/ID pipeline register that tells whether the branch was taken the last time it was executed.

  ➔ If the prediction is wrong, flush incorrect instruction in pipeline, restart pipeline with the right instruction and invert the prediction bit
- BTB: Branch Target Buffer: in the IF stage can cache the branch target address. The prediction bit in IF/ID selects which next instruction will be loaded into IF/ID, or btb can cache the branch taken instruction while the instruction memory is fetching the next instruction.
- 1-bit prediction: will be incorrect twice when not taken

- 2-bit predictors



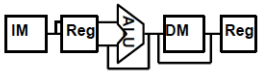
Exceptions:

- Exceptions aka interrupts are another form of control hazard. Arise from
  - R-type arithmetic overflow
  - Trying to execute an undefined instruction
  - An I/O device request
  - An OS service request
  - Hardware malfunction
- The pipeline has to stop executing the offending instruction in midstream, let all prior instructions complete, flush all following instructions, set a register to show the cause of the exception, save the address of the offending instruction, and then jump to a prearranged address (the address of the exception handler code)
- The software (OS) looks at the cause of the exception and "deals" with it

Two Types of Exceptions

- Interrupts – asynchronous to program execution
  - caused by external events
  - may be handled between instructions, so can let the instructions currently active in the pipeline *complete* before passing control to the OS interrupt handler
  - simply suspend and resume user program
- Traps (Exception) – synchronous to program execution
  - caused by internal events
  - condition must be remedied by the trap handler for that instruction, so much stop the offending instruction *midstream* in the pipeline and pass control to the OS trap handler
  - the offending instruction may be retried (or simulated by the OS) and the program may continue or it may be aborted

Where in the Pipeline Exceptions Occur

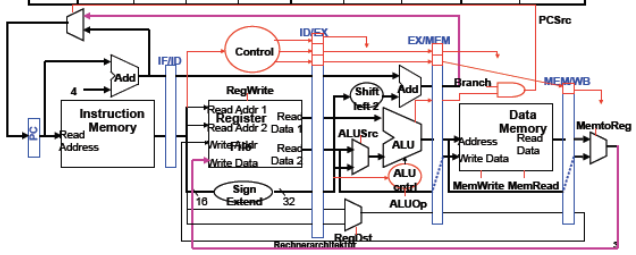


	Stage(s)?	Synchronous?
Arithmetic overflow	execution EX ALU	yes
Undefined instruction	instruction decode ID Reg	yes
TLB or page fault	instruction fetch IF, MEM IM/DM	yes
I/O service request	any	no
Hardware malfunction	any	no

- Beware that multiple exceptions can occur simultaneously in a *single* clock cycle

Control Settings

	EX Stage				MEM Stage			WB Stage	
	RegDst	ALUOp1	ALUOp0	ALUSrc	Brch	MemRead	MemWrite	RegWrite	MemtoReg
R	1	1	0	0	0	0	0	1	0
lw	0	0	0	1	0	1	0	1	1
sw	X	0	0	1	0	0	1	0	X
beq	X	0	1	0	1	0	0	0	X



Superpipeline: increase the depth of the pipeline leading to shorter clock cycles = increase clock rate.
Mulitple issue: fetch and execute more than 1 instruction at one time. (expand every pipeline stage to accommodate multiple instructions)
Speedup = #scalar cycles/ #superscalar cycles.
Paralellism:
Instruction-level parallelism (of a program): measure of the average number of instructions in a program that a processor might be able to execute at the same time.

- Mostly determined by the number of true (data) dependencies and procedural (control) dependencies in relation to the number of other instructions.

Data-level parallelism: perform identical operations on data
Machine Parallelism: (of a processor): a measure of the ability of the processor to take advantage of the ILP of the program.

- Determined by the number of instructions that can be fetched and executed at the same time.

Multiple-Issue Processor Style:

- Static multiple-issue processor (VLIW)
  - Decision on which instructions to execute simultaneously are being made statically (at compile time by compiler)
- Dynamic multiple-issue processors (superscalar)
  - Decision on which instructions to execute simultaneously are being made dynamically (at run time by the hardware)

Multiple-Issue Datapath Responsibilities

- Must handle, with a combination of hardware and software fixes, the fundamental limitations of
  - Storage (data) dependencies – aka data hazards
    - Limitation more severe in a SS/VLIW processor due to (usually) low ILP
  - Procedural dependencies – aka control hazards
    - Ditto, but even more severe
    - Use dynamic branch prediction to help resolve the ILP issue
  - Resource conflicts – aka structural hazards
    - A SS/VLIW processor has a much larger number of potential resource conflicts
    - Functional units may have to arbitrate for result buses and register-file write ports
    - Resource conflicts can be eliminated by duplicating the resource or by pipelining the resource

In-Order Issue with In-Order Completion

- Issue instructions in exact program order and complete them in the same order they were fetched.

In-Order Issue with Out-of-Order completion

- A later instruction may complete before a previous instruction
- Used in single-issue pipelined processors to improve the performance of long-latency operations such as divide
- Stall when there is a resource conflict or when the instructions ready to issue need a result that has not yet been computed
  - ➔ Handling output dependencies: e.g. write before write dependency

Out-of-order Issue with Out-of-order completion

- Fetch and decode instructions beyond the conflicted one, store them in an instruction buffer and flag those instructions in the buffer that don't have resource conflicts or data dependencies
- Flagged instructions are then issued from the buffer without regards to their program order

Antidependencies: when a later instruction (that completes earlier) produces a data value that destroys a data value used as a source in an earlier instruction (that issues later)

Dependencies Review

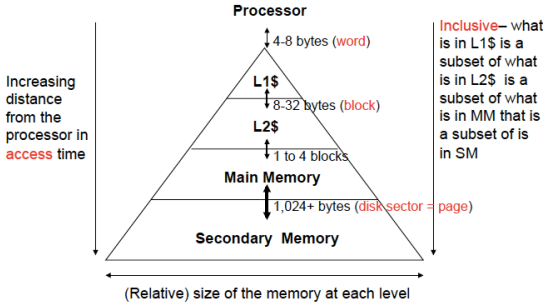
- Each of the three data dependencies
  - True data dependencies (read before write)
  - Antidependencies (write before read)
  - Output dependencies (write before write)

  } storage conflicts

  manifests itself through the use of registers (or other storage locations)
- True dependencies represent the flow of data and information through a program
- Anti- and output dependencies arise because the limited number of registers mean that programmers reuse registers for different computations
- When instructions are issued out-of-order, the correspondence between registers and values breaks down and the values *conflict* for registers

Memory:

Characteristics of the Memory Hierarchy



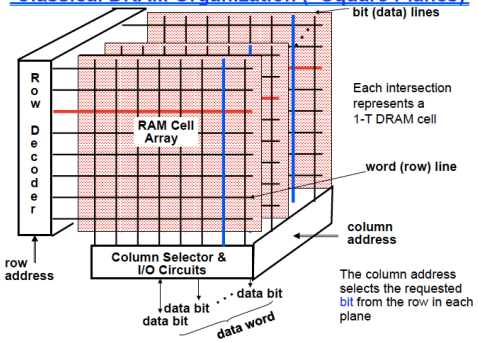
DRAM: Dynamic memory -> needs to be refreshed dynamically. Used by caches for speed and technology compatibility.

- Low density, high power, expenxive, fast
- Static: content will las forever (until power turned off=
- More expensive

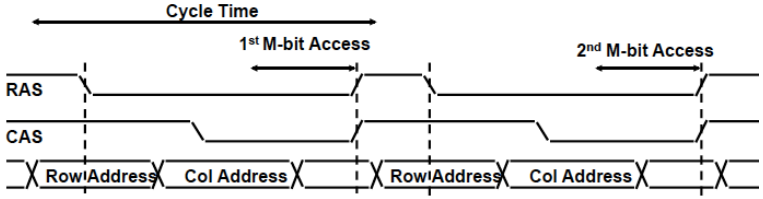
SRAM: static Ram does not need to be refreshed. Used by main memory for size (density)

- Hight density, low power, cheap, slow
- Dynamic: needs to be refreshed regularly (ca. every 8ms)
- Addresses divided into 2 halves (row/column)
    - RAS Row Access Strobe triggering row decoder
    - CAS Column Access Strobe triggering column selector

Classical DRAM Organization (~Square Planes)



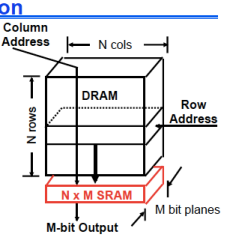
DRAM organization: N rows x N columns X M-bit -> read/write M-bit at a time

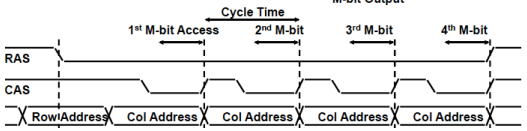


More efficient: add SRAM size NxM transfer “slices” of data to the SRAM where it can be processed more efficiently

Page Mode DRAM Operation

- Page Mode DRAM
    - N x M SRAM to save a row
- After a row is read into the SRAM “register”
    - Only CAS is needed to access other M-bit words on that row
    - RAS remains asserted while CAS is toggled





Memory Performance Metrics:

- Latency: time to access one word
    - Access time: time between requests and when the data is available
    - Cycle time: time between requests
    - Usually cycle time > access time
- Bandwidth: how much data from the memory can be supplied to the processor per unit time
    - Width of the data channel * rate at which it can be used

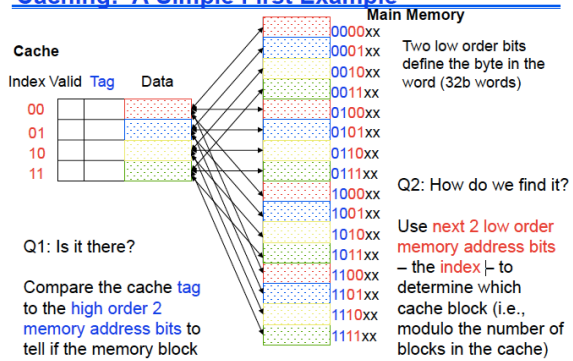
Locality

- Temporal Locality: keep most recently accessed data items closer to the processor -> direct Mapped Cache
- Spatial Locality: move blocks consisting of contiguous words to the upper levels -> Multiword Block Direct Mapped Cache: let cache block hold more than one word

Cache:

- Direct mapped: for each item of data at the lower level there is exactly one location in the cache where it might be – so lots of items at the lower level must share location in the upper level
- Address mapping: (block address) modulo (#blocks in the cache)
- First consider block size of one word

Caching: A Simple First Example



Two low order bits define the byte in the word (32b words)

Q1: Is it there?

Compare the cache tag to the high order 2 memory address bits to tell if the memory block is in the cache

Q2: How do we find it?

Use next 2 low order memory address bits – the index |– to determine which cache block (i.e., modulo the number of blocks in the cache)

(block address) modulo (# of blocks in the cache)

Handling Cache Hits

- Read hits (I$ and D$)
  - this is what we want!
- Write hits (D$ only)
  - allow cache and memory to be inconsistent
    - write the data only into the cache block (write-back the cache contents to the next level in the memory hierarchy when that cache block is "evicted")
    - need a dirty bit for each data cache block to tell if it needs to be written back to memory when it is evicted
  - require the cache and memory to be consistent
    - always write the data into both the cache block and the next level in the memory hierarchy (write-through) so don't need a dirty bit
    - writes run at the speed of the next level in the memory hierarchy – so slow! – or can use a write buffer, so only have to stall if the write buffer is full

Sources of Cache Misses

- Compulsory (cold start or process migration, first reference):
  - First access to a block, "cold" fact of life, not a whole lot you can do about it
  - If you are going to run "millions" of instruction, compulsory misses are insignificant
- Conflict (collision):
  - Multiple memory locations mapped to the same cache location
  - Solution 1: increase cache size
  - Solution 2: increase associativity (next lecture)
- Capacity:
  - Cache cannot contain all blocks accessed by the program
  - Solution: increase cache size

Handling Cache Misses

- Read misses (I$ and D$)
  - stall the entire pipeline, fetch the block from the next level in the memory hierarchy, install it in the cache and send the requested word to the processor, then let the pipeline resume
- Write misses (D$ only)
  1. stall the pipeline, fetch the block from next level in the memory hierarchy, install it in the cache (which may involve having to evict a dirty block if using a write-back cache), write the word from the processor to the cache, then let the pipeline resume

  or (normally used in write-back caches)
  2. Write allocate – just write the word into the cache updating both the tag and data

  or (normally used in write-through caches with a write buffer)
  3. No-write allocate – skip the cache write and just write the word to the write buffer (and eventually to the next memory level), no need to stall if the write buffer isn't full; must invalidate the cache block since it will be inconsistent (now holding stale data)

Multiword Block Considerations

- Read misses (I$ and D$)
  - Processed the same as for single word blocks – a miss returns the entire block from memory
  - Miss penalty grows as block size grows
    - Early restart – datapath resumes execution as soon as the requested word of the block is returned
    - Requested word first – requested word is transferred from the memory to the cache (and datapath) first
  - Nonblocking cache – allows the datapath to continue to access the cache while the cache is handling an earlier miss
- Write misses (D$)
  - Can't use write allocate or will end up with a "garbled" block in the cache (e.g., for 4 word blocks, a new tag, one word of data from the new block, and three words of data from the old block), so must fetch the block from memory first and pay the stall time

Measuring Cache Performance

- Assuming cache hit costs are included as part of the normal CPU execution cycle, then

$$\text{CPU time} = IC \times CPI \times CC$$
$$= IC \times (\underbrace{CPI_{ideal} + \text{Memory-stall cycles}}_{CPI_{stall}}) \times CC$$

- Memory-stall cycles come from cache misses (a sum of read-stalls and write-stalls)

$$\text{Read-stall cycles} = \frac{\text{reads}}{\text{program}} \times \text{read miss rate} \times \text{read miss penalty}$$
$$\text{Write-stall cycles} = \left(\frac{\text{writes}}{\text{program}} \times \text{write miss rate} \times \text{write miss penalty}\right) + \text{write buffer stalls}$$

- For write-through caches, we can simplify this to

$$\text{Memory-stall cycles} = \frac{\text{readsWrites}}{\text{program}} \times \text{miss rate} \times \text{miss penalty}$$

Reducing cache miss rate:

- Allow more flexible block placement -> associative cache can solve ping pong effect in direct mapped cache because two memory locations can coexist.
- Use multiple levels of caches: unified L2 cache (holds instructions and data)
  - Primary cache should focus on minimizing hit time in support of shorter clock cycle (smaller with smaller block sizes)
  - Secondary cache should focus on reducing miss rate to reduce the penalty of long main memory access times (larger with larger block sizes)

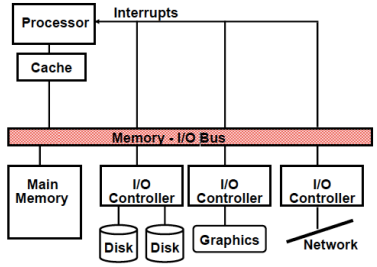
Improving Cache Performance

0. Reduce the time to hit in the cache
   - smaller cache
   - direct mapped cache
   - smaller blocks
   - for writes
     - no write allocate – no "hit" on cache, just write to write buffer
     - write allocate – to avoid two cycles (first check for hit, then write) pipeline writes via a delayed write buffer to cache

1. Reduce the miss rate
   - bigger cache
   - more flexible placement (increase associativity)
   - larger blocks (16 to 64 bytes typical)
   - victim cache – small buffer holding most recently discarded blocks

2. Reduce the miss penalty
   - smaller blocks
   - use a write buffer to hold dirty blocks being replaced so don't have to wait for the write to complete before reading
   - check write buffer (and/or victim cache) on read miss – may get lucky
   - for large blocks fetch critical word first
   - use multiple cache levels – L2 cache not tied to CPU clock rate
   - faster backing store/improved memory bandwidth
     - wider buses
     - memory interleaving, page mode DRAMs

Input Output Systems



Performance Measures:

- I/O bandwidth (throughput) – amount of information that can be input(output) and communicated across an interconnect (e.g. bus) to the processor/memory per unit time.
- I/O response time (latency) – total elapsed time to accomplish an input or output operati

I/O System Performance Example

- A disk workload consisting of 64KB reads and writes where the user program executes 200,000 instructions per disk I/O operation and
  - a processor that sustains 3 billion instr/s and averages 100,000 OS instructions to handle a disk I/O operation

The maximum disk I/O rate (# I/O's/s) of the processor is

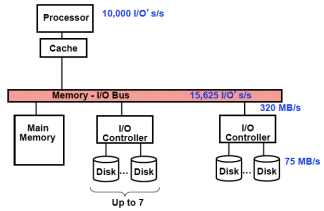
$$\frac{\text{Instr execution rate}}{\text{Instr per I/O}} = \frac{3 \times 10^9}{(200 + 100) \times 10^3} = 10{,}000 \text{ I/O's/s}$$

  - a memory-I/O bus that sustains a transfer rate of 1000 MB/s

Each disk I/O reads/writes 64 KB so the maximum I/O rate of the bus is

$$\frac{\text{Bus bandwidth}}{\text{Bytes per I/O}} = \frac{1000 \times 10^6}{64 \times 10^3} = 15{,}625 \text{ I/O's/s}$$

  - SCSI disk I/O controllers with a DMA transfer rate of 320 MB/s that can accommodate up to 7 disks per controller
  - disk drives with a read/write bandwidth of 75 MB/s and an average seek plus rotational latency of 6 ms



So the processor is the bottleneck, not the bus

- disk drives with a read/write bandwidth of 75 MB/s and an average seek plus rotational latency of 6 ms

Disk I/O read/write time = seek + rotational time + transfer time = 6ms + 64KB/(75MB/s) = 6.9ms

Thus each disk can complete 1000ms/6.9ms or 146 I/O's per second. To saturate the processor requires 10,000 I/O's per second or 10,000/146 = 69 disks

To calculate the number of SCSI disk controllers, we need to know the average transfer rate per disk to ensure we can put the maximum of 7 disks per SCSI controller and that a disk controller won't saturate the memory-I/O bus during a DMA transfer

Disk transfer rate = (transfer size)/(transfer time) = 64KB/6.9ms = 9.56 MB/s

Thus 7 disks won't saturate either the SCSI controller (with a maximum transfer rate of 320 MB/s) or the memory-I/O bus (1000 MB/s). This means we will need 69/7 or 10 SCSI controllers.

Bus: shared communication link that needs to support a range of devices with widely varying latencies and data transfer rates.

- Advantages : versatile (new devices can be added easily and moved between computer systems that use the same bus standard), low cost
- Disadvantages: create a communication bottleneck – bus bandwidth limits the maximum I/O throughput
- Characteristics:
    - Control lines: signal requests and acknowledgments, indicate what type of information is on the data lines
    - Data lines: data, addresses and complex commands
    - Transactions consist of:
        - Master issuing the command (and address) -> request
        - Slave receiving (or sending) the data -> action
        - Defined by what the transaction does to memory (Input data from I/O device to memory or output data from memory to I/O device)

Types of buses:

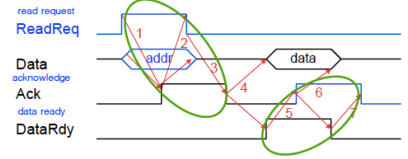
- Processor-memory bus
    - Short and high speed
    - Matched to the memory system to maximize the memory processor bandwidth
    - Optimized for cache block transfers
- I/O bus
    - Usually lengthy and slower
    - Needs to accommodate a wide range of I/O devices
    - Connects to the processor-memory bus or backplane bus
- Backplane bus
    - The backplane is an interconnection structure within the chassis
    - Used as an intermediary bus connecting I/O busses to the processor-memory bus
- Synchronous vs Asynchronous:
    - Synchronous: includes a clock in the control lines and has fixed protocol for communication that is relative to the clock. + involves little logic and can run fast, - every device on the bus must use same clock rate, cannot be long if they are fast to avoid clock skew.
    - Asynchronous: not clocked, requires handshaking protocol and additional control lines. + more flexible, can accommodate wider range of devices and device speeds, can be lengthened without worrying about clock skew or synchronization problems. – slow(er)

Bus Arbitration:

- Multiple devices may need to use the bus at the same time so must have a way to arbitrate multiple requests:
- Try to balance bus priority (highest priority serviced first) and fairness (even lowest priority device should never be completely locked out from the bus)
- Four classes:
    - Daisy chain arbitration: + simple. – cannot assure fairness, slower
    - Centralized, parallel arbitration: + flexible, can assure fairness. – more complicated hardware
    - Distributed arbitration by self-selection: each device wanting the bus places a code indicating its identity on the bus.
    - Distributed arbitration by collision detection: device uses the bus when it's not busy and if a collision happens the device tries again later.

Asynchronous Bus Handshaking Protocol

- Output (read) data from memory to an I/O device



I/O device signals a request by raising ReadReq and putting the addr on the data lines

1. Memory sees ReadReq, reads addr from data lines, and raises Ack
2. I/O device sees Ack and releases the ReadReq and data lines
3. Memory sees ReadReq go low and drops Ack
4. When memory has data ready, it places it on data lines and raises DataRdy
5. I/O device sees DataRdy, reads the data from data lines, and raises Ack
6. Memory sees Ack, releases the data lines, and drops DataRdy
7. I/O device sees DataRdy go low and drops Ack

Bandwidth determinants:

- Whether it's synchronous or asynchronous and timing characteristics of the protocol
- Data bus width
- Whether the bus supports block transfers or only word at a time transfers.

Communication of I/O Devices and Processor

- How the processor directs the I/O devices
    - Special I/O instructions
        - Must specify both the device and the command
    - Memory-mapped I/O
        - Portions of the high-order memory address space are assigned to each I/O device
        - Read and writes to those memory addresses are interpreted as commands to the I/O devices
        - Load/stores to the I/O address space can only be done by the OS
- How the I/O device communicates with the processor
    - Polling – the processor periodically checks the status of an I/O device to determine its need for service
        - Processor is totally in control – but does all the work
        - Can waste a lot of processor time due to speed differences
    - Interrupt-driven I/O – the I/O device issues an interrupts to the processor to indicate that it needs attention

Interrupt-Driven I/O

- An I/O interrupt is asynchronous wrt instruction execution
  - Is not associated with any instruction so doesn't prevent any instruction from completing
    - You can pick your own convenient point to handle the interrupt
- With I/O interrupts
  - Need a way to identify the device generating the interrupt
  - Can have different urgencies (so may need to be prioritized)
- Advantages of using interrupts
  - Relieves the processor from having to continuously poll for an I/O event; user program progress is only suspended during the actual transfer of I/O data to/from user memory space
- Disadvantage – special hardware is needed to
  - Cause an interrupt (I/O device) and detect an interrupt and save the necessary information to resume normal processing after servicing the interrupt (processor)

Direct Memory Access (DMA)

- For high-bandwidth devices (like disks) interrupt-driven I/O would consume a *lot* of processor cycles
- DMA – the I/O controller has the ability to transfer data directly to/from the memory without involving the processor
  1. The processor initiates the DMA transfer by supplying the I/O device address, the operation to be performed, the memory address destination/source, the number of bytes to transfer
  2. The I/O DMA controller manages the entire transfer (possibly thousand of bytes in length), arbitrating for the bus
  3. When the DMA transfer is complete, the I/O controller interrupts the processor to let it know that the transfer is complete
- There may be multiple DMA devices in one system
  - Processor and I/O controllers contend for bus cycles and for memory

The DMA Stale Data Problem

- In systems with caches, there can be two copies of a data item, one in the cache and one in the main memory
  - For a DMA read (from disk to memory) – the processor will be using stale data if that location is also in the cache
  - For a DMA write (from memory to disk) and a write-back cache – the I/O device will receive stale data if the data is in the cache and has not yet been written back to the memory
- The coherency problem is solved by
  1. Routing all I/O activity through the cache – expensive and a large negative performance impact
  2. Having the OS selectively invalidate the cache for an I/O read or force write-backs for an I/O write (flushing)
  3. Providing hardware to selectively invalidate or flush the cache – need a hardware snooper

I/O and the Operating System

- The operating system acts as the interface between the I/O hardware and the program requesting I/O
  - To protect the shared I/O resources, the user program is not allowed to communicate directly with the I/O device
- Thus OS must be able to give commands to I/O devices, handle interrupts generated by I/O devices, provide equitable access to the shared I/O resources, and schedule I/O requests to enhance system throughput
  - I/O interrupts result in a transfer of processor control to the supervisor (OS) process