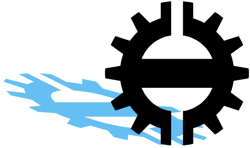
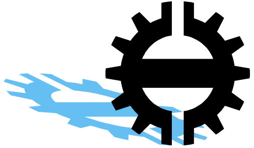


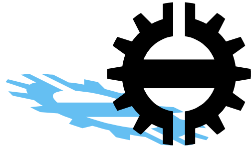
Processor Design – Pipelining and Parallelism

Professor Jari Nurmi
Institute of Digital and Computer Systems
Tampere University of Technology, Finland
email jari.nurmi@tut.fi

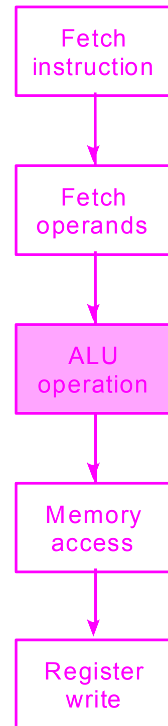


Increasing Parallelism

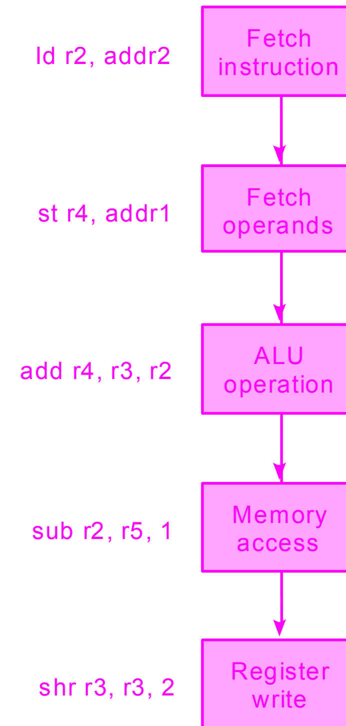
- Pipelining
  - Pipelined processor design
  - Hazards
- Instruction-level parallelism (ILP)
  - Superscalar
  - Very long instruction word (VLIW) machines
- Data-level parallelism (DLP)
  - Short vectors



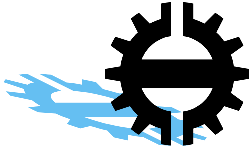
Executing Machine Instructions without and with Pipelining



(a) Without pipelining



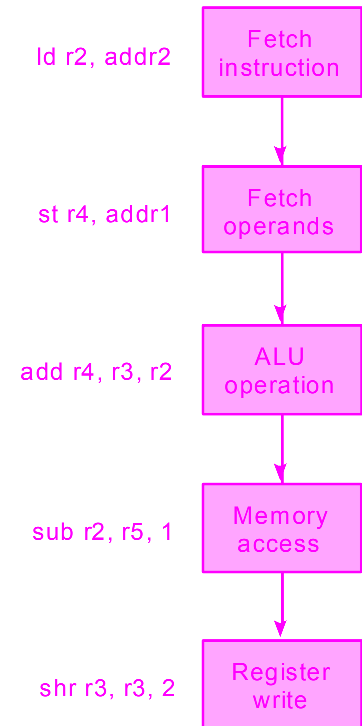
(b) With pipelining

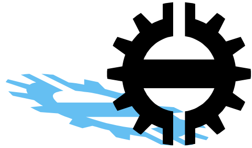


The Pipeline Stages

- 5 pipeline stages are shown
  1. Fetch instruction
  2. Fetch operands
  3. ALU operation
  4. Memory access
  5. Register write
- 5 instructions are executing

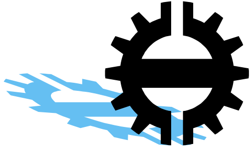
```
shr r3, r3, #2  ;Storing result into r3
sub r2, r5, #1  ;Idle—no memory access needed
add r4, r3, r2  ;Performing addition in ALU
st  r4, addr1   ;Accessing r4 and addr1
ld  r2, addr2   ;Fetching instruction
```



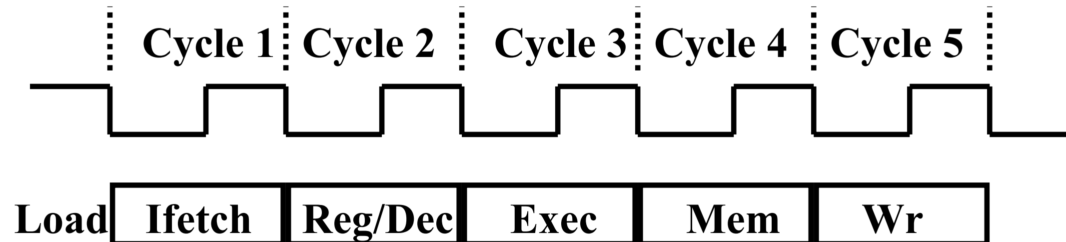


Notes on Pipelining Instruction Processing

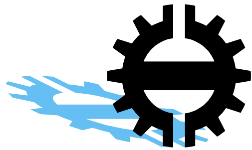
- Pipeline stages are shown top to bottom in order traversed by one instruction
- Instructions listed in the order they are fetched
- Order of instructions in pipeline is reverse of listed
- If each stage takes 1 clock:
  - every instruction takes 5 clocks to complete
  - some instruction completes every clock tick
- Two performance issues: instruction latency and instruction bandwidth (throughput)



Alternative notation!



- Ifetch: Instruction Fetch
  - Fetch the instruction from the Instruction Memory
- Reg/Dec: Registers Fetch and Instruction Decode
- Exec: Calculate the memory address
- Mem: Read the data from the Data Memory
- Wr: Write the data back to the register file



From Single-Cycle to Pipelined

Cycle 1 | Cycle 2

Clk

Single Cycle Implementation:

Load | Store | Waste

Cycle 1 | Cycle 2 | Cycle 3 | Cycle 4 | Cycle 5 | Cycle 6 | Cycle 7 | Cycle 8 | Cycle 9 | Cycle 10

Clk

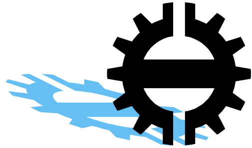
Multiple Cycle Implementation:

Load: Ifetch | Reg | Exec | Mem | Wr
Store: Ifetch | Reg | Exec | Mem
R-type: Ifetch

Pipeline Implementation:

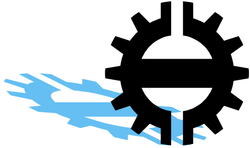
Load: Ifetch | Reg | Exec | Mem | Wr
Store: Ifetch | Reg | Exec | Mem | Wr
ALU op: Ifetch | Reg | Exec | Mem | Wr

Source: Patterson, Hennessy – Computer Architecture; The Hardware/Software Interface



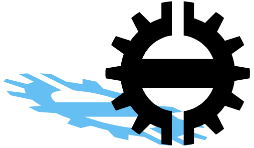
Overall Throughput

- Suppose we execute 100 instructions
- Single Cycle Machine
  - 45 ns/cycle x 1 CPI x 100 inst = 4500 ns
- Multicycle Machine
  - 10 ns/cycle x 4.6 CPI (due to inst mix) x 100 inst = 4600 ns
- Ideal pipelined machine
  - 10 ns/cycle x (1 CPI x 100 inst + 4 cycle drain) = 1040 ns



Dependence Among Instructions

- Execution of some instructions can depend on the completion of others in the pipeline
- One solution is to “stall” the pipeline
  - early stages stop while later ones complete processing
- Dependences involving registers can be detected and data “forwarded” to instruction needing it, without waiting for register write
- Dependence involving memory is harder and is sometimes addressed by restricting the way the instruction set is used
  - “Branch delay slot” is example of such a restriction
  - “Load delay” is another example



Branch and Load Delay Examples

Branch Delay

```
brz r2, r3
add r6, r7, r8
st  r6, addr1
```

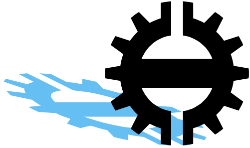
- `add r6, r7, r8` — This instruction always executed
- `st r6, addr1` — Only done if $r2 \neq 0$

Load Delay

```
ld  r2, addr
add r5, r1, r2
shr r1,r1,#4
sub r6, r8, r2
```

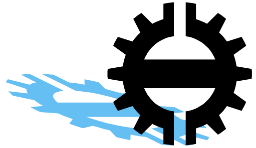
- `add r5, r1, r2` — This instruction gets “old” value of r2
- `sub r6, r8, r2` — This instruction gets r2 value loaded from addr

- Working of instructions is not changed, but the way they work together is



Characteristics of Pipelined Processor Design

- Main memory must operate in one cycle
  - This can be accomplished by expensive memory, but
  - It is usually done with cache, to be discussed later
- Instruction and data memory must appear separate
  - Harvard architecture has separate instruction and data memories
  - Again, this is usually done with separate caches (except for DSPs)
- Few buses are used
  - Most connections are point to point
  - Some few-way multiplexers are used
- Data is latched (stored in temporary registers) at each pipeline stage—called "pipeline registers"
- ALU operations take only 1 clock cycle



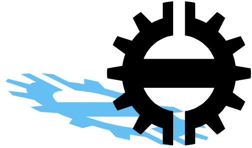
Pipelined Hardware Block Diagram

PIPELINE
STAGES

PIPELINE
REGISTERS

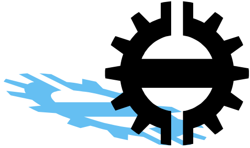
operation

REGISTER
TRANSFERS



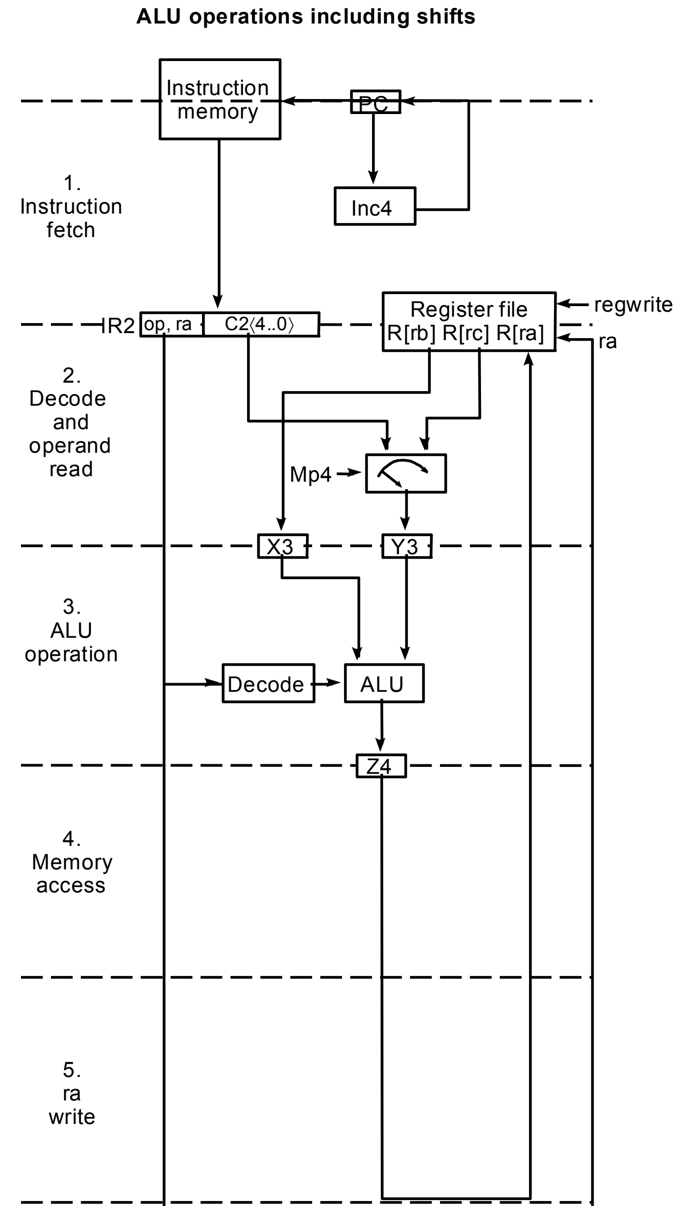
Adapting Instructions to Pipelined Execution

- All instructions must fit into a common pipeline stage structure
- A 5-stage pipeline is typically used in RISC processors
  - (1) Instruction fetch
  - (2) Decode and operand access
  - (3) ALU operations
  - (4) Data memory access
  - (5) Register write
- We must fit load/store, ALU, and branch instructions into this pattern



ALU Instructions

- Instructions fit into 5 stages
- Second ALU operand comes either from a register or instruction register c2 field
- Opcode must be available in stage 3 to tell ALU what to do
- Result register, ra, is written in stage 5
- No memory operation



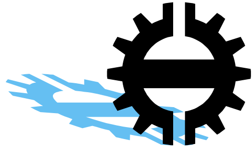


Fig 5.4 The Memory Access Instructions: ld, ldr, st, and str

- ALU computes effective addresses
- Stage 4 does read or write
- Result register written only on load

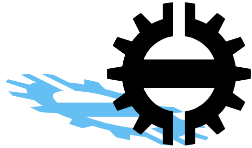
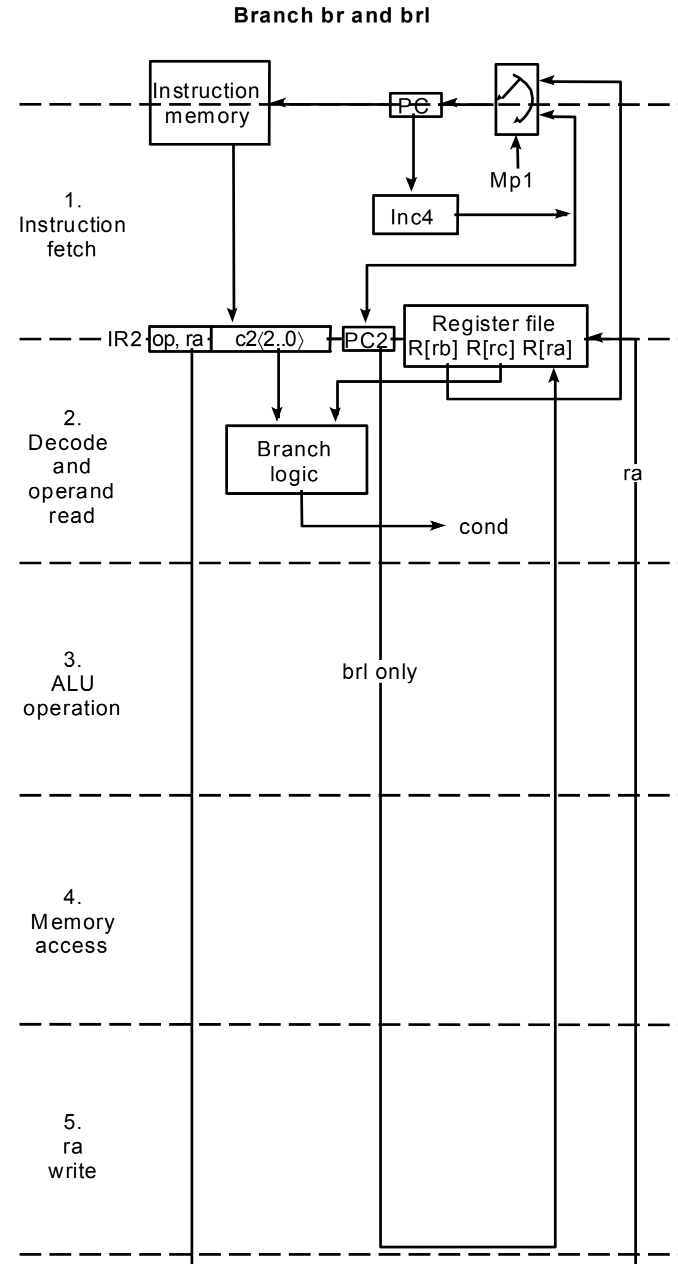


Fig 5.5 The Branch Instructions

- The new program counter value is known in stage 2—but not in stage 1
- Only branch and link does a register write in stage 5
- There is no ALU or memory operation



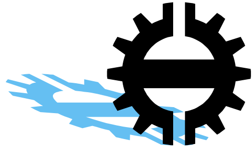


Fig 5.6 The SRC Pipeline Registers and RTN Specification

- The pipeline registers pass information from stage to stage
- RTN specifies output register values in terms of input register values for stage

Instruction memory
PC
PC + 4

1. Instruction fetch
IR2 ← M[PC] :
C2 ← PC + 4 ;

R[rb]
IR2: op ra rb rc c1 c2
PC2
Register file: rb R[rb] rc R[rc] ra R[ra]

2. Decode and operand read
X3 ← l-s2 → (rel2 → PC2 : disp2 → R[rb]) :
brl2 → PC2 : alu2 → R[rb] :
Y3 ← l-s2 → (rel2 → c1 : disp2 → c2) :
branch2 → : alu2 → (imm2 → c2 : ¬imm2→ R[rc]) :
MD3 ← store2 → R[ra] : IR3← IR2 : stop2 → Run ← 0 :
PC ← ¬branch2 → PC + 4 : branch2 → (cond(IR2, R[rc]) → R[rb] ;
¬cond(IR2, R[rc]) → PC + 4) ;

IR3 X3 Y3 MD3

3. ALU operation
Z4 ← (l-s3 → X3 + Y3 :
brl3 → X3 :
alu3 → X3 op Y3) :
MD4 ← MD3 :
IR4 ← IR3 ;

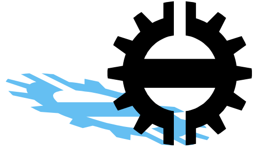
IR4 Z4 MD4

4. Memory access
Z5 ← (load4 → M[Z4]:
ladr4 ∨ branch4 ∨ alu4 → Z4) :
store4 → (M[Z4] ← MD4) :
IR5 ← IR4 ;

Data memory

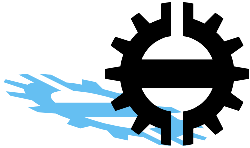
IR5 Z5

5. Register write
regwrite5 → (R[ra] ← Z5) ;



Global State of the Pipelined SRC

- PC, the general registers, instruction memory, and data memory represent the global machine state
- PC is accessed in stage 1 (and stage 2 on branch)
- Instruction memory is accessed in stage 1
- General registers are read in stage 2 and written in stage 5
- Data memory is only accessed in stage 4



Restrictions on Access to Global State by Pipeline

- We see why separate instruction and data memories (or caches) are needed
- When a load or store accesses data memory in stage 4, stage 1 is accessing an instruction
  - Thus two memory accesses occur simultaneously
- Two operands may be needed from registers in stage 2 while another instruction is writing a result register in stage 5
  - Thus as far as the registers are concerned, 2 reads and a write happen simultaneously
- Increment of PC in stage 1 must be overridden by a successful branch in stage 2

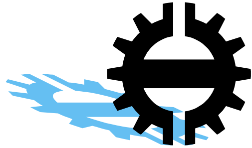


Fig 5.7 The Pipelined Data Path with Selected Control Signals

- Most control signals shown and given values
- Multiplexer control is stressed in this figure

1. Instruction fetch
2. Decode and operand read
3. ALU operation
4. Memory access
5. Register write

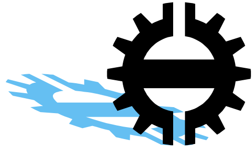
Mp1 ← (¬(branch2 ∧ cond) → Inc4):
((branch2 ∧ cond) → PC2):

Mp2 ← (¬store → rc):
(store → ra):

Mp3 ← (rl ∨ branch → PC2):
(dsp ∨ alu → R1):

Mp4 ← (rl → c1):
(dsp ∨ imm → c2):
(alu ∧ 71mm ¬imm → R2):

Mp5 ← (¬load → Z4):
(load → mem data):



Example of Propagation of Instructions Through Pipe

```
100: add  r4, r6, r8;     R[4] ← R[6] + R[8]
104: ld   r7, 128(r5);    R[7] ← M[R[5]+128]
108: brl  r9, r11, 001;   PC ← R[11]: R[9] ← PC
112: str  r12, 32;        M[PC+32] ← R[12]
     . . . . . .
512: sub  ...             next instr. ...
```

- It is assumed that R[11] contains 512 when the brl instruction is executed
- R[6] = 4 and R[8] = 5 are the add operands
- R[5] =16 for the ld and R[12] = 23 for the str

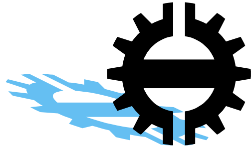


Fig 5.8 First Clock Cycle: add Enters Stage 1 of Pipeline

- Program counter is incremented to 104

512: sub ...

. . . .

112: str r12, #32

108: brl r9, r11, 001

104: ld r7, r5, #128

100: add r4, r6, r8

1. Instruction fetch

Instruction memory

PC 100

104

Mp1

Inc4

100: add r4, r6, r8

104

2. Decode and operand read

IR2 op ra rb rc c1 c2

PC2

Register file a1 R1 a2 R2 a3 R3

G1 GA1 G2 W3

rb rc ra c2 c1 c2⟨2..0⟩

Mp2 cond

Branch logic

Mp3 Mp4

IR3 X3 Y3 MD3

3. ALU operation

op ra

Decode

ALU op'n

ALU

IR4 Z4 MD4

4. Memory access

op ra

addr

Data memory

Decode load/store

Mp5

IR5 Z5

5. ra write

op ra

Decode load ∨ lader ∨ brl ∨ alu

value

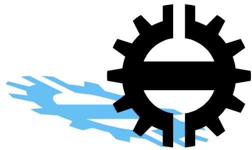


Fig 5.9 Second Clock Cycle: add Enters Stage 2, While ld is Being Fetched at Stage 1

- add operands are fetched in stage 2

512: sub ...
. . . .
112: str r12, #32
108: brl r9, r11, 001
104: ld r7, r5, #128
100: add r4, r6, r8

1. Instruction fetch
Instruction memory
PC
104
108
Mp1
Inc4
104: ld r7 , r5, 128
108
PC2
104
IR2
add r4, r6, r8
Register file
r6 4 r8 5 a3 R3
G1
GA1
G2
W3
rb
Mp2
cond
rc
ra
4
c2
c2⟨2..0⟩
c1
5
Branch logic
2. Decode and operand read
add r4
Mp3
Mp4
IR3
X3
Y3
MD3
op ra
ALU op'n
Decode
ALU
3. ALU operation
IR4
Z4
MD4
addr
Data memory
op ra
4. Memory access
Decode
load/store
Mp5
IR5
Z5
ra
op
value
5. ra write
Decode
load ∨ lader ∨ brl ∨ alu

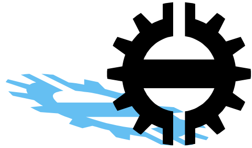


Fig 5.10 Third Clock Cycle: `brl` Enters the Pipeline

- add performs its arithmetic in stage 3

512: sub ...

. . . .

112: str r12, #32

108: brl r9, r11, 001

104: ld r7, r5, #128

100: add r4, r6, r8

1. Instruction fetch
Instruction memory
PC 108
112
Mp1
Inc4
108: brl r9 , r11, 001
112
PC2
IR2
ld r7 , r5, 128
108
a1 R1 a2 R2 a3 R3
r5 16
G1
GA1
G2
W3
2. Decode and operand read
rb
rc
ra
c2
c1
c2⟨2..0⟩
Mp2
cond
Branch logic
16
128
Mp3
Mp4
ld r7
IR3
add r4
X3 4
Y3 5
MD3
3. ALU operation
op ra
Decode
add
ALU
add r4
9
IR4
Z4
MD4
4. Memory access
op ra
addr
Data memory
Decode
load/store
Mp5
IR5
Z5
5. ra write
op
ra
value
Decode
load ∨ lader ∨ brl ∨ alu

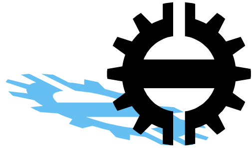


Fig 5.11 Fourth Clock Cycle: `str` Enters the Pipeline

- `add` is idle in stage 4
- Success of `brl` changes program counter to 512

512: sub ...

. . . . . .

112: str r12, #32
108: brl r9, r11, 001
104: ld r7, r5, #128
100: add r4, r6, r8

1. Instruction fetch
2. Decode and operand read
3. ALU operation
4. Memory access
5. ra write

Instruction memory
PC 112 ← 512
Mp1
Inc4
512
112: str r12, 32
116
PC2 112
IR2: brl r9 , r11 001
op ra rb rc c1 c2
a1 R1 a2 R2 a3 R3
r11 512
G1 GA1 G2 W3
Mp2
cond
rb rc ra c1
c2⟨2..0⟩=001
Branch logic
112
Mp3 Mp4
brl r9
IR3: ld r7
X3 16 Y3 128
MD3
op ra
Decode
add
ALU
ld r7
144
IR4: add r4
Z4 9
MD4
addr
Data memory
op ra
Decode
load/store 9
Mp5
add r4
IR5
Z5
ra
op
value
Decode
load ∨ lader ∨ brl ∨ alu

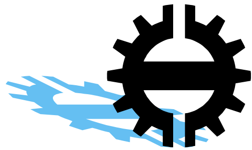
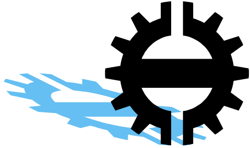


Fig 5.12 Fifth Clock Cycle: add Completes, sub Enters the Pipeline

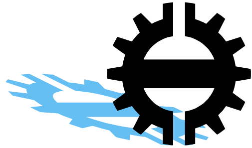
- add completes in stage 5
- sub is fetched from location 512 after successful brl

```
512:  sub   ...
      . . . . . .
112:  str   r12, #32
108:  brl   r9, r11, 001
104:  ld    r7, r5, #128
100:  add   r4, r6, r8
```



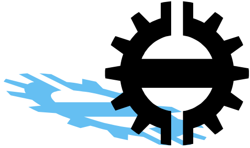
Functions of the Pipeline Registers in SRC

- Registers between stages 1 and 2:
  - IR2 holds full instruction including any register fields and constant
  - PC2 holds the incremented PC from instruction fetch
- Registers between stages 2 and 3:
  - IR3 holds opcode and ra (needed in stage 5)
  - X3 holds PC or a register value (for link or 1st ALU operand)
  - Y3 holds c1 or c2 or a register value as 2nd ALU operand
  - MD3 is used for a register value to be stored in memory



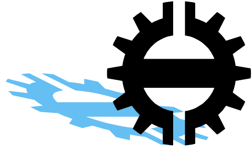
Functions of the Pipeline Registers in SRC (cont'd)

- Registers between stages 3 and 4:
  - IR4 has op code and ra
  - Z4 has memory address or result register value
  - MD4 has value to be stored in data memory
- Registers between stages 4 and 5:
  - IR5 has opcode and destination register number, ra
  - Z5 has value to be stored in destination register: from ALU result, PC link value, or fetched data



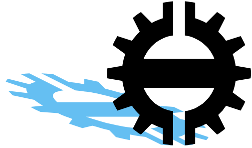
Functions of the SRC Pipeline Stages

- Stage 1: fetches instruction
  - PC incremented or replaced by successful branch in stage 2
- Stage 2: decodes instruction and gets operands
  - Load or store gets operands for address computation
  - Store gets register value to be stored as 3rd operand
  - ALU operation gets 2 registers or register and constant
- Stage 3: performs ALU operation
  - Calculates effective address or does arithmetic/logic
  - May pass through link PC or value to be stored in memory



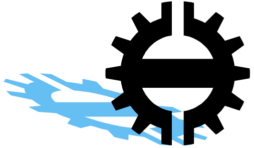
Functions of the SRC Pipeline Stages (cont'd)

- Stage 4: accesses data memory
  - Passes Z4 to Z5 unchanged for nonmemory instructions
  - Load fills Z5 from memory
  - Store uses address from Z4 and data from MD4 (no longer needed)
- Stage 5: writes result register
  - Z5 contains value to be written, which can be ALU result, effective address, PC link value, or fetched data
  - ra field always specifies result register in SRC



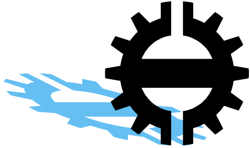
Dependence Between Instructions in Pipe: Hazards

- Instructions that occupy the pipeline together are being executed in parallel
- This leads to the problem of instruction dependence, well known in parallel processing
- The basic problem is that an instruction depends on the result of a previously issued instruction that is not yet complete
- Two (or three) categories of hazards
  - Data hazards: incorrect use of old and new data
  - Branch hazards: fetch of wrong instruction on a change in PC
  - (Structural hazards: insufficient resources to execute all combinations of instructions)



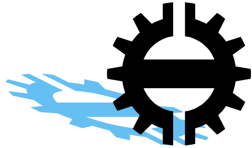
Classification of Data Hazards

- A read after write hazard (RAW) arises from a flow dependence, where an instruction uses data produced by a previous one
- A write after read hazard (WAR) comes from an anti-dependence, where an instruction writes a new value over one that is still needed by a previous instruction
- A write after write hazard (WAW) comes from an output dependence, where two parallel instructions write the same register and must do it in the order in which they were issued



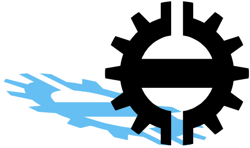
Data Hazards in SRC

- Since all data memory access occurs in stage 4, memory writes and reads are sequential and give rise to no hazards
- Since all registers are written in the last stage, WAW and WAR hazards do not occur
  - Two writes always occur in the order issued, and a write always follows a previously issued read
- SRC hazards on register data are limited to RAW hazards coming from flow dependence
- Values are written into registers at the end of stage 5 but may be needed by a following instruction at the beginning of stage 2



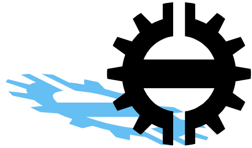
Possible Solutions to the Register Data Hazard Problem

- Detection:
  - The machine manual could list rules specifying that a dependent instruction cannot be issued less than a given number of steps after the one on which it depends
  - This is usually too restrictive
  - Since the operation and operands are known at each stage, dependence on a following stage can be detected
- Correction:
  - The dependent instruction can be “stalled” and those ahead of it in the pipeline allowed to complete
  - Result can be “forwarded” to a following inst. in a previous stage without waiting to be written into its register
- Preferred SRC design will use detection, forwarding and stalling only when unavoidable



Detecting Hazards and Dependence Distance

- To detect hazards, pairs of instructions must be considered
- Data is normally available after being written to register
- Can be made available for forwarding as early as the stage where it is produced
  - Stage 3 output for ALU results, stage 4 for memory fetch
- Operands normally needed in stage 2
- Can be received from forwarding as late as the stage in which they are used
  - Stage 3 for ALU operands and address modifiers, stage 4 for stored register, stage 2 for branch target



Instruction Pair Hazard Interaction

Write to Reg. File

Result Normally/Earliest available

Read from Reg. File

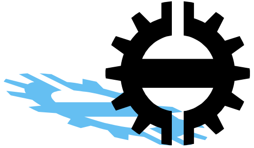
Value Normally/Latest needed

Class	N/L	alu 6/4	load 6/5	ladr 6/4	brl 6/2
alu	2/3	4/1	4/2	4/1	4/1
load	2/3	4/1	4/2	4/1	4/1
ladr	2/3	4/1	4/2	4/1	4/1
store	2/3	4/1	4/2	4/1	4/1
branch	2/2	4/2	4/3	4/2	4/1

(Column header row: Class / N/E)

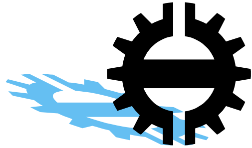
Instruction separation to eliminate hazard, Normal/Forwarded

- Latest needed stage 3 for store is based on address modifier register. The stored value is not needed until stage 4
- Store also needs an operand from ra. See Text Tbl 5.1



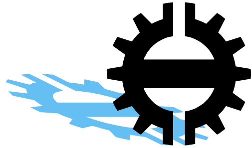
Delays Unavoidable by Forwarding

- In the Table 5.1 “Load” column, we see the value loaded cannot be available to the next instruction, even with forwarding
  - Can restrict compiler not to put a dependent instruction in the next position after a load (next 2 positions if the dependent instruction is a branch)
- Target register cannot be forwarded to branch from the immediately preceding instruction
  - Code is restricted so that branch target must not be changed by instruction preceding branch (previous 2 instructions if loaded from memory)
  - Do not confuse this with the branch delay slot, which is a dependence of instruction fetch on branch, not a dependence of branch on something else



Stalling the Pipeline on Hazard Detection

- Assuming hazard detection, the pipeline can be stalled by inhibiting earlier stage operation and allowing later stages to proceed
- A simple way to inhibit a stage is a pause signal that turns off the clock to that stage so none of its output registers are changed
- If stages 1 and 2, say, are paused, then something must be delivered to stage 3 so the rest of the pipeline can be cleared
- Insertion of nop into the pipeline is an obvious choice



Example of Detecting ALU Hazards and Stalling Pipeline

- The following expression detects hazards between ALU instructions in stages 2 and 3 and stalls the pipeline

$(\ alu3 \wedge alu2 \wedge ((ra3 = rb2) \vee (ra3 = rc2) \wedge \neg imm2\ )\ ) \rightarrow$
$(\ pause2{:}\ pause1{:}\ op3 \leftarrow 0\ ){:}$

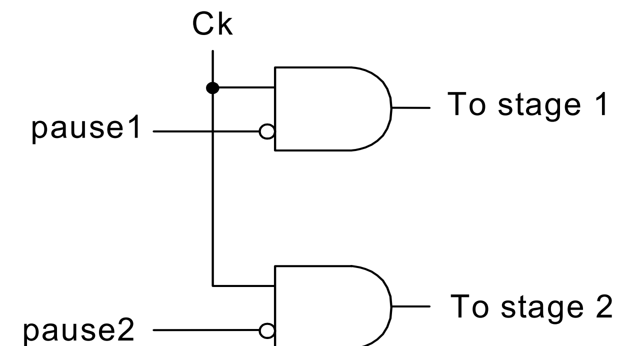
- After such a stall, the hazard will be between stages 2 and 4, detected by

$(\ alu4 \wedge alu2 \wedge ((ra4 = rb2) \vee (ra4 = rc2) \wedge \neg imm2\ )\ ) \rightarrow$
$(\ pause2{:}\ pause1{:}\ op3 \leftarrow 0\ ){:}$

- Hazards between stages 2 & 5 require

$(\ alu5 \wedge alu2 \wedge ((ra5 = rb2) \vee (ra5 = rc2) \wedge \neg imm2\ )\ ) \rightarrow$
$(\ pause2{:}\ pause1{:}\ op3 \leftarrow 0\ ){:}$

Fig 5.13 Pipeline Clocking Signals



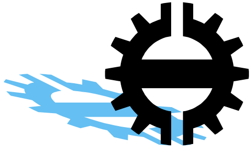
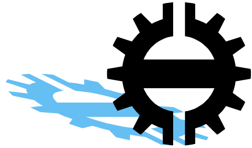


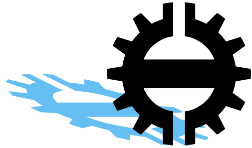
Fig 5.14 Stall Due to a Data Dependence Between Two ALU Instructions



Data Forwarding: from ALU Instruction to ALU Instruction

- The pair table for data dependencies says that if forwarding is done, dependent ALU instructions can be adjacent, not 4 apart
- For this to work, dependences must be detected and data sent from where it is available directly to X or Y input of ALU
- For a dependence of an ALU instruction in stage 3 on an ALU instruction in stage 5 the equation is

$$\text{alu5} \wedge \text{alu3} \rightarrow ((\text{ra5} = \text{rb3}) \rightarrow X \leftarrow Z5:$$
$$(\text{ra5} = \text{rc3}) \wedge \neg\text{imm3} \rightarrow Y \leftarrow Z5\ ):$$



Data Forwarding: ALU to ALU Instruction (cont'd)

- For an ALU instruction in stage 3 depending on one in stage 4, the equation is

$$\text{alu4} \wedge \text{alu3} \rightarrow ((\text{ra4} = \text{rb3}) \rightarrow X \leftarrow Z4:$$
$$(\text{ra4} = \text{rc3}) \wedge \neg\text{imm3} \rightarrow Y \leftarrow Z4\ ):$$

- We can see that the rb and rc fields must be available in stage 3 for hazard detection
- Multiplexers must be put on the X and Y inputs to the ALU so that Z4 or Z5 can replace either X3 or Y3 as inputs

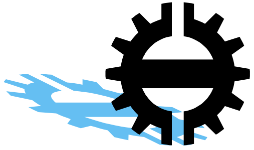
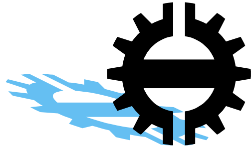


Fig 5.15 Hazard Detection and Forwarding

- Can be from either Z4 or Z5 to either X or Y input to ALU
- rb and rc needed in stage 3 for detection



Restrictions Left If Forwarding Done Wherever Possible

(1) Branch delay slot

- The instruction after a branch is always executed, whether the branch succeeds or not.

```
br r4
add . . .
• • •
```

(2) Load delay slot

- A register loaded from memory cannot be used as an operand in the next instruction.

```
ld r4, 4(r5)
nop
neg r6, r4
```

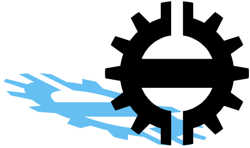
- A register loaded from memory cannot be used as a branch target for the next two instructions.

```
ld r0, 1000
nop
nop
br r0
```

(3) Branch target

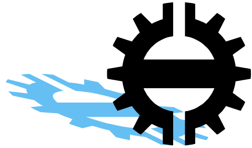
- Result register of ALU or ladr instruction cannot be used as branch target by the next instruction.

```
not r0, r1
nop
br r0
```



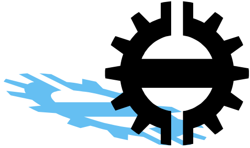
Instruction-Level Parallelism

- A pipeline that is full of useful instructions completes at most one every clock cycle
  - Sometimes called the Flynn limit
- If there are multiple function units and multiple instructions have been fetched, then it is possible to start several at once
- Two approaches are: superscalar
  - Dynamically issue as many prefetched instructions to idle function units as possible
- and Very Long Instruction Word (VLIW)
  - Statically compile long instruction words with many operations in a word, each for a different function unit



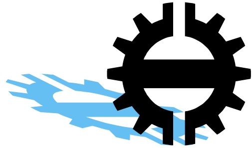
Character of the Function Units in Multiple Issue Machines

- There may be different types of function units
  - Floating-point
  - Integer
  - Branch
  - Load/store
- There can be more than one of the same type
- Each function unit is itself pipelined
- Branches become more of a problem
  - There are fewer clock cycles between branches
  - Branch units try to predict branch direction
  - Instructions at branch target may be prefetched, and even executed speculatively, in hopes the branch goes that way



Comparison of Superscalar and VLIW

- Instruction issue
  - Superscalar uses HW to detect instructions suitable to be issued in parallel
  - VLIW pushes this task to off-line SW, the compiler
- Instruction completion
  - In superscalar the instructions may complete out-of-order and have to be reordered to avoid hazards
  - In VLIW the compiler will take care of the correct order
- Predictability
  - Superscalar may form different execution bundles depending on the previous piece of software (e.g. in different calls to a subroutine)
  - VLIW execution bundles are predefined in the instruction code
- Both need high bandwidth to memory and multiport reg.file



Superscalar and VLIW Pipelines

RISC pipeline

Ifetch	Reg/Dec	Exec	Mem	Wr

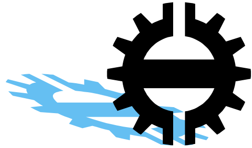
Superscalar pipeline

Ifetch	Pre-Dec	Issue/dec	Exec	Mem	Re-ord	Wr
		Issue/dec	Exec		Re-ord	Wr
		Issue/dec	Exec		Re-ord	Wr
		Issue/dec	Exec		Re-ord	Wr

VLIW pipeline

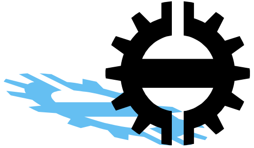
Ifetch	Reg/Dec	Exec	Mem	Wr
	Reg/Dec	Exec		Wr
	Reg/Dec	Exec		Wr
	Reg/Dec	Exec		Wr

1 data memory access unit only assumed in the pipeline diagrams



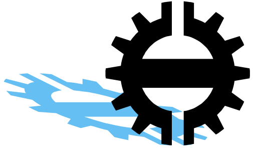
Drawbacks of Superscalar and VLIW

- Superscalar
  - Dynamic issue and reordering require complex HW
  - The number of parallel units does not scale well because of the dynamic ordering and register file problems
- VLIW
  - If orthogonality is desired, the register file complexity (# of ports) grows very quickly with the number of execution units
  - Thus, does not scale well either
  - Compatibility to single-issue RISC cannot be maintained
  - Overhead from long instruction word if variable-length execution bundles are not supported



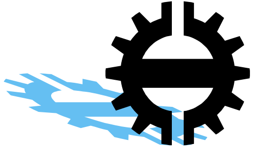
Data-Level Parallelism

- Especially in DSP/multimedia processing the data parallelism can be exploited
  - E.g. processing one 32-bit word (control or address operations), 2 16-bit words (stereo audio) or 3-4 8-bit words (RGB/YUV video) at a time in the same unit
- Often implemented with “sliced” computation units
- Can be considered as processing short vectors with a single instruction
- Implies improvement in short-vector performance without complicating the HW too much
- May cause performance degradation if the datapath is the bottleneck!
- (Conventional vector processors only save the instruction memory by applying the instruction to a block of data)



Summary

- This lecture has dealt with some alternative ways of designing a computer with a degree of parallelism
- A pipelined design is aimed at making the computer fast—target of one instruction per clock
- Forwarding, branch delay slot, and load delay slot are steps in approaching this goal
- More than one issue per clock is possible, the basic mechanisms and their drawbacks were reviewed



End of Pipelining

next we will look at control and interrupts