Instruction Pipelining

Suvinay Subramanian

(adapted from prior 6.823 offerings)

Single-cycle CPU

PCen
PCSrc
RegWrite & Wen
MemWrite & Wen
WBSrc
0x4
Add
Add
clk
PC
clk
clk
IR
31
we
rs1
rs2
rd1
ws
wd
rd2
GPRs
Imm Ext
ALU
z
ALU Control
clk
we
addr
rdata
Data Memory
wdata

IRen OpCode RegDst ExtSel OpSel BSrc zero? AddrSrc

Can we do better?

- Issue: Limited concurrency

- Several hardware resources
  Not all of them utilized at the same time

Pipelining

- Basic Idea: Divide instruction execution into distinct stages of processing

Five stages of instruction execution:
1. Instruction Fetch
2. Instruction Decode
3. Execute
4. Memory Access
5. Writeback

Pipelining: Tradeoffs

- Advantage:
  - Improves instruction throughput (allows us to increase frequency)
- Cost:
  - Additional hardware
  - Complexity

Ideal Pipeline

- Repetition of identical operations
- Repetition of independent operations
- Balanced stages

Pipelining: Issues

1. Balanced stages

2. Correctness
   - Handling dependences
   - Handling resource contention
   - Advanced: Exceptions, interrupts

Pipeline Hazards

1. Structural
2. Data
3. Control

Pipeline Hazards

Structural Hazard:

- An instruction in the pipeline may need a resource being used by another instruction in the pipeline

Example 1: Memory in Princeton-style architecture

Example 2: Registers in register file?

- Can we read multiple registers simultaneously?

Pipeline Hazards

Data Hazard:

– Dependency between instructions

Example: Read-after-write (RAW)

r4 <- r1 + 5

r3 <- r4 + 10

Pipeline Hazards

Control Hazard:

– Flow of execution depends on previous instruction

Example: Branches

Handling Pipeline Hazards: Recipe

1. Stall

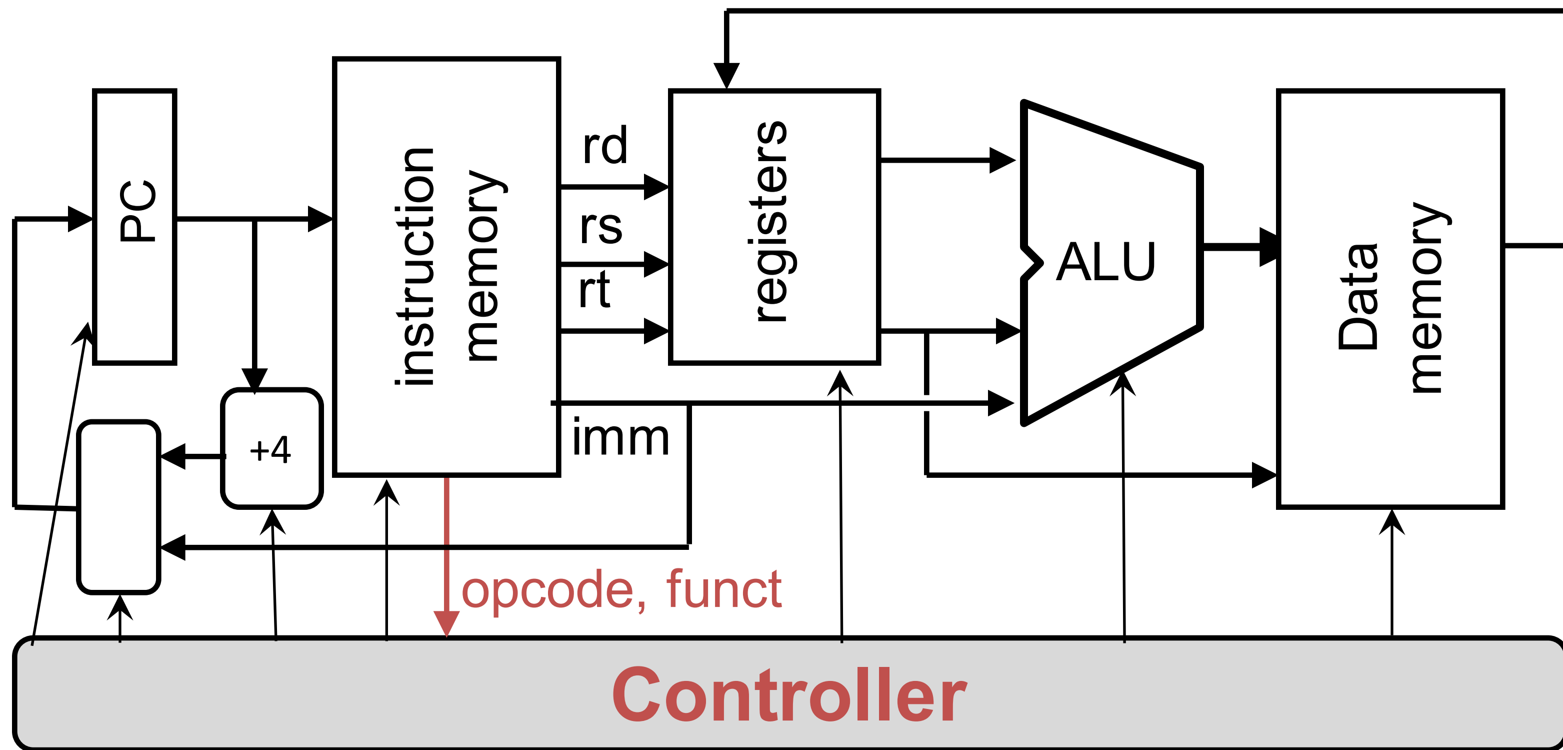
2. Bypass

3. Speculate

Terminology

- Data path:
  Part of processor that contains hardware necessary to perform operations

- Control path:
  Part of processor that tells the data path what needs to be done



1. Instruction Fetch

2. Decode/ Register Read

3. Execute

4. Memory

5. Register Write

Mechanics of Processor Design

- Analyze ISA
  - Determine data path requirements
- Select components of data path
  - ALU, Register file etc.
- Analyze implementation of instruction
  - Do we need to bypass, stall, speculate?
  - Assemble control signals

Analyzing the ISA

R-type:

op	rs	rt	rd		func

I-type:

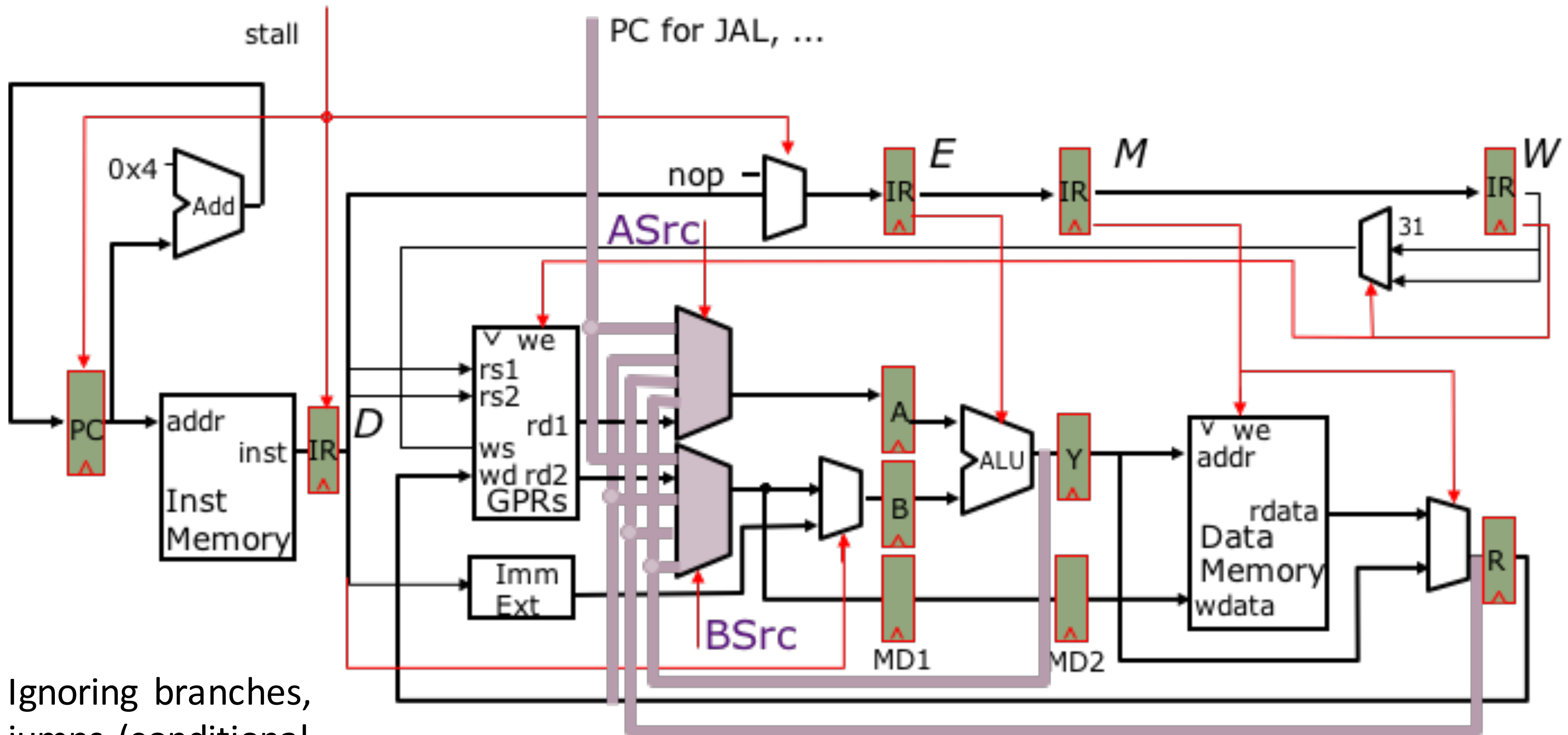
op	rs	rt	immediate16

J-type:

op	immediate26

		source(s)	*destination*
ALU	rd ← (rs) func (rt)	rs, rt	rd
ALUi	rt ← (rs) op imm	rs	rt
LW	rt ← M [(rs) + imm]	rs	rt
SW	M [(rs) + imm] ← (rt)	rs, rt	
BZ	*cond* (rs)		
	true: PC ← (PC) + imm	rs	
	false: PC ← (PC) + 4	rs	
J	PC ← (PC) + imm		
JAL	r31 ← (PC), PC ← (PC) + imm		31
JR	PC ← (rs)	rs	
JALR	r31 ← (PC), PC ← (rs)	rs	31

Fully Bypassed Data Path



Ignoring branches, jumps (conditional hazards)

$$\text{stall} = (rs_D = ws_E).(opcode_E = LW_E).(ws_E \neq 0).re1_D + (rt_D = ws_E).(opcode_E = LW_E).(ws_E \neq 0).re2_D$$

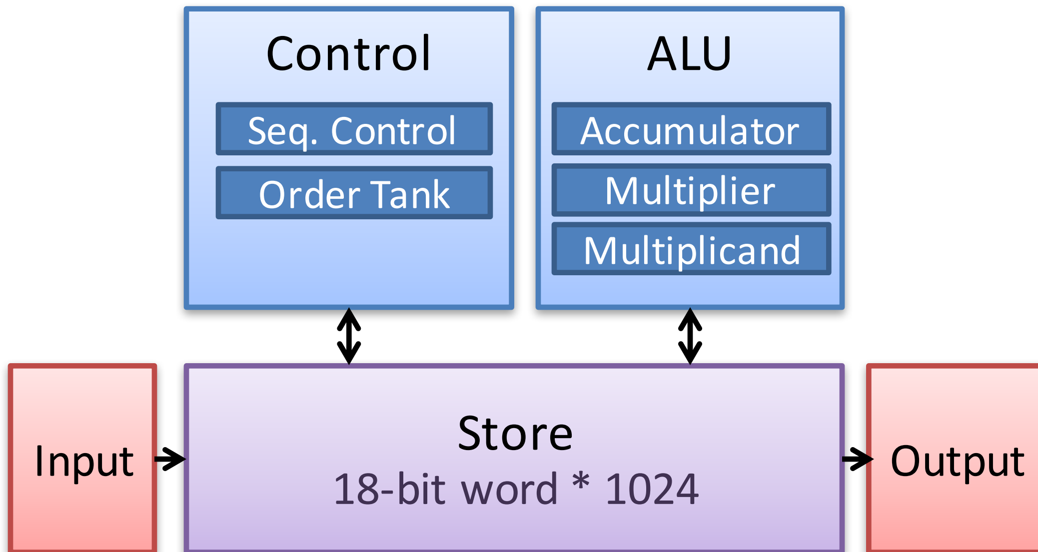
Conclusion

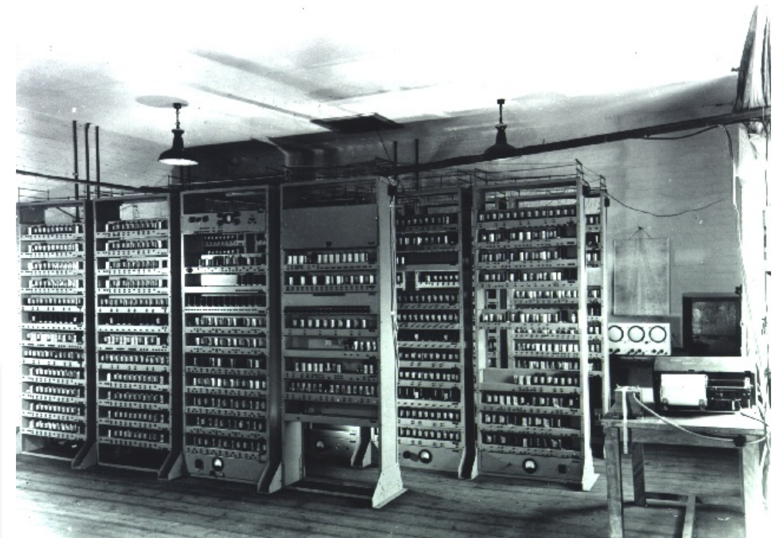
- Pipelining increases throughput by overlapping execution of multiple instructions in different stages
- Pipeline stages should be balanced for highest clock frequency
- Hazards limit performance:
  - Structural, data, control
- Solutions for hazards: Stall, bypass, speculate

Revisiting Self-Modifying Code

Cambridge EDSAC (1949)

- **EDSAC:** Electronic Delay Storage Automatic Calculator





- Single accumulator + absolute addressing of memory
- Typical execution time: ~600 simple commands/sec

Ref1: http://en.wikipedia.org/wiki/Electronic_Delay_Storage_Automatic_Calculator
Ref2: http://www.dcs.warwick.ac.uk/~edsac/Software/EdsacTG.pdf

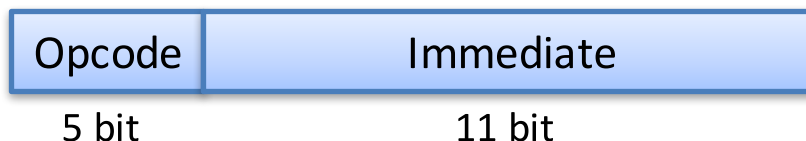
EDSACjr

- A simplified version of EDSAC instruction set

Opcode	Description	Bit Representation
ADD n	Accum $\leftarrow$ Accum + M[n]	00001 n
SUB n	Accum $\leftarrow$ Accum - M[n]	10000 n
STORE n	M[n] $\leftarrow$ Accum	00010 n
CLEAR	Accum $\leftarrow$ 0	00011 00000000000
OR n	Accum $\leftarrow$ Accum | M[n]	00000 n
AND n	Accum $\leftarrow$ Accum & M[n]	00100 n
SHIFTR n	Accum $\leftarrow$ Accum shiftr n	00101 n
SHIFTL n	Accum $\leftarrow$ Accum shiftl n	00110 n
BGE n	If Accum ≥ 0 then PC $\leftarrow n$	00111 n
BLT n	If Accum < 0 then PC $\leftarrow n$	01000 n
END	Halt machine	01010 00000000000

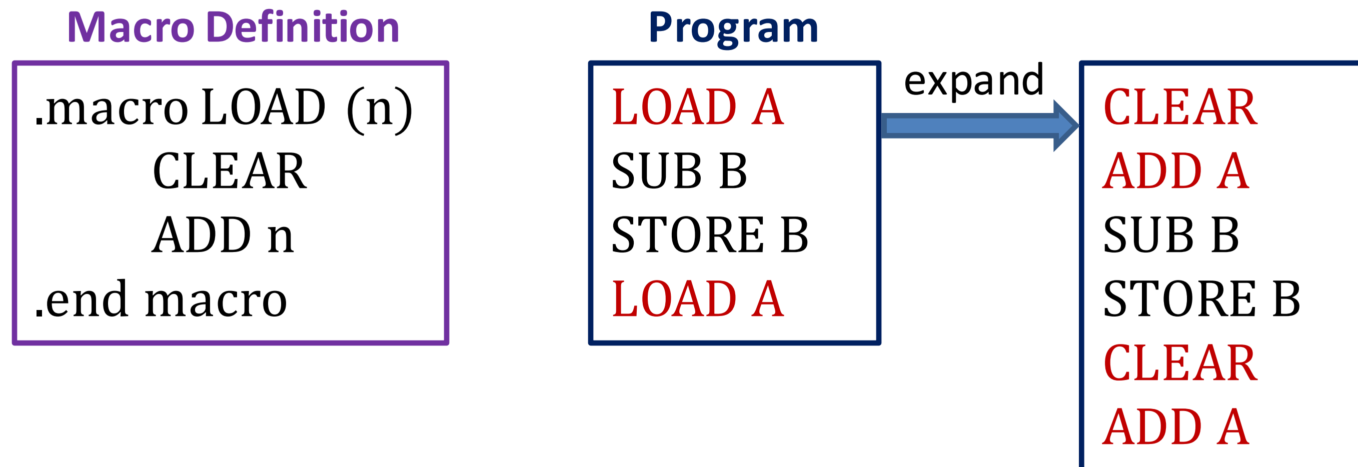
M[x]: the contents of the memory location addressed by x

Instruction Format:



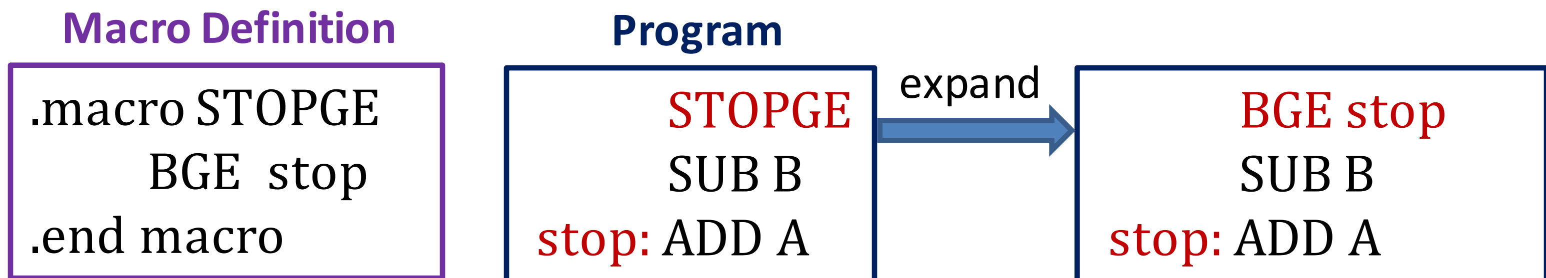
Programming with EDSACjr

- **Macro:** a sequence of instructions defined with a short name and is expanded in place at assembly time
- A macro is not the same as a subroutine or function
- Example:



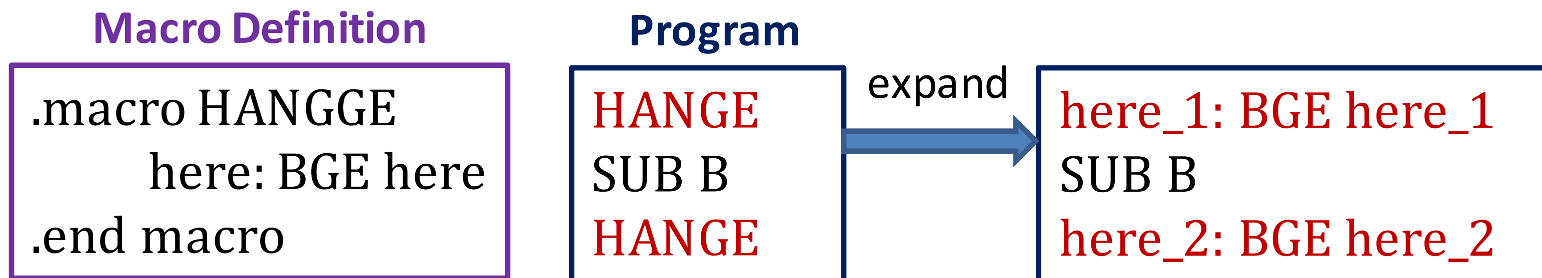
Programming with EDSACjr

- **Global label:** a label defined outside the macro
- Global labels are useful for accessing commonly used memory locations
- Example:



Programming with EDSACjr

- **Local label:** a label defined within the macro
- Local labels will be replaced by unique label names during expansion
- Example1:



Programming with EDSACjr

- **Local label:** a label defined within the macro
- Local labels will be replaced by unique label names during expansion
- Example2:

Macro Definition

```
.macro ADDGE n
            BLT cont
            ADD n
      cont:
.end macro
```

Program

```
ADDGE A
SUB B
```

expand →

```
         BLT cont_1
         ADD A
cont_1:  SUB B
```

Self Modifying Code Example

Memory

Label	Contents	Address
	......	0x0000
I :	i	0x0008
J :	j	0x000a
	......	
A :	A[0][0]	0x0012
	A[0][1]	0x0014
	......	
	A[2^a-1][2^a-2]	
	A[2^a-1][2^a-1]	
	......	
B :	0	0x0040
ADDR_A :	0x0012	0x0042
TMP_1 :	0	0x0044

Global variables

```
_store_op:  STORE 0   ; STORE template
_bge_op:    BGE 0     ; BGE template
_blt_op:    BLT 0     ; BLT template
_add_op:    ADD 0     ; ADD template
```

Pseudo-code:

B = A[i][j] + A[j][i]

A is a 2^a by 2^a matrix where a is a constant.

Self Modifying Code Example

Memory

Label	Memory	Address
	......	0x0000
I :	i	0x0008
J :	j	0x000a
	......	
A :	A[0][0]	0x0012
	A[0][1]	0x0014
	......	
	A[2^a-1][2^a-2]	
	A[2^a-1][2^a-1]	
	......	
B :	0	0x0040
ADDR_A :	0x0012	0x0042
TMP_1 :	0	0x0044

Pseudo-code:

B = A[i][j] + A[j][i]

Step 1: Load A[i][j]

```
CLEAR
ADD      I
```

Calculate address offset: 2^a*i + j

i

Accumulator

Self Modifying Code Example

Memory

	......	0x0000
I :	i	0x0008
J :	j	0x000a
	......	
A :	A[0][0]	0x0012
	A[0][1]	0x0014
	......	
	A[2^a-1][2^a-2]	
	A[2^a-1][2^a-1]	
	......	
B :	0	0x0040
ADDR_A :	0x0012	0x0042
TMP_1 :	0	0x0044

Pseudo-code:

B = A[i][j] + A[j][i]

Step 1: Load A[i][j]

```
CLEAR
ADD      I
SHIFTL   a
```

Calculate address offset: 2^a*i + j

2^a*i

Accumulator

Self Modifying Code Example

Memory

Label	Contents	Address
	......	0x0000
I :	i	0x0008
J :	j	0x000a
	......	
A :	A[0][0]	0x0012
	A[0][1]	0x0014
	......	
	A[2^a-1][2^a-2]	
	A[2^a-1][2^a-1]	
	......	
B :	0	0x0040
ADDR_A :	0x0012	0x0042
TMP_1 :	0	0x0044

Pseudo-code:

B = A[i][j] + A[j][i]

Step 1: Load A[i][j]

```
CLEAR
ADD      I
SHIFTL   a
ADD      J
```

Calculate address offset: 2^a*i + j

2^a*i+j

Accumulator

Self Modifying Code Example

Memory

	......	0x0000
I :	i	0x0008
J :	j	0x000a
	......	
A :	A[0][0]	0x0012
	A[0][1]	0x0014
	......	
	A[2^a-1][2^a-2]	
	A[2^a-1][2^a-1]	
	......	
B :	0	0x0040
ADDR_A :	0x0012	0x0042
TMP_1 :	0	0x0044

Pseudo-code:

B = A[i][j] + A[j][i]

Step 1: Load A[i][j]

```
CLEAR
ADD       I
SHIFTL    a
ADD       J
ADD       ADDR_A
```

address of A[i][j]

0x0012+2^a*i+j

Accumulator

Self Modifying Code Example

Memory

	......	0x0000
I :	i	0x0008
J :	j	0x000a
	......	
A :	A[0][0]	0x0012
	A[0][1]	0x0014
	......	
	A[2^a-1][2^a-2]	
	A[2^a-1][2^a-1]	
	......	
B :	0	0x0040
ADDR_A :	0x0012	0x0042
TMP_1 :	0	0x0044

Pseudo-code:

B = A[i][j] + A[j][i]

Step 1: Load A[i][j]

```
CLEAR
ADD      I
SHIFTL   a
ADD      J
ADD      ADDR_A
ADD      _add_op
```

_add_op: ADD 0

ADD (0x0012+2^a*i+j)

Accumulator

Self Modifying Code Example

Memory

	......	0x0000
I :	i	0x0008
J :	j	0x000a
	......	
A :	A[0][0]	0x0012
	A[0][1]	0x0014
	......	
	A[2^a-1][2^a-2]	
	A[2^a-1][2^a-1]	
	......	
B :	0	0x0040
ADDR_A :	0x0012	0x0042
TMP_1 :	0	0x0044

Pseudo-code:

B = A[i][j] + A[j][i]

Step 1: Load A[i][j]

```
      CLEAR
      ADD     I
      SHIFTL  a
      ADD     J
      ADD     ADDR_A
      ADD     _add_op
      STORE   _L1

_L1:  CLEAR
```

ADD (0x0012+2^a*i+j)

Accumulator

Self Modifying Code Example

Memory

Label	Contents	Address
	......	0x0000
I :	i	0x0008
J :	j	0x000a
	......	
A :	A[0][0]	0x0012
	A[0][1]	0x0014
	......	
	A[2^a-1][2^a-2]	
	A[2^a-1][2^a-1]	
	......	
B :	0	0x0040
ADDR_A :	0x0012	0x0042
TMP_1 :	0	0x0044

Pseudo-code:

B = A[i][j] + A[j][i]

Step 1: Load A[i][j]

```
      CLEAR
      ADD      I
      SHIFTL   a
      ADD      J
      ADD      ADDR_A
      ADD      _add_op
      STORE    _L1
      CLEAR
_L1:  CLEAR    ADD (0x0012+2^a*i+j)
```

(In the code above, the second CLEAR is highlighted in red; the CLEAR at _L1 is struck through and replaced by "ADD (0x0012+2^a*i+j)".)

0

Accumulator

Self Modifying Code Example

Memory

Label	Contents	Address
	......	0x0000
I :	i	0x0008
J :	j	0x000a
	......	
A :	A[0][0]	0x0012
	A[0][1]	0x0014
	......	
	A[2^a-1][2^a-2]	
	A[2^a-1][2^a-1]	
	......	
B :	0	0x0040
ADDR_A :	0x0012	0x0042
TMP_1 :	0	0x0044

Pseudo-code:

B = A[i][j] + A[j][i]

Step 1: Load A[i][j]

```
      CLEAR
      ADD     I
      SHIFTL  a
      ADD     J
      ADD     ADDR_A
      ADD     _add_op
      STORE   _L1
      CLEAR
_L1:  CLEAR   ADD (0x0012+2^a*i+j)
```

(CLEAR at _L1 is struck through and replaced by ADD (0x0012+2^a*i+j))

A[i][j]

Accumulator

Self Modifying Code Example

Memory

	......	0x0000
I :	i	0x0008
J :	j	0x000a
	......	
A :	A[0][0]	0x0012
	A[0][1]	0x0014
	......	
	A[2^a-1][2^a-2]	
	A[2^a-1][2^a-1]	
	......	
B :	0	0x0040
ADDR_A :	0x0012	0x0042
TMP_1 :	**A[i][j]**	0x0044

Pseudo-code:

B = A[i][j] + A[j][i]

Step 1: Load A[i][j]

```
      CLEAR
      ADD     I
      SHIFTL  a
      ADD     J
      ADD     ADDR_A
      ADD     _add_op
      STORE   _L1
      CLEAR
_L1:  CLEAR   ADD (0x0012+2^a*i+j)
      STORE   TMP_1
```

(CLEAR at _L1 is struck through and replaced by ADD (0x0012+2^a*i+j))

A[i][j]

Accumulator

Self Modifying Code Example

Memory

	......	0x0000
I :	i	0x0008
J :	j	0x000a
	......	
A :	A[0][0]	0x0012
	A[0][1]	0x0014
	......	
	A[2^a-1][2^a-2]	
	A[2^a-1][2^a-1]	
	......	
B :	0	0x0040
ADDR_A :	0x0012	0x0042
TMP_1 :	A[i][j]	0x0044

Pseudo-code:

B = A[i][j] + A[j][i]

Step 2: Load A[j][i]

```
     CLEAR
     ADD     J
     SHIFTL  a
     ADD     I
     ADD     ADDR_A
     ADD     _add_op
     STORE   _L2
     CLEAR
_L2: CLEAR
```

A[j][i]

Accumulator

Self Modifying Code Example

Memory

	Contents	Address
	......	0x0000
I :	i	0x0008
J :	j	0x000a
	......	
A :	A[0][0]	0x0012
	A[0][1]	0x0014
	......	
	A[2^a-1][2^a-2]	
	A[2^a-1][2^a-1]	
	......	
B :	A[i][j] + A[j][i]	0x0040
ADDR_A :	0x0012	0x0042
TMP_1 :	A[i][j]	0x0044

Pseudo-code:

B = A[i][j] + A[j][i]

Step 2: Load A[j][i]

```
     CLEAR
     ADD     J
     SHIFTL  a
     ADD     I
     ADD     ADDR_A
     ADD     _add_op
     STORE   _L2
     CLEAR
_L2: CLEAR
```

Step 3: Store A[i][j] + A[j][i]

```
     ADD     TMP_1
     STORE   B
```