Control Flow Instruction

→ go to the out of sequence.
- conditional branches (make decision (if, else)) & jump (loops, Function calls)

(MIPS) beq, bne, blez, bgtz $s0, $s1, offset.

→ Use conditional branch for looping.

MIPS; Call Procedure → jump & link (jal) (MARS)
return from procedure → jump register (jr)
Argument values: $a0 - $a3
Return values: $v0

```
int main() {
    simple();
    a=b+c;
}
void simple() {
    return
}
```

```
main: jal simple
      add $s0, $s1, $s2
simple: jr $ra.
```

(return address register = $ra)

Assembly Programming

- Sequential construct
- Conditional construct
- Iterative construct.

• TRAP Instr. → OS service call.

Debugging

Conditional statement/Loops in MIPS Assembly

(if)
```
bne $s3, $s4, L1
add $s0, $s1, $s2
L1: sub $s0, $s0, $s2
```

(while)
```
while: beq ○○, done
       ~~~
       j while
done: ~
```

(if-else)
```
bne ~, L1
add ~
j done
L1: sub ——
done:
```

(for)
```
for: beq $s0 $s0 done
     addi $s0, $s0, 1
     j for
```

Arrays in MIPS

Load upper immediate + OR immediate.
(lui)? (ori)? (sll)?

Function calls

↳ Conventions:
- Caller: passes argument, jumps to callee
- Callee: perform procedure, return result to caller, return to the point of call & not overwrite

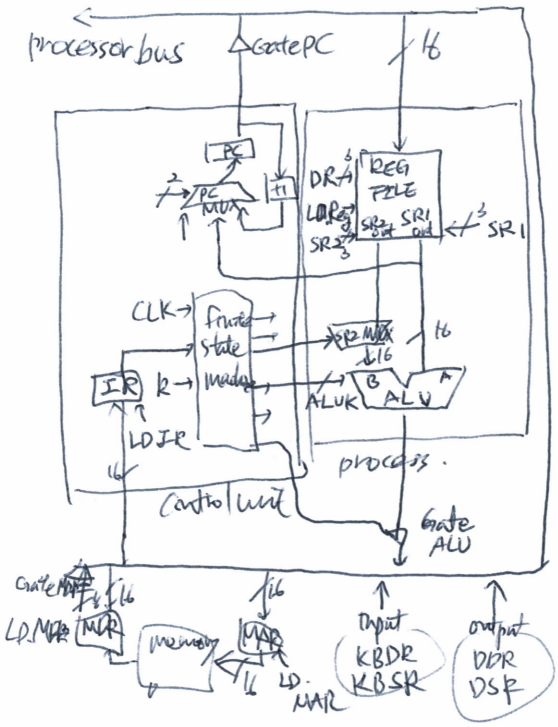
• stack: memory area used to save local variables

LIFO

• Temporary register $t0 ~ $t9 ⇒ nonpreserved reg.

Lecture 11 Microarchitecture.

Recall: LC-3 VonNeumann Architecture



under specific design constraints & goals

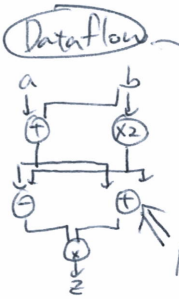
* Microarchitecture = implementation of ISA

- Von Neumann Arch = stored program computer
  - 2 keys
    - Stored program in linear mem. array
    - Sequential instruction processing
      - program counter (instruction pointer)
- Dataflow Model (of a Computer)
  - Instruction is fetched executed in dataflow order (not instruction pointer!)
  - Inherently more parallel.

VN arch

```
V <= a+b;
W <= b*2;
X <= V-W;
Y <= V+W;
Z <= X*Y;
```

vs Dataflow

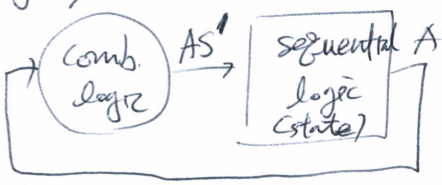


consists of dataflow nodes

Node: { add, mult / conditional / relational / Barrier synch

MIPS

- single cycle machine



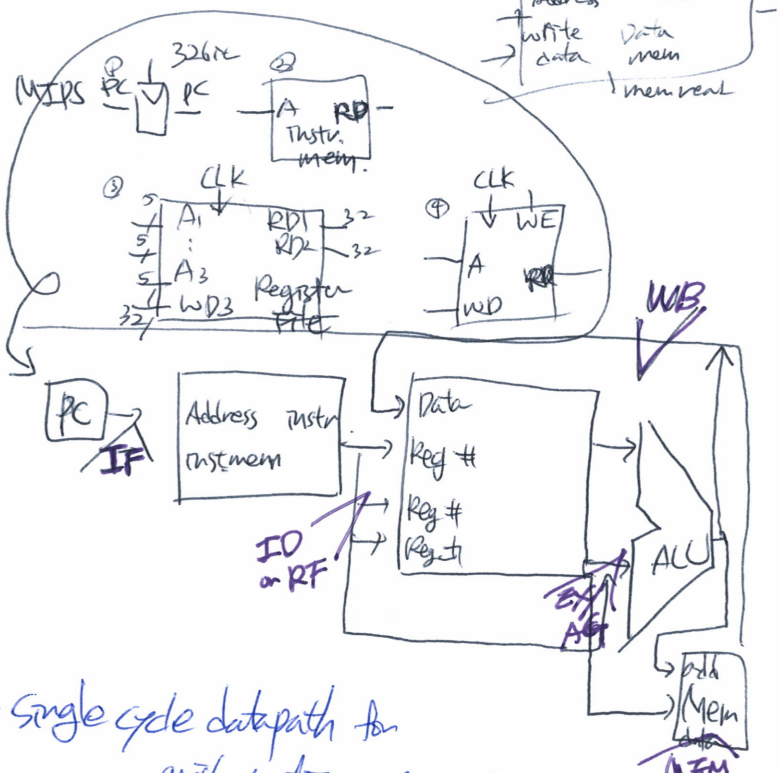
AS (Architectural State)

- multi-cycle machines
- performance analysis

↳ = {# of instructions} × {Average CPI} × {clock cycle time}
time

State elements

Single cycle machines



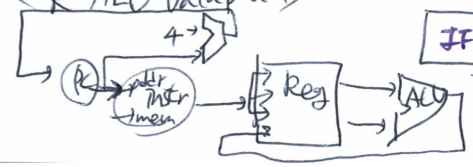
- Single cycle datapath for arithmetic & logical instructions

R-type : 3 register operands

add $S0, $S1, $S2

semantics → if MEM[PC] == add rd rs rt
GPR[rd] ← GPR[rs] + GPR[rt]
PC ← PC+4

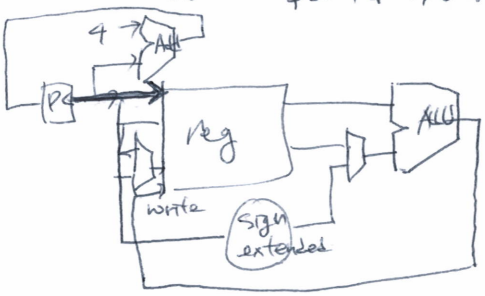
<R-ALU Datapath>



IF | ID | EX | MEM | WB

I type : 2 register operands & 1 immediate

(ex) addi $s0, $s1, 5.

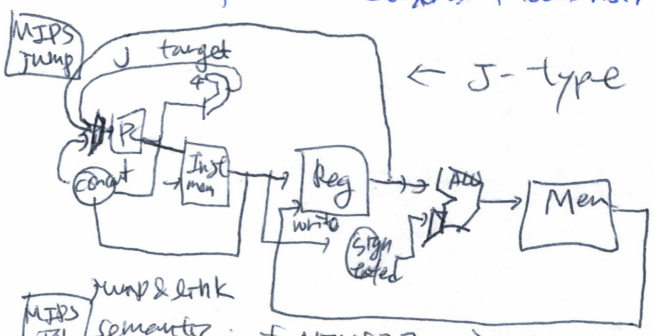


- Single cycle Datapath for Data movement Instr

MIPS load: lw $s3, 8($s0) ← I-type

MIPS store: sw $s3, 8($s0) ← I-type

- Single cycle Datapath for Control Flow Instr.



← J-type

MIPS jal: jump & link
semantics; if MEM[PC] == jal imm.
$ra ← PC+4
target = {PC[31:28], imm}
PC ← target

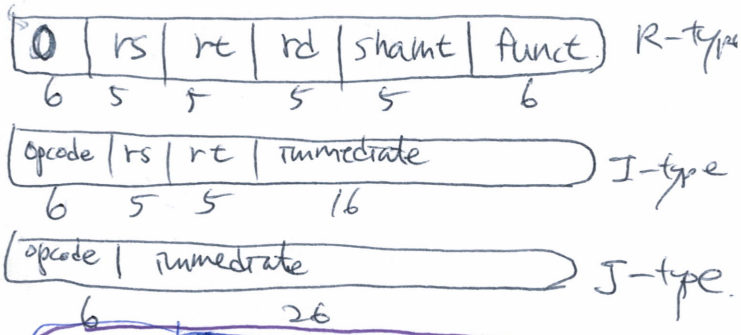
MIPS jr: jump register
semantics; if MEM[PC] == jr rs
PC ← GPR(rs)

jalr: jump & link register
if MEM[PC] == jalr rs
$ra ← PC+4
PC ← GPR(rs)

beq: beq $s0, $s1, offset ← I-type

(combine all?)

Lecture 12. Microarchitecture



- Single control signals

	De-asserted	asserted	equation
RegDes	GPR Write: rt	rd	opcode == 0
ALUSrc	2nd ALU input: 2nd GPR	imm	(opcode != 0) & (opcode != BEQ) & (opcode != BNE)
MemtoReg	ALU result → GPR write	memory → GPR write	opcode = LW
RegWrite	GPR Write disabled	GPR Write enabled	SW, Bxx, J, JR
MemRead	disabled	Mem read port → load value	LW
MemWrite	disabled	mem write enable	SW
PCSrc1	According to PCSrc2	next PC = + imm (16bit) (jump)	J, JAL
PCSrc2	next PC = PC + 4	next PC = + imm (16bit)	Bxx & beond satisfied

- ALU Control

– case opcode → '0' 'ALUI' 'LW' 'SW' 'Bxx'
↓ funct ↓ opcode ↓ add ↓ add ↓ beond

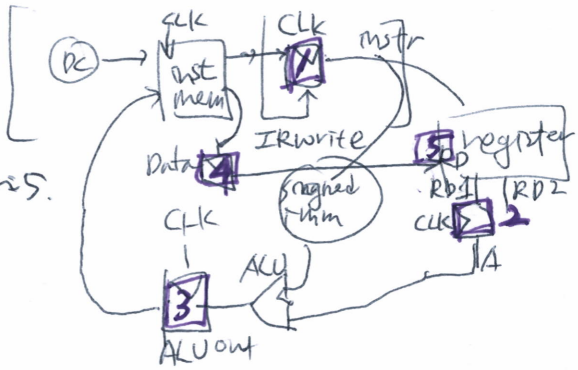
- Single Cycle Datapath Analysis: Latency

↳ the longest path matters

Multicycle Microarchitecture

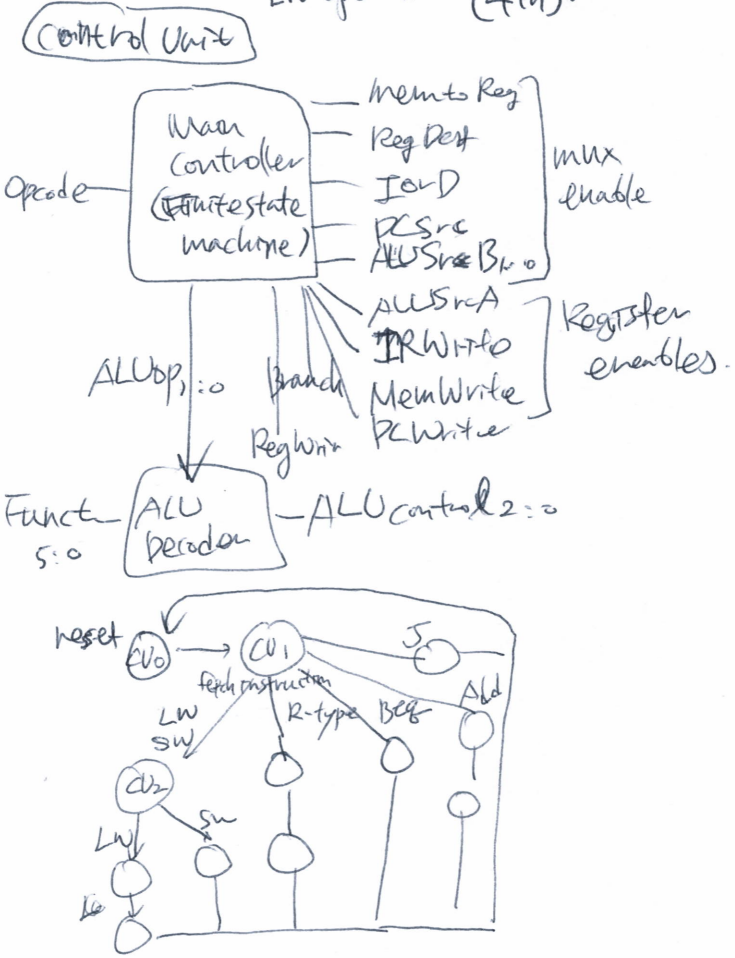
LW

Step 1 : Fetch Instruction



Step 2 ~ 5.

PC ⇒ use the same ALU to add 4. at different (3rd) cycle to LW operation (4th).



Lecture 13. Microprogramming

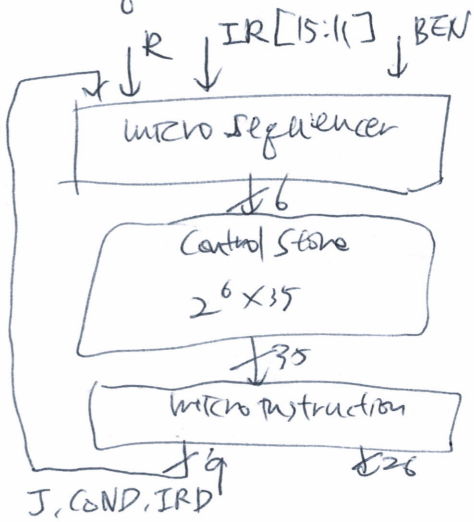
MDC

- Multicycle performance

$$T_c = t_{pcq} + t_{mux} + \max(t_{ALU} + t_{mux}, t_{mem}) + t_{setup}$$

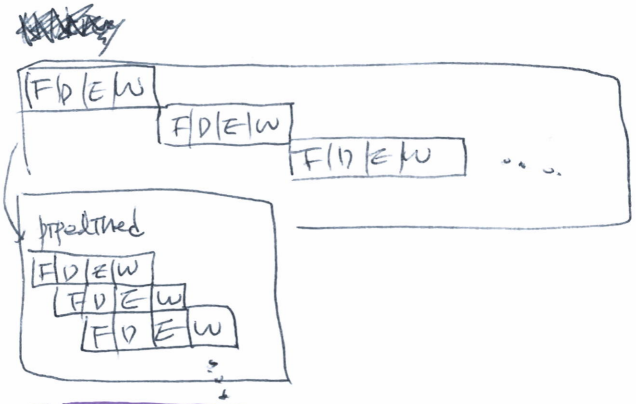
- Another example: microprogrammed multi-cycle microarchitecture

- microinstruction
- microsequencing
- Control Store
- microsequencer



Lecture 14. Pipeline

- Multi-cycle Design ⇒ Limited Concurrency
- Goal: More Concurrency → Higher instruction throughput.



Ideal Pipelining

More realistic pipelining

- Nonpipelined version with delay T

  $BW = 1/(T+S)$ where S = latch delay

- k-stage pipelined version

  $BW_{k\text{-stage}} = 1/(T/k + S)$

  $BW_{max} = 1/(1 \text{ gate delay} + S)$

Cost? $= G + L$

G: combinational cost; L: Latch

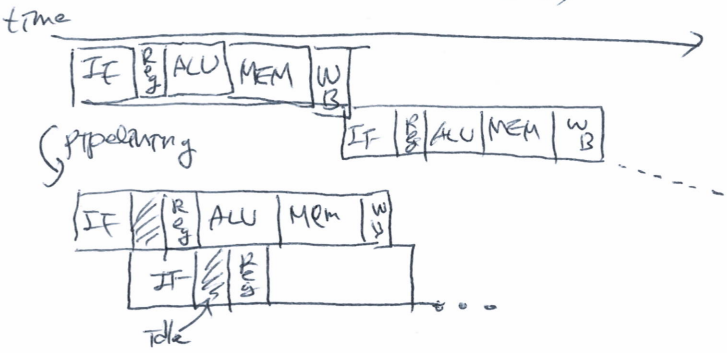
$Cost_{k\text{-stage}} = G + Lk$.

recall:

MDC (11)

Instruction Processing Cycle

1. Instruction Fetch (IF)
2. Instr. decode & register operand fetch (ID/RF)
3. Execute / Evaluate memory access (EX/AG)
4. Memory Operand fetch (MEM)
5. Store / Writeback result (WB)



lw → 5 stages.

- Control signal in pipeline
  - Option 1: decode once using the same logic as single-cycle and buffer signals until consumed
  - Option 2: carry relevant "instruction word/file" down the pipeline & decode locally within each or in a previous stage
- Non-ideals
  - Not identical operations → external fragmentation
  - Not uniform suboperations → internal fragmentation
  - Not independent operations → pipeline stalls.

- Causes of Pipeline Stalls.
  - → Resource contention = resource dependence
  - Dependency
    - Long-latency (multicycle) operation
  - → dependency (or "hazard")
    - ⇓
    - ordering requirement between instr.
      - Data dependence ← Flow dep (read→write), Output dep (write→write), Anti dep (write→read)
      - Control dependence.
  - resource contention (2 pipeline stages need the same resource)
    - → ① Duplicate resource ⑤ increase the throughput.
    - ② Detect & Stall
  - (EX) RegFile → read/write = only a half cycle

Flow dep: $r_3 \leftarrow r_1$ op r_2; $r_5 \leftarrow r_3$ op r_4 — Read after Write

Anti dep: $r_3 \leftarrow r_1$ op r_2; $r_1 \leftarrow r_4$ op r_5 — Write after Read

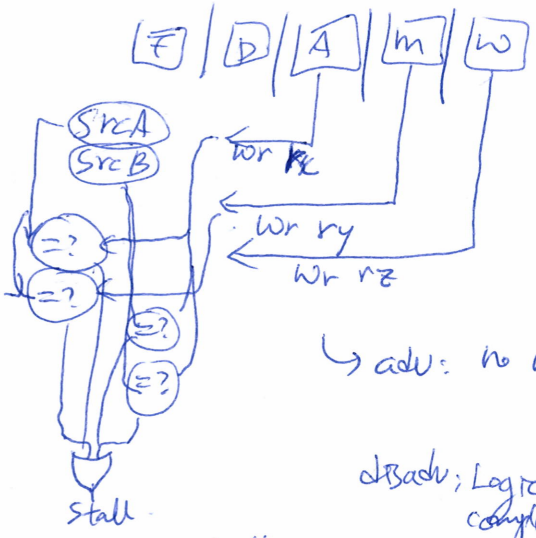
Output dep: $r_3 \leftarrow r_1$ op r_2; $r_3 \leftarrow r_5$ op r_4; $r_3 \leftarrow r_6$ op r_7 — write after write

Data Dependence handling

- Anti & output → write to the destination
- Flow dep → ① Detect & wait until reg write
  - ② Detect & forward/bypass to dependent instr.
  - ③ Detect & eliminate the dependency at the software level (= insert the independent instr between dependent instr)
  - ④ Predict & Verify the needed value(s), execute "speculatively"
- Interlocking: detection of dep. (Software / Hardware)

need hazard unit

- Dep. Detection (I); Scoreboarding & Register renaming
  - → Advantage: Simple. 1 bit reg
  - → Disadv: Need to stall all types of dep., not only flow dep.
- D. Det (II); Combinational dependence check logic.



→ adv: no need to stall on anti, output dep

disadv: Logic complexity ⇑

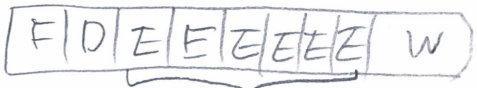
- Stall? (double stop PC & IF/ID → EN' to Fetch, Decode; (EX) insert invalid instr → CLR' to EX pipeline register)

Lecture 15. Pipelining Issues ....

- Beq (Branch) → not determine & fetch next instr → "Always Not taken" prediction
  - → misprediction penalty?
  - ⇒ Some instructions flushed when the branch is taken.
  - (&) Early Branch Resolution
    - → Adv: Reduced branch misprediction penalty → Reduced CPI
    - Disadv: potential increase in clock cycle time; additional hardware.
- Smarter Branch Prediction → (machine learning statistics...)
- Pipeline Performance

⇒ Load 25% Jump 2%
Store 10% R-types 2%
Branch 11%

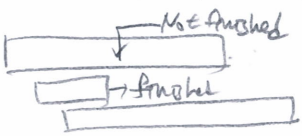
Multi-cycle Execution

FloatMult ⇒ 

while add ⇒ 

multiple execution!

↳ FMULT
ADD
FMULT



⇒ sequential semantics of ISA Not preserved

Exception vs Interrupts

- Cause → Ex → internal to the running thread (5÷0=?)
  → Int → external to the running thread
- Priority: process (exception) depends (interrupt)
- Handling context: process (exception) system (interrupt)
- Precise Exc/Int
  → ① All prev instr should be completely retired
  ② No later instr should be retired.

Solutions: ① Reorder Buffer (ROB)
↳ Content addressable memory (CAM)
Register renaming w/ ROB.

- In-Order Pipeline w/ Reorder Buffer.

(In-order dispatch/execution
out-of-order completion → write to ROB.
In-order retirement) ← none ← check for exception
↓ else flush pipeline + start from exception handler.

ROB Adv = { simple
eliminate false dependence
disadv = Use CAM or indirection to access to ROB
→ latency↑, complexity↑

② History Buffer
③ Future file
④ Checkpoint

Lecture 15. Out-of Order (OOO) Execution

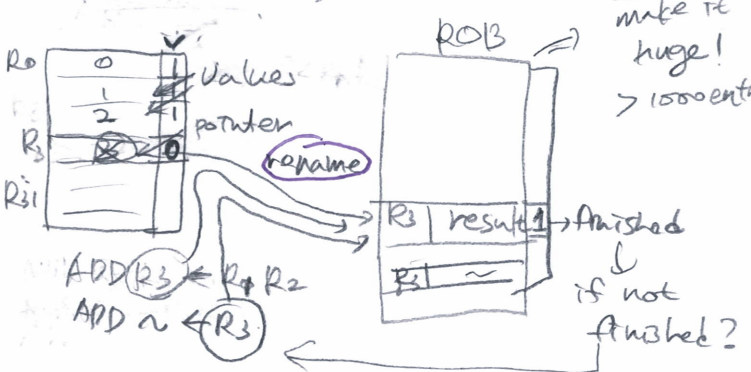
~~Dataflow, Superscalar Execution~~

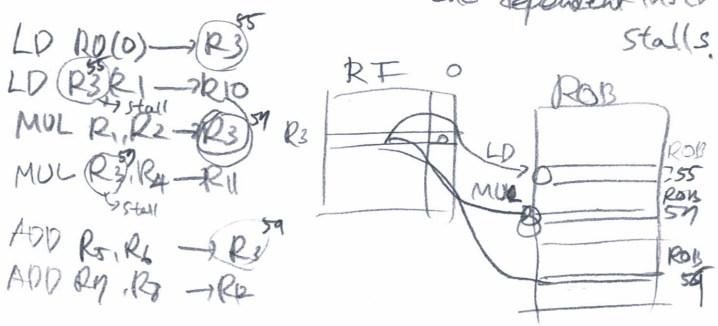
OOO Execution (Dynamic Instr. Schedule)

~~MUL R3 ← R1, R2~~
~~ADD R5 ← R3, R4~~

Lecture 16. Out-of Order Execution (Dynamic Instr. Schedule)

* Register renaming
RAT (Reg. Alias table)





Reorder Buffer Tradeoffs

* In-Order pipeline.

- Dispatch: Act of sending an instr to a func unit.
- Renaming w/ ROB eliminates stalls due to false dependencies
- Problem; True data dep → stalls dispatch of younger instr into functional unit.

* Can we do better?
↳ Preventing Dispatch Stall?
↳ Out of order dispatch (scheduling, or execution)
Idea ~ Dataflow; fetch & fire when instr is ready