EE282
Computer Architecture

Lecture 4:
Microarchitecture (Part 1)

October 9, 2001

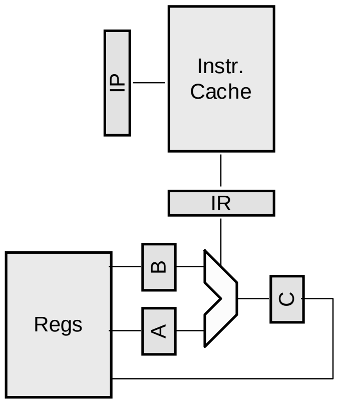
Marc Tremblay
Stanford University
marctrem@csl.stanford.edu

Agenda

- Cost of structures (area and delay)
- DLX Instruction set
- DLX Implementation
  - Simple implementation
  - Instruction flow
  - Control signals, RTL code
  - “Playing” an ADD
- Microcode
- **Pipelining**
- Hazards

Microarchitecture

- High-level implementation of an instruction set
- Datapath resources
  - physical registers
  - memory arrays
  - arithmetic units
  - interconnecting buses and switches
- Control
  - sequencing of register transfers



Building Blocks

- Arithmetic Units
  - adders, multipliers, dividers, shifters, ...
- Single Registers
- Register Files
- Memory Arrays
- Multiplexers
- Wires

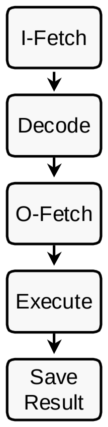
- Microarchitecture involves trading off area and delay of alternative organizations
- Units
  - area - tracks^2 (χ^2)
  - delay - 4-load inverter delay (τ_4)
- Typically organized into datapaths
  - 10-20 tracks per bit slice
- Hard to estimate cost of control logic

Cost of Typical Building Blocks

Unit	Area	Delay
Single Register	40 χ^2/bit	2 τ_4
Register File	$(P+6)^2$ χ^2 /bit	$\log_4(W\times B)+4$ τ_4
Fast Adder	200 χ^2 /bit	$3 \times \log_4(B)+ 5$ τ_4
Multiplier	$200B+30B^2$ χ^2	$5 \times \log_4(B)+ 5$ τ_4
1000χ Wire		1 τ_4

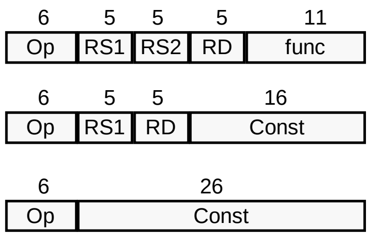
Instruction Execution

- 5 basic steps
  - fetch instruction (IF) (F)
  - decode instruction (ID) (D)
  - fetch operands (OF) (R)
  - execute (EX) (X)
  - store result (RS) (W)



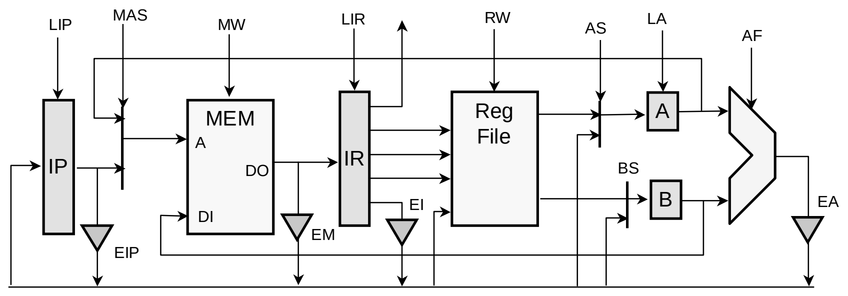
The DLX ISA (less FP)

- 31 General Registers
  - R1-R31
  - R0 always 0
- Data types
  - byte, half-word, word
- Addressing modes
  - immediate
  - displacement
- Opcodes
  - ALU ops
    - ADD, SUB,…
  - Load (LW, LH, LB,…)
  - Store (SW, SH, SB,…)
  - Control (BEQZ, J, JR, JAL,…)



A Simple DLX Implementation

- Single bus
- Single ALU
- Sequential execution of an instruction



RTL Sequences

```
ADD:
   IR <- M[IP], A <- BUS <- IP ;
   A <- R[IR.RS1], B <- R[IR.RS2], IP <- BUS <- A + 4 ;
   R[IR.RD] <- BUS <- A + B ;

JUMP:
   IR <- M[IP], A <- BUS <- IP ;
   IP,A <- BUS <- A + 4 ; // could load B here if 2nd bus
   B <- BUS <- IR.CONST26 ;
   IP <- BUS <- A + B ;

BEQZ:
   IR <- M[IP], A <- BUS <- IP ;
   IP,B <- BUS <- A + 4, A <- R[IR.RS1] ;
   if(Z){  A <-BUS <- IR.CONST16 ;
           IP <- A + B ; }
```

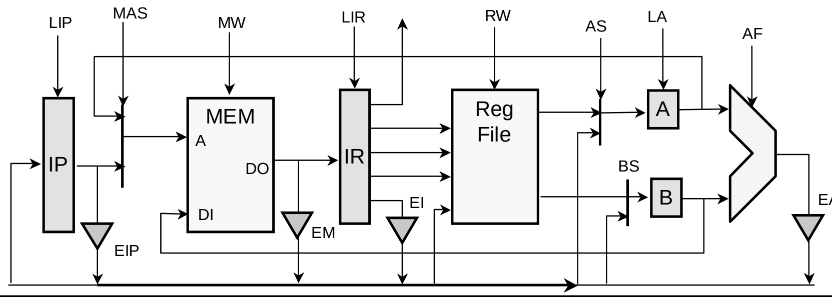
RTL Sequences (2)

```
LOAD:
   IR <- M[IP], A <- BUS <- IP ;
   A <- R[IR.RS1], IP <- BUS <- A + 4 ; // could load B
   B <- BUS <- IR.CONST16 ;
   A <- A + B ;
   R[IR.RD] <- BUS <- M[A] ;

STORE:
   IR <- M[IP], A <- BUS <- IP ;
   A <- R[IR.RS1], IP <- BUS <- A + 4 ; // could load B
   B <- BUS <- IR.CONST16 ;
   A <- A + B, B <- R[IR.RD] ;
   M[A] <- B ;
```

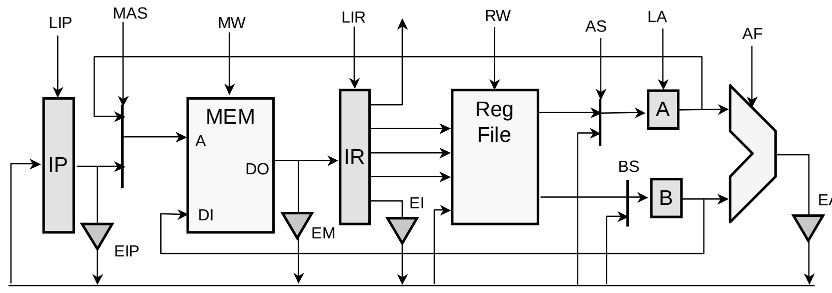
ADD The Movie

```
IR <- M[IP], A <- BUS <- IP ;
A <- R[IR.RS1], B <- R[IR.RS2], IP <- BUS <- A + 4 ;
R[IR.RD] <- BUS <- A + B ;
```



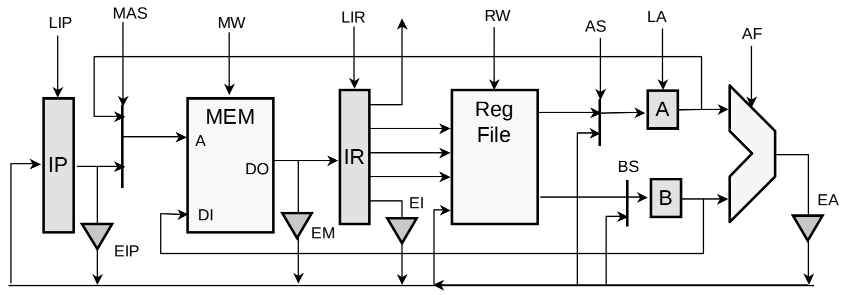
ADD The Movie (2)

```
IR <- M[IP], A <- BUS <- IP ;
A <- R[IR.RS1], B <- R[IR.RS2], IP <- BUS <- A + 4 ;
R[IR.RD] <- BUS <- A + B ;
```

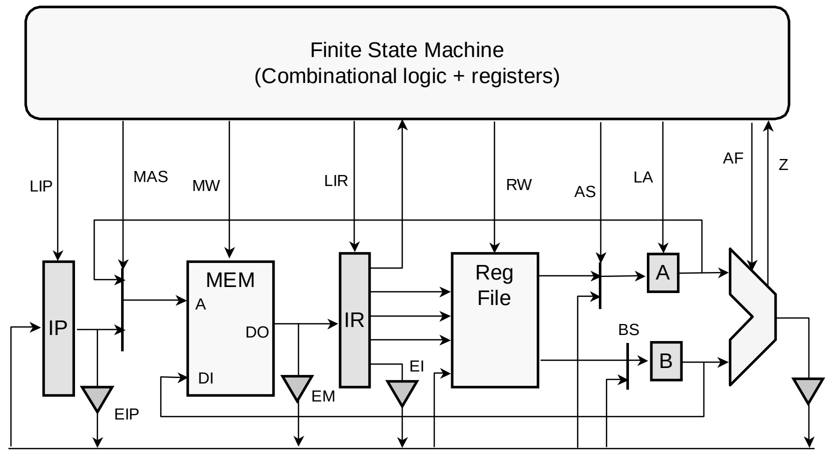


ADD The Movie (3)

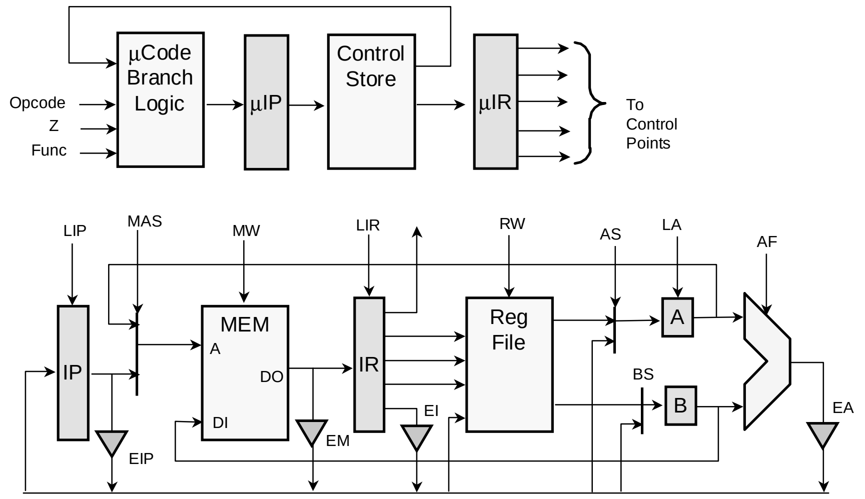
```
IR <- M[IP], A <- BUS <- IP ;
A <- R[IR.RS1], B <- R[IR.RS2], IP <- BUS <- A + 4 ;
R[IR.RD] <- BUS <- A + B ;
```



Hardwired Control



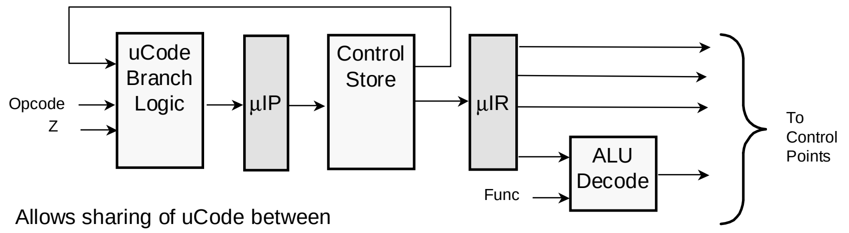
Microcoded Control



Microcode

	BUS	MAS	MW	LIP	LIR	RW	A	B	AF	NEXT
F1	IP	0	0	0	1	0	BUS	-	-	D1
D1	ALU	0	0	1	0	0	R	R	A+4	DECODE
A1	ALU	0	0	0	0	1	-	-	A+B	F1
J1	IP	0	0	0	0	0	BUS	0		J2
J2	C26	0	0	0	0	0	A	BUS		J3
J3	ALU	0	0	1	0	0	-	-	A+B	F1
B1	IP	0	0	0	0	0	BUS	0		Z(F1,B2)
B2	C16	0	0	0	0	0	A	BUS		B3
B3	ALU	0	0	1	0	0	-	-	A+B	F1
L1	C16	0	0	0	0	0	A	BUS		L2
L2	ALU	0	0	0	0	0	BUS	-	A+B	L3
L3	MEM	1	0	0	0	1	-	-	-	F1
S1	C16	0	0	0	0	0	A	BUS	-	S2
S2	ALU	0	0	0	0	0	BUS	-	A+B	S3
S3	-	1	1	0	0	0	-	-	-	F1

Factoring Microcode



Allows sharing of uCode between

ADD, SUB, AND, OR, …

Still need different code for ADDI, etc…

Can you factor that?

Continuum between uCode and hardwired

Reservation Tables

ADD	STATE		
	F1	D1	A1
BUS	X	X	X
MEM	X		
ALU		X	X

JUMP	STATE				
	F1	D1	J1	J2	J3
BUS	X	X	X	X	X
MEM	X				
ALU		X			X

Show resource usage as a function of time (state)

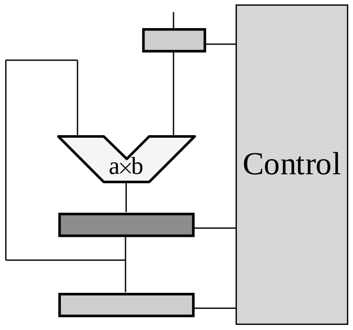
Identify bottlenecks (bus)

Identify opportunities to overlap execution

High-performance Datapaths

- Datapath + control FSM can do complex sequential computations.
- E.g. - Compute kX^4
  - requires 4 clocks on this machine
  - only uses 1 multiplier
  - one result every 4 clocks

How do we increase performance?



Increasing throughput

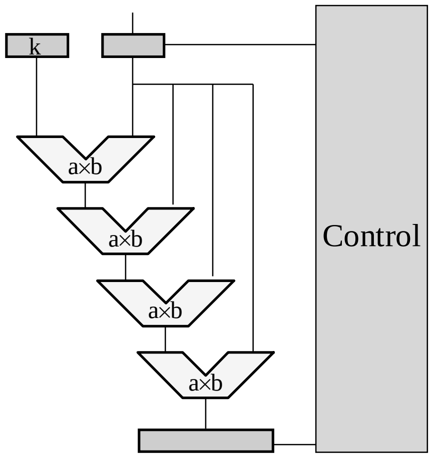
Instead of:

1 multiplier - N cycles

N multipliers - 1 cycle

Problem:

Cycle time ~n times longer

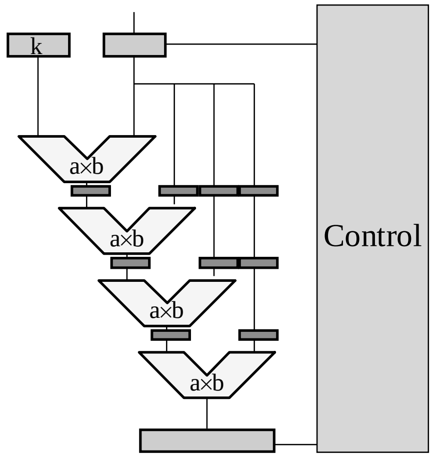


Pipelining

Instead of:

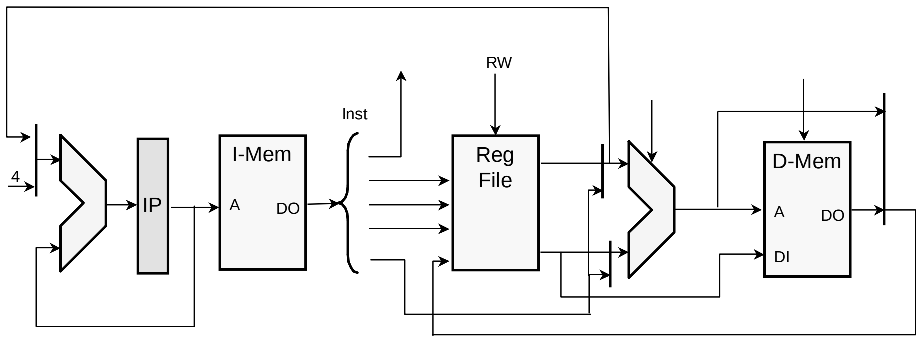
1 multiplier - N cycles
or
N multipliers - 1 cycle

N multipliers - N cycles latency
1 cycle throughput



A Combinational Implementation

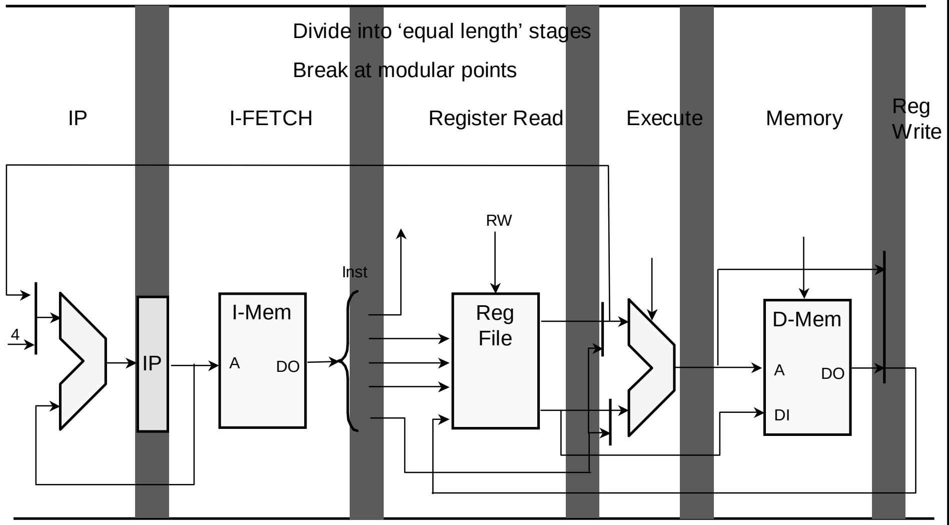
No shared resources
No hidden state



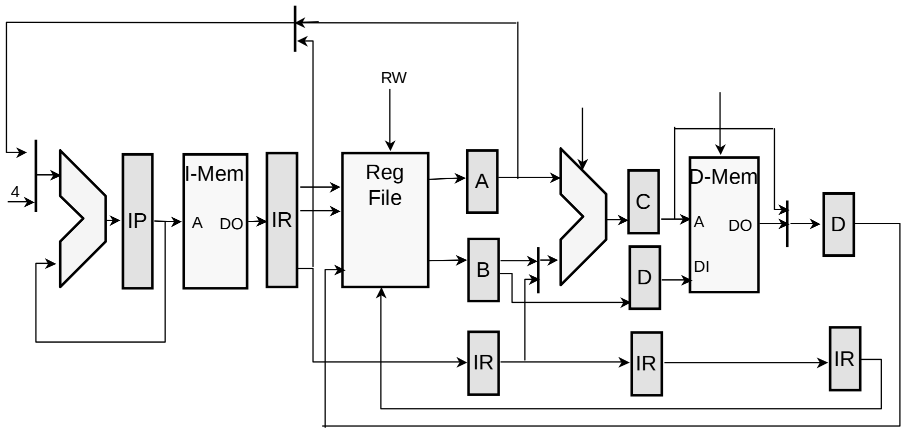
Pipeline by adding a register every t ns.

Divide into ‘equal length’ stages

Break at modular points



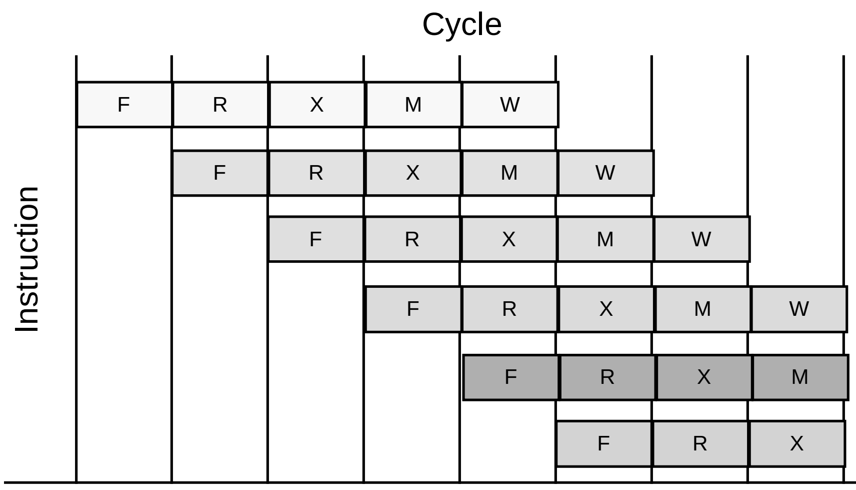
Pipelined Implementation



Pipelining Rules

- Forward travelling signals at each stage are latched
- Only perform logic on signals in the same stage
  - signal labeling useful to prevent errors,
    - e.g., IR_{RR}, IR_{EX}, IR_{M}, IR_{WB}
- Backward travelling signals at each stage represent *hazards*

Pipeline Diagrams



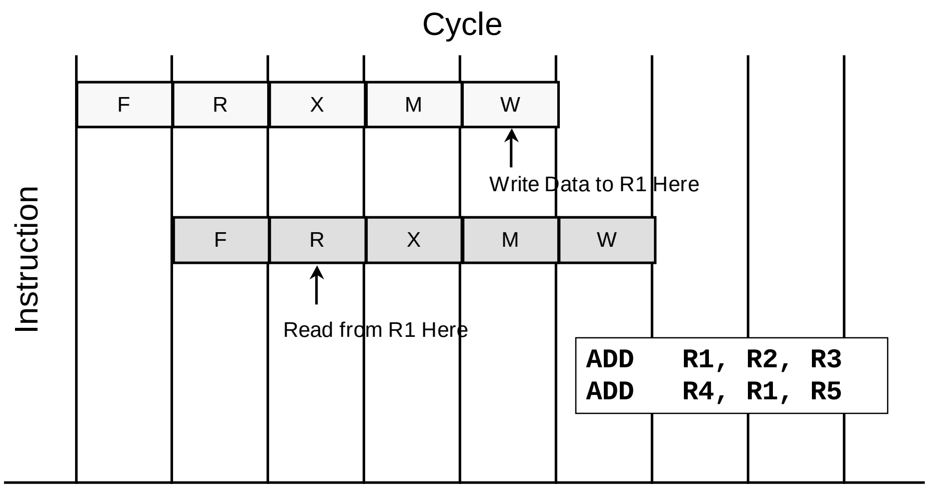
Pipeline Hazards

- Data hazards
  - an instruction uses the result of a previous instruction (RAW)

    ADD R1, R2, R3
    ADD R4, R1, R5
- Control hazards
  - the location of an instruction depends on a previous instruction

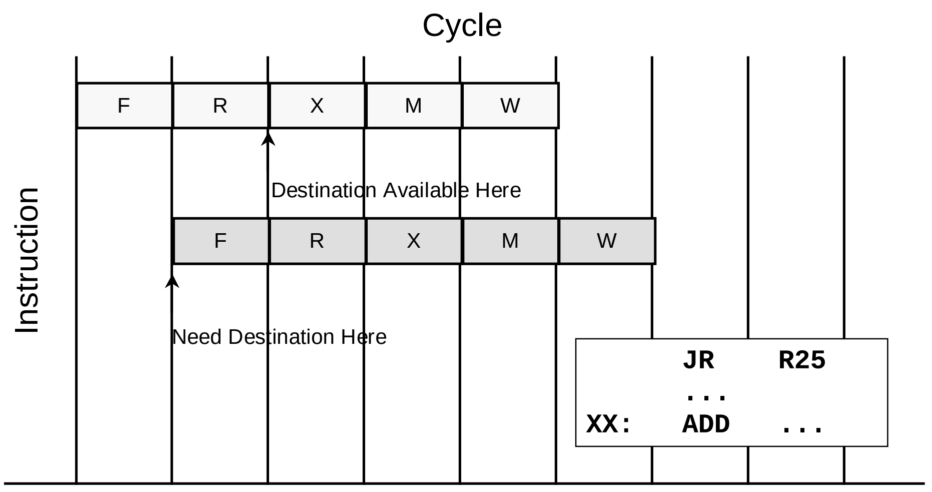
    ```
            JMP    LOOP
            …
    LOOP:   ADD    R1, R2, R3
    ```
- Structural hazards
  - two instructions need access to the same resource
    - collision in reservation table

Data Hazards (RAW)



```
ADD   R1, R2, R3
ADD   R4, R1, R5
```

Control Hazards



```
        JR    R25
        ...
XX:     ADD   ...
```

Next Time

- Pipelining
  - multi-cycle operations
  - data hazards
    - RBW, WBR, WBW
  - control hazards
  - exceptions
- Dealing with hazards
  - stall
  - squish
  - bypass
- Branch Prediction