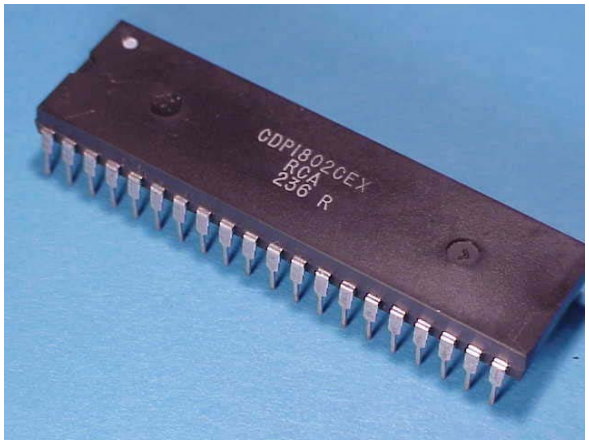
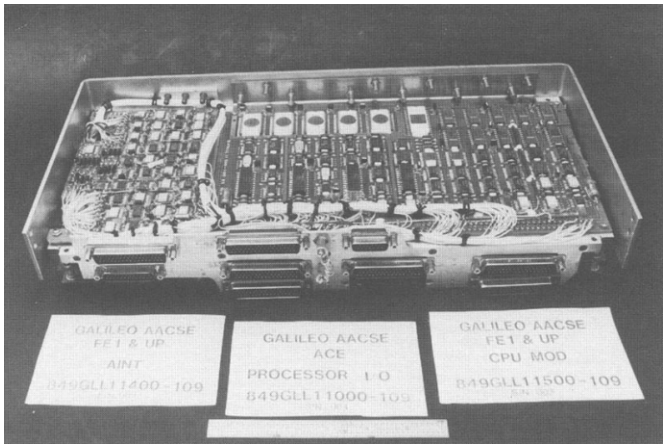
CSE/EE 322: Computer Architecture II
Spring 2008

A Case Study: The RCA 1802
(The Microprocessor that went to Jupiter)

Peter M. Kogge





References

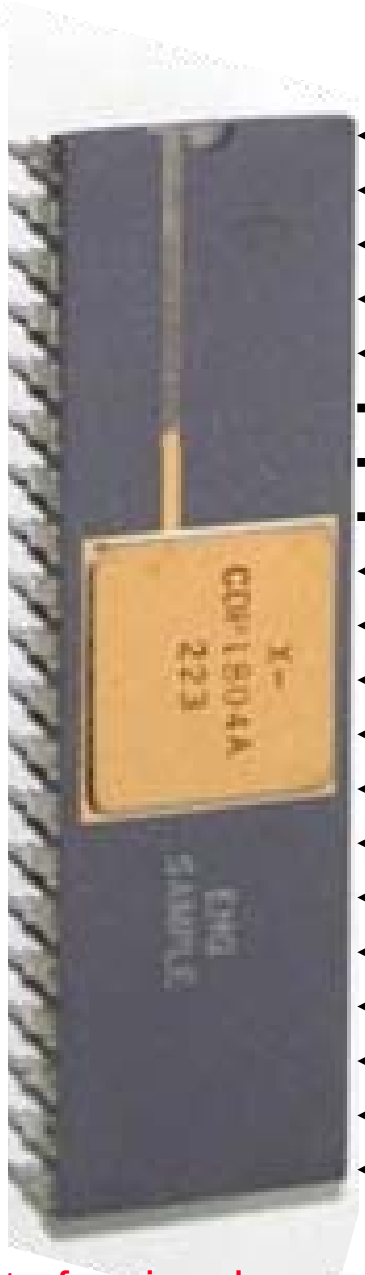
- Intersil Data Sheet: “CDP1802A, CDP 1802AC, CDP1802BC CMOS 8-bit Microprocessors”
  - http://homepage.mac.com/ruske/cosmacelf/cdp1802.pdf
- A Wikipedia summary
  - http://en.wikipedia.org/wiki/RCA_1802
- Programming the 1802
  - http://www.ittybittycomputers.com/IttyBitty/ShortCor.htm
- Die Phote of RCA 1802 chip
  - http://www.cpu-world.com/CPUs/1802/die/L_RCA-CDP1802.jpg
- The picture of the Galileo Jupiter computer board
  - http://history.nasa.gov/computers/p199.jpg

Why Is This an Interesting Microprocessor

- One of 1st microprocessors to be designed in a very regular fashion
  - With a register file at its heart
- Aspects of several different types of ISAs
- Built from Silicon-on-Sapphire (SOS)
  - Makes it inherently Radiation-Hard – good for space
- Went to Jupiter & Beyond

General Characteristics

The Chip



Blue: Memory interface
Gold: I/O
Black: Other

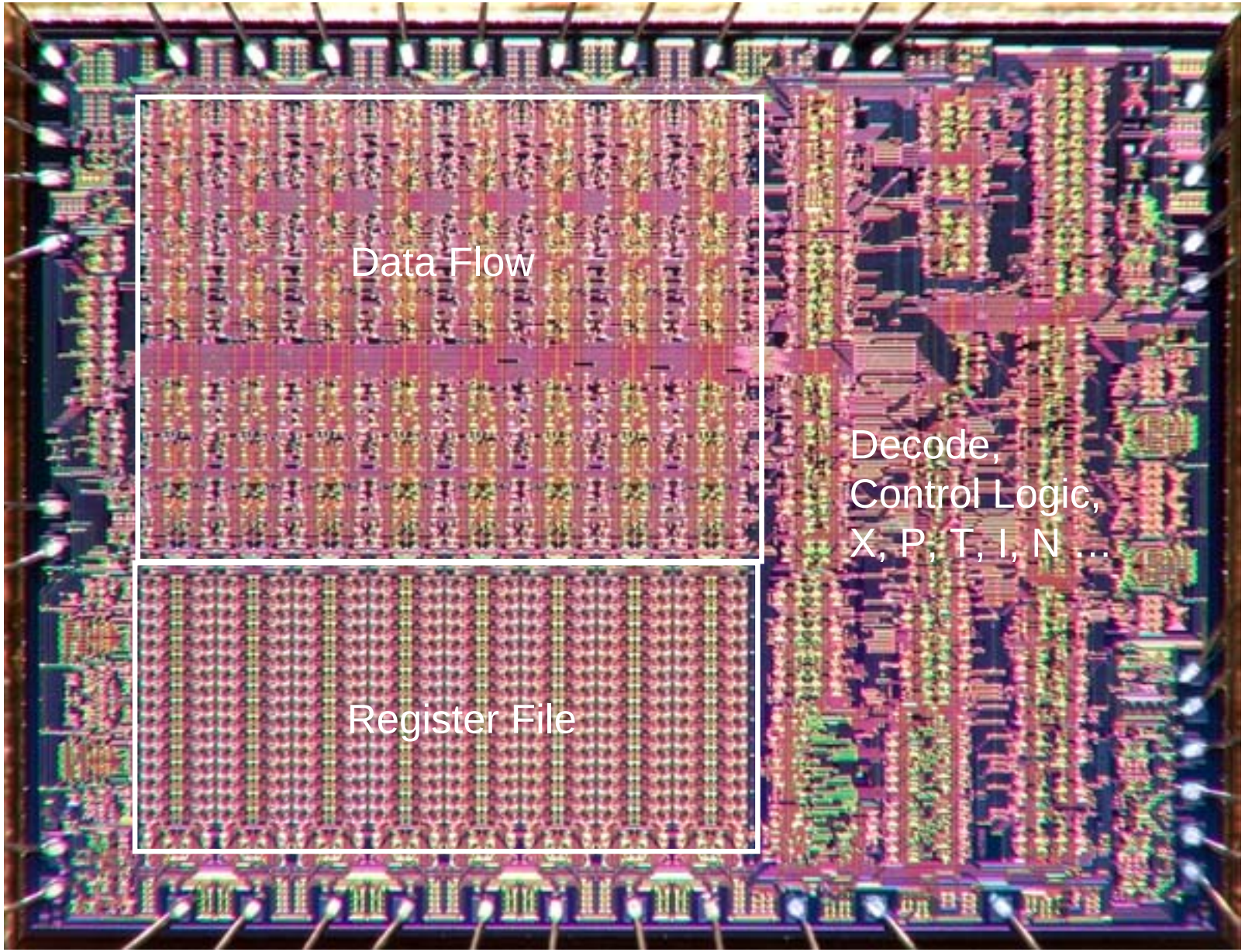
CLOCK
~WAIT
~CLEAR
Q
Current State { SC1, SC0
~Memory Read
DATA 7
DATA 6
DATA 5
DATA 4
DATA 3
DATA 2
DATA 1
DATA 0
GROUND
I/O Address { N2, N1, N0
VSS

Vdd
~CRYSTAL
~DMA-IN
~DMA-OUT
INTERRUPT
~Memory Write
Timing Pulse A
Timing Pulse B
MEMORY ADDRESS 7
MEMORY ADDRESS 6
MEMORY ADDRESS 5
MEMORY ADDRESS 4
MEMORY ADDRESS 3
MEMORY ADDRESS 2
MEMORY ADDRESS 1
MEMORY ADDRESS 0
~External Flag 1
~External Flag 2
~External Flag 3
~External Flag 4

16-bit Address Transferred In 2 8-bit Pieces

Note: a "~" in front of a signal means that it is "active low"
i.i. a "0" indicates the signal is "active"

The Die



Memory Interface

- Processor supports up to 64K words of memory
- Each word is 8 bits wide
- Memory takes16 bit address
  - But only 8-bit memory bus
  - Address sent to memory parts 8-bits at a time
    - Memory parts must “latch” first part (matches what happens in all current DRAM chips)
- Data is accessed in 1 word = 8 bit units
- Most instructions are 8 bits long; a few are either 16 or 24 bits

Key Computational Registers (There are a few others for I/O & interrupts)

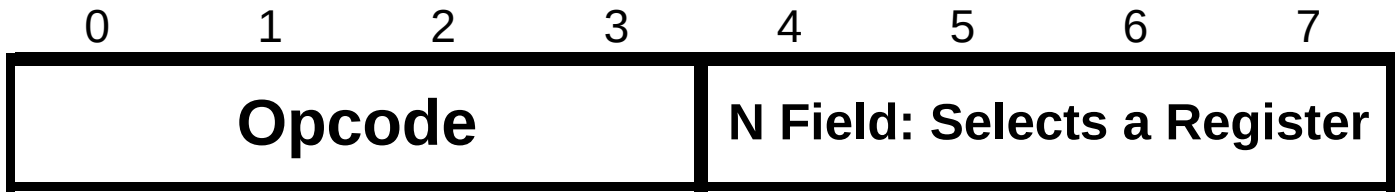
- **Register File**: 16 16-bit registers R(0) … R (15)
  - Each register is used as a pointer to hold a memory address
    - 3 registers also used for special I/O-related addresses
      - R(0) for DMA address, R(1) for interrupt PC, R(2) for interrupt X
  - 8 bit halves of each may be accessed separately
    - R(i).1 is the "high" 8 bits
    - R(i).0 is the "low" 8 bits
  - Some instructions may automatically increment or decrement the specified register in addition to the memory access
- **P register**: 4 bit register which specifies which register in the register file holds the address of the next instruction
  - I.e. which R register is the current PC
- **X register**: 4 bit register which specifies which register in register file is used in computational instructions to access memory for data
- **D register**: 8 bit register that receives the results from the ALU
- **DF**: 1 bit "Data Flag" used to capture carries, shifts, ….

Normal Instruction Execution Cycle

- "CPU cycle" equals one memory access cycle
  - Made up from 8 chip clock pulses
- Most instructions take *exactly* 2 such CPU cycles
  - "S1 EXECUTE"
    - Select register from RF that contains address & present to memory
    - Make *memory reference* for data for current instruction
  - "S0 FETCH"
    - Do required computation in data flow and update machine registers
    - Select the RF entry that currently holds the PC
    - And *simultaneously fetch* next instruction from memory
- "Long Branches" take 3 cycles because of the extra word they access from memory
- Chip has only 8 bit address bus, so 16 bit address is output in two pieces (which means memory system must "latch" lower part)
- Data flow has separate circuitry for simultaneous increment/decrement of register file entry being used for address

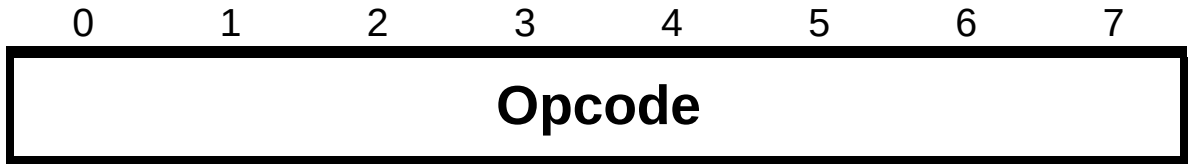
ISA Overview

Instructions With Explicit Register Specifiers



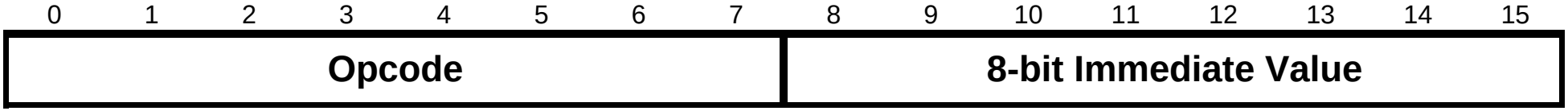
- LDN: Load via N: D=M[R(N)]
- LDN: Load via N: D=M[R(N)] and R(N)++
- STN: Store via N: M[R(N)]=D
- INC: Increment Register N: R(N)++
- DEC: Decrement Register N: R(N)–
- GLO: Get Low: D=R(N).0
- GHI: Get High: D=R(N).1
- PLO: Put Low: R(N).0=D
- PHI: Put High: R(N).1=D
- SEX: Set X: X=N
- SEP: Set P: P=N

Instructions that Use X, R(X), and Memory[R(X)]



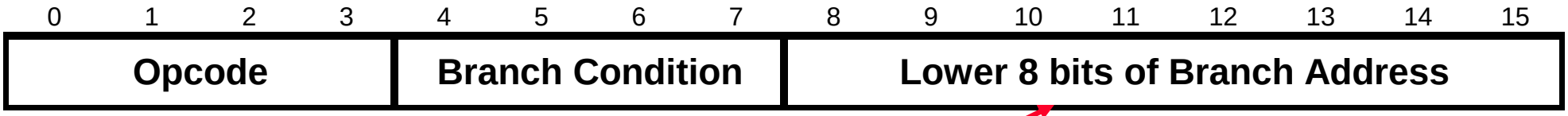
- LDX: Load via X: D=M[R(X)]
- LDXA: Load via X and Advance: D=M[R(X)] and R(X)++
- STDX: Store via X and Decrement: M[R(X)]=D and R(X)—
- IRX: Increment Register X: R(X)++
- OR: D=M[R(X)] || D
- AND: D=M[R(X)] && D
- XOR: D=M[R(X)] ^ D
- ADD: D=M[R(X)] + D
- SD: Subtract D: D=M[R(X)] – D
- SM: Subtract Memory: D=D - M[R(X)]
- Others: Shift lefts and rights of various forms, adds , subs with carries

Instructions With 8-bit Immediate Constants



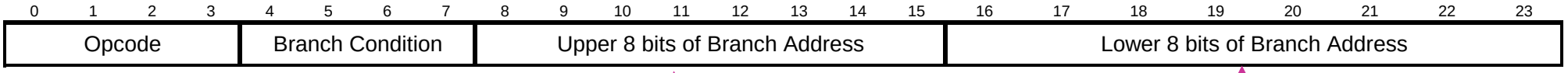
- Note: During 1st cycle of these, R(P) points to Immediate word
- LDI: Load Immediate: D=M[R(P)] and R(P)++
- ORI : Or Immediate: D=M[R(P)] ||D
- Likewise for AND, XOR, ADD, and Subtract Immediate
  - Options for Add and Subtract to use carry/borrow for extended arithmetic
  - Subtract has two forms: D-Immediate and Immediate-D

Short Branches



- Note: During Execute cycle of these, R(P) points to Branch Address
- Branch Condition field specifies some condition to test
- If Condition is True, then R(P).0=M[R(P))
  - i.e. *replace* lower 8 bits of current PC with the constant from instruction
- If Condition is False, then R(P)++
  - i.e. increment PC to next word after this instruction

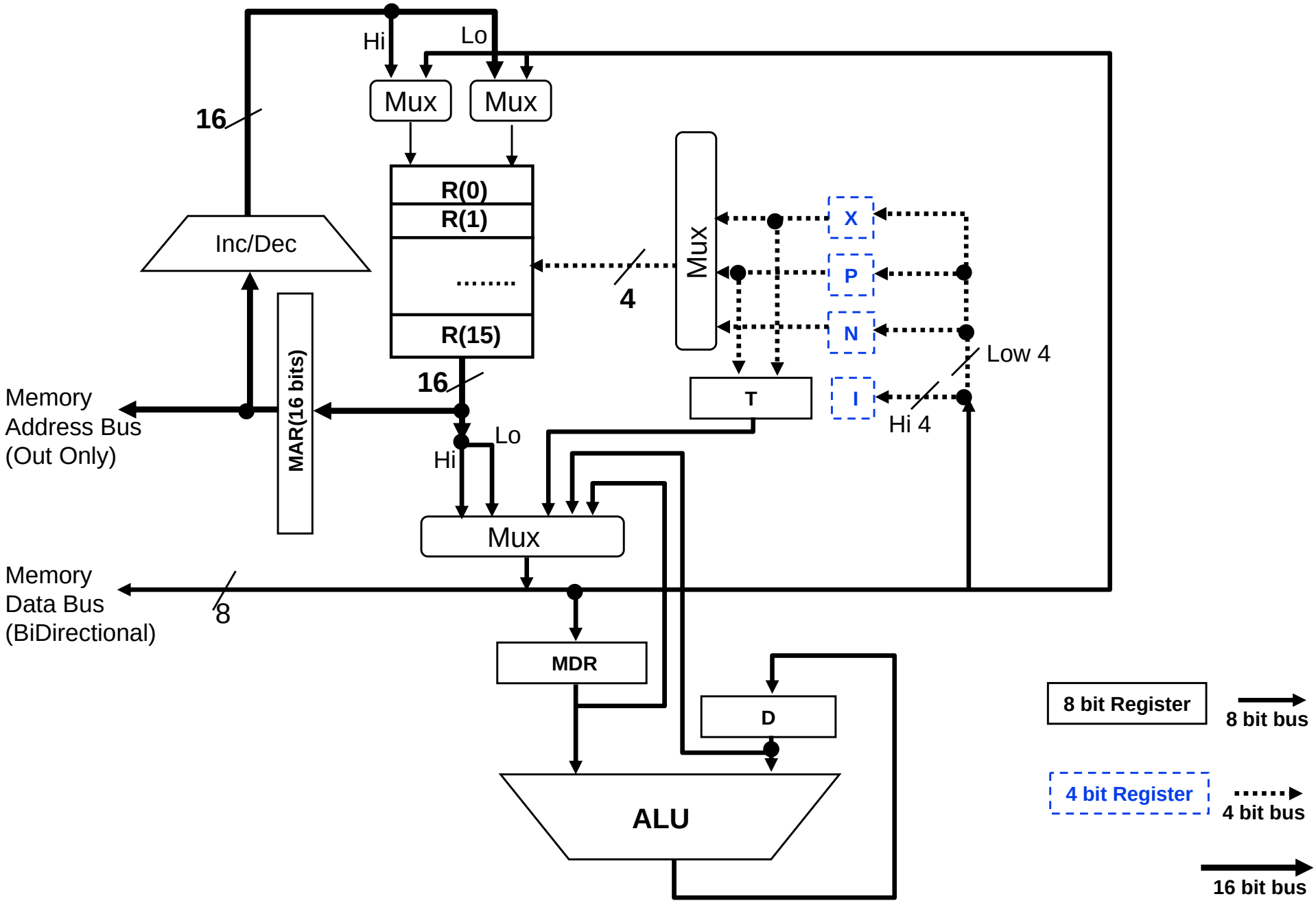
Long Branches



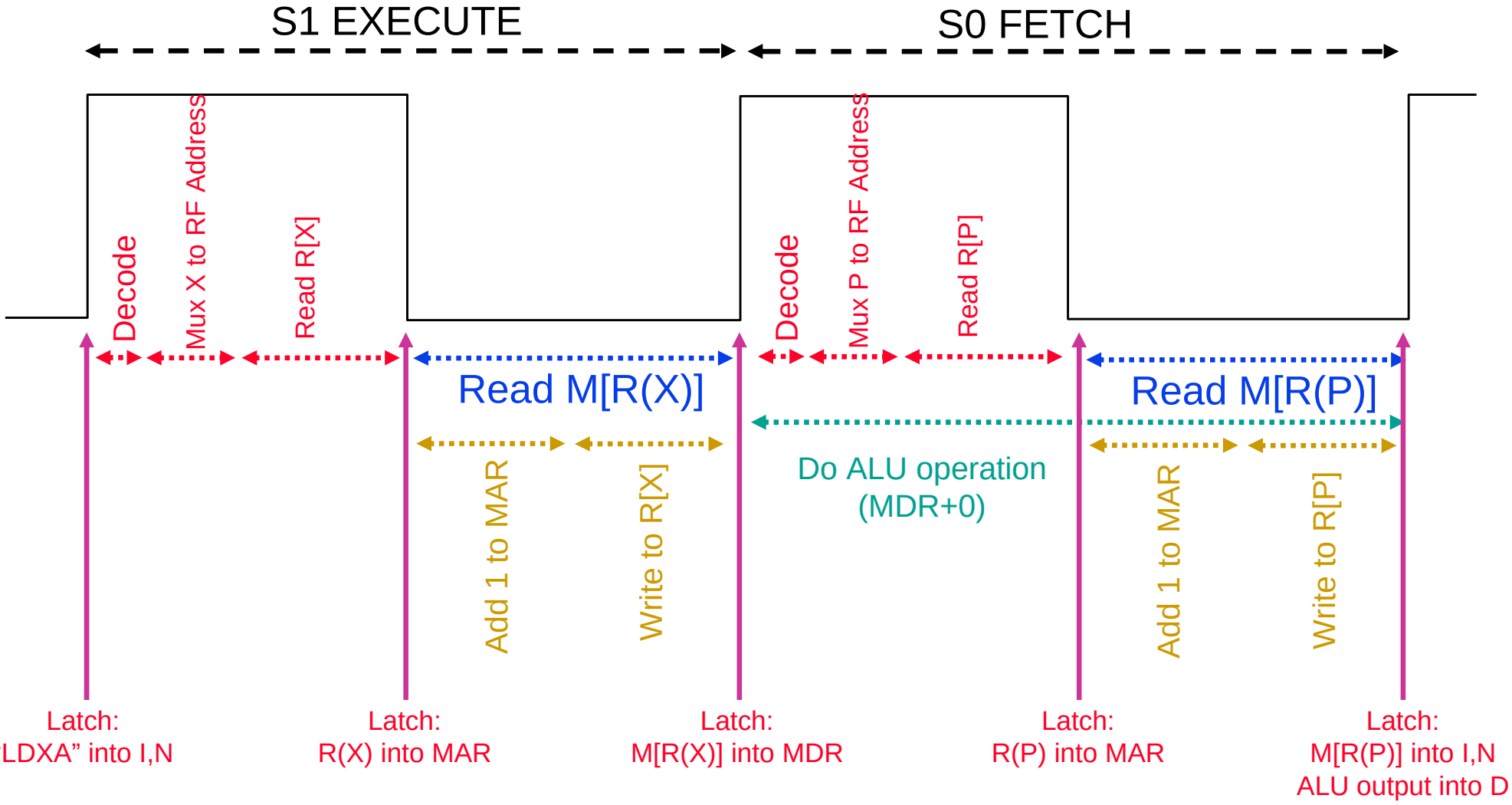
- 3 cycle execution
- During 1st execution cycle, R(P) points to upper byte of Branch Address
  - And R(P) is simultaneously incremented
- During 2nd cycle, R(P) points to lower byte of Branch Address
  - And R(P) is simultaneously incremented again
- Branch Condition field specifies some condition to test
- If Condition is True, then R(P).1=M[R(P)] and R(P).0=M[R(P)+1]
  - i.e. replace all of current PC with the 16 bit constant from instruction
- If Condition is False, then R(P)+=2
  - i.e. increment PC to next word after this instruction

Data Flow and Timing

A "Simplified" Data Flow



Example: Machine Cycles for LDXA



External Chip Control Signals

- CLEAR, WAIT: two control signals from outside of chip to control program's execution

~CLEAR	~WAIT	Function
Low	Low	LOAD: Hold CPU in IDLE state with address & data buses disabled; External device can then load memory
Low	High	RESET: I=N=Q=X=P=R(0)=0; IE=1;
High	Low	PAUSE: Stop the clocks
High	High	RUN: Let the clocks run

Input/Output Considerations

I/O Instructions

- **Q**: 1 bit register whose output comes off the chip
  - SEQ: Set Q: Q=1
  - REQ: Reset Q: Q=0
  - Various Branch conditions test Q
- **EF1** to **EF4**: 4 1-bit “External Flags”
  - Inputs to chip from outside devices
  - Testable by Short Branch conditions
- **N1-N3**: 3 “I/O Address” Lines that leave CPU
- **OUT** # (1-7): output data
  - Place # on N1-N3
  - Signal Output
  - Place D on chip data bus
- **IN** # (1-7): input data
  - Place # on N1-N3
  - Signal Input
  - Copy data bus into D

Interrupts

- One interrupt line from off-chip
- **IE**: 1-bit register “Interrupt Enable”
- **T**: 8-bit register for Interrupt data
- If signal on interrupt line and IE=1 then take interrupt
  - Save (X,P) into T
  - X=2
  - P=1
  - IE=0 (disabled)
- Interrupt instructions
  - SAV: Save T: M[R(X)]=T
  - RET: Return and enable: (X,P)=M[R(X)] and R(X)++ and IE=1
  - DIS: Return and disable: (X,P)=M[R(X)] and R(X)++ and IE=0
  - MARK: Push X,P to Stack: T=(X,P) and M[R(2)]=(X,P); then X=P and R(2)--

DMA Capability

- 2 input signals from offside chip
  - **~DMA-IN**: signal external device wants to do a DMA to memory
  - **~DMA-OUT**: signal external device wants to do a DMA from memory
- If either is active, then at end of next instruction
  - Place R[0] onto address bus
  - If DMA-IN active, write what data is on data bus (from device) to memory
  - IF DMA-OUT active, read memory and allow device to sample memory output

Programming

Accessing a Variable in Memory

- All operand addresses must be in a register
  - Assume here all objects are 8 bits
- If address of Z is in register “k”, then
  - To load Z into D use `LDN k`
  - To store Z from D use `STR k`
- To place a known address into register k
  - `LDI low_byte_of_Z`
  - `PLO k`
  - `LDI hi_byte_of_Z`
  - `PHI k`
- If register k has an address that is 1 away from desired one, then use `INC k` or `DEC k` to compute address
  - Great for arrays or data streams

Computation with variables in memory (The "X" Factor)

- Address in register designated by X register has special properties
  - Instruction SEX k sets X to point to register k
- X need not be specified in loads (LDX) or stores (STX)
- Can be used to perform operation with D
  - As in ADD that performs D = M[R(X)} + D
- On loads, R(X) can be optionally incremented after use
  - LDXA
- On stores, R(X) can be optionally decremented after use
  - STXD
- Additional instructions to increment R(X): IRX

Multi-byte Integers

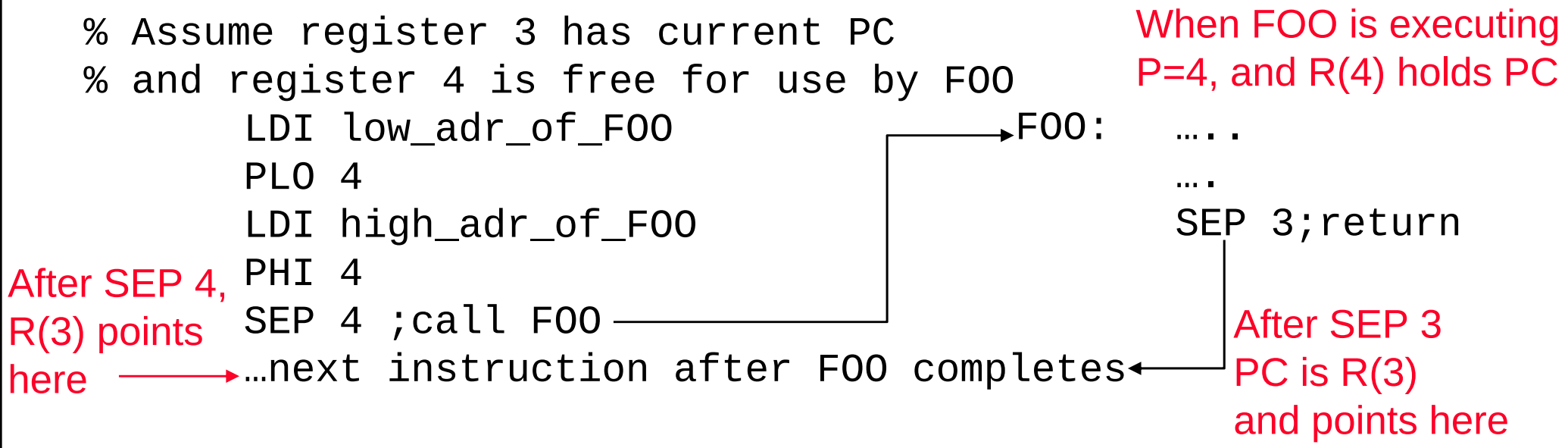
- Dealing with multi-byte integers requires multiple instructions and use of DF flag for byte-to-byte carries
- Example: assume we want to add two 32-bit (4 byte) integers A and B, and place result in C
  - All variables in “Little Endian” format
    - Lowest address is to least significant byte
  - Address of least significant byte of A in R(3)
  - Address of least significant byte of B in R(X) (whatever X is)
  - Address of least significant byte of C in R(4)

```
LDN 3; get 1st byte of A        LDN 3    LDN 3    LDN 3
ADD; Add on 2nd byte of B       ADC      ADC      ADC
STN 4; store in 1st byte of C   STN 4    STN 4    STN 4
INC 3; adjust address of A      INC 3    INC 3
IRX; adjust address of X        IRX      IRX
INC 4; adjust address of C      INC 4    INC 4
```

ADC does add including carry from last add

Procedure Calls

- All addresses, including program addresses, are in registers
  - P points to register holding current PC
- To call a procedure:
  - Must have address of that procedure in some register
  - Must be able to switch the P to point to that new register
  - Must be able to switch back at end of procedure



Self-Resetting Procedure Calls

- Assume procedure FOO is called a lot
- It gets annoying to continually reload R(4)
- Thus dedicate R(4) to FOO, and load it once at beginning
- Rewrite FOO as

```
FOOEXIT SEP 3; return, leaving R(4) at start of FOO
FOO:    …
        …
        BR FOOEXIT
```

Even Neater: Co-Routines

- Assume FOO and FUM mutually call each other
  - As is typical in many real-rime consumer-producer applications
  - I.e. each is a separate “Thread”
- Procedure FUM uses R(3), and Procedure FOO uses R(4)

```
FUM:    …
        …; generate 1st data for FOO
        SEP 4; start FOO
        …; generate next data for FOO
        SEP 4; restart FOO
        …

FOO:    …; process 1st data from FUM
        SEP 3; return to FUM
        …; process next data from FUM
        SEP 3; return to FUM
        …
```

What about X Register?

- Each procedure might want to manage its own X register
  - Especially with co-routines or interrupt handlers
- SEX k allows us to “reset” X, but how to “remember” caller’s X?
- Answer: (Useful in interrupt routines)
  - MARK saves both X and P to memory pointed to by R(2)
    - And then decrements R(2)
  - Then if X=2, RETURN reloads X and P from memory
    - With side effect of “enabling interrupts”
  - DISABLE does same thing but “disables interrupts”