# 18-447 Lecture 9: Microcontrolled Multi-Cycle Implementations

James C. Hoe
Dept of ECE, CMU
February 18, 2009

Announcements: Read P&H Appendix D
Get started on Lab

Handouts: Handout #8: Project 1 (on Blackboard)



## Single-Cycle Implementations

- Matches naturally with the sequential and atomic semantics inherent to most ISAs
  - instantiate programmer-visible state one-for-one
  - map instructions to combinational next-state logic
- But, contrived and inefficient
  - all instructions run as slow as the slowest instruction
  - must provide worst-case combinational resource in parallel as required by any instruction
- Not necessarily the simplest way to implement an ISA

Imagine trying to build a single-cycle implementation of a CISC ISA

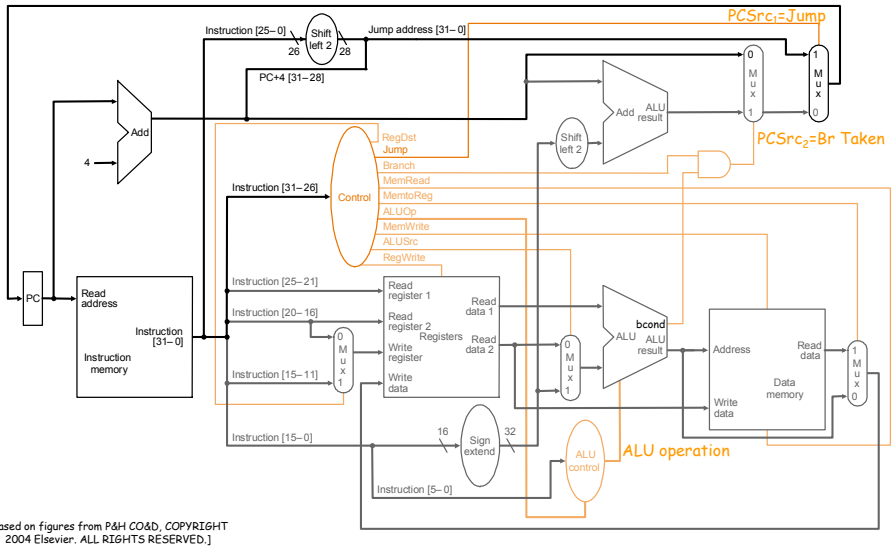
# Single-Cycle Datapath Analysis

- Assume
  - memory units (read or write): 200 ps
  - ALU and adders: 100 ps
  - register file (read or write): 50 ps
  - other combinational logic: 0 ps

| steps | IF | ID | EX | MEM | WB | Delay |
|---|---|---|---|---|---|---|
| resources | mem | RF | ALU | mem | RF | |
| R-type | 200 | 50 | 100 | | 50 | 400 |
| I-type | 200 | 50 | 100 | | 50 | 400 |
| LW | 200 | 50 | 100 | 200 | 50 | 600 |
| SW | 200 | 50 | 100 | 200 | | 550 |
| Branch | 200 | 50 | 100 | | | 350 |
| Jump | 200 | | | | | 200 |



# Worksheet



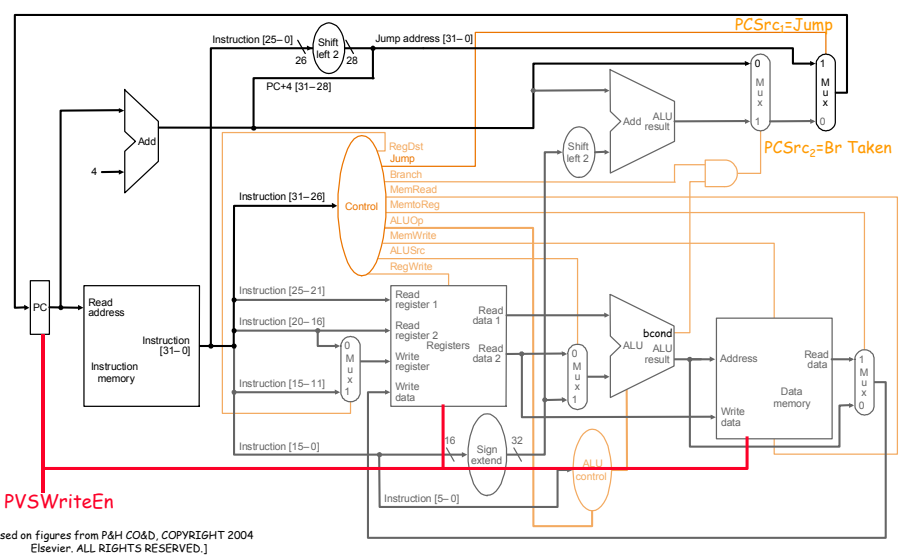
# Multi-cycle Implementation: Ver 1.0

- Why not let each instruction type take only as much time as it needs
- Idea
  - run a 50 ps clock
  - let each instruction type take as many clock cycles as needed
  - programmer-visible state only updates at the end of an instruction's cycle-sequence
  - an instruction's effect is still purely combinational from PVS to PVS

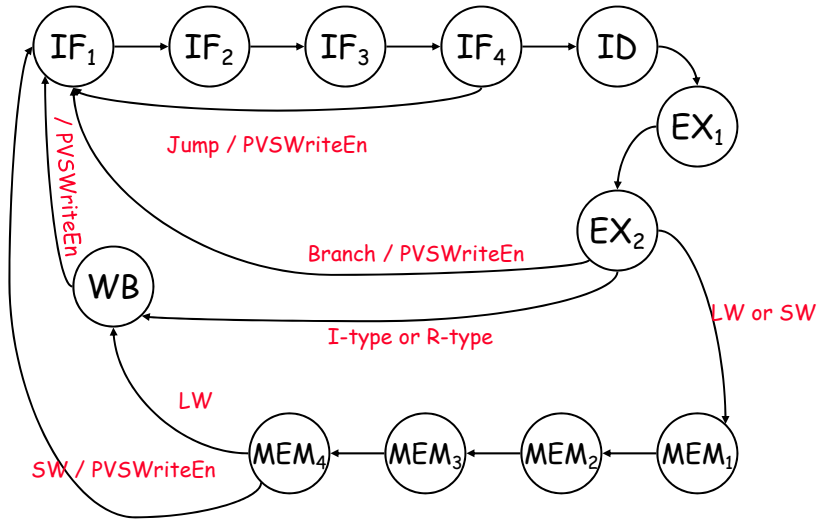
A more realistic alternative to the "variable-length" clock design in the textbook



# Multi-Cycle Datapath: Ver 1.0



# Sequential Control: Ver 1.0




What about the rest of the control signals?



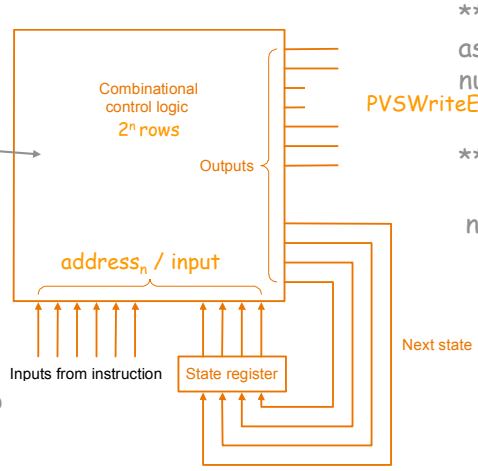
# MicroSequencer: Ver 1.0

- ROM as a combinational logic lookup table

** ROM size grows as $O(2^n)$ as the number of inputs

** ROM size grows as $O(m)$ as the number of outputs

literally holds the truth table




Mealy or Moore?

# Microcoding: Ver 1.0

| state label | cntrl flow | conditional targets | | | | |
|---|---|---|---|---|---|---|
| | | R/I-type | LW | SW | Br | Jump |
| $IF_1$ | next | - | - | - | - | - |
| $IF_2$ | next | - | - | - | - | - |
| $IF_3$ | next | - | - | - | - | - |
| $IF_4$ | branch | ID | ID | ID | ID | $IF_1$ |
| ID | next | - | - | - | - | - |
| $EX_1$ | next | - | - | - | - | - |
| $EX_2$ | branch | WB | $MEM_1$ | $MEM_1$ | $IF_1$ | - |
| $MEM_1$ | next | - | - | - | - | - |
| $MEM_2$ | next | - | - | - | - | - |
| $MEM_3$ | next | - | - | - | - | - |
| $MEM_4$ | next | - | WB | $IF_1$ | - | - |
| WB | branch | $IF_1$ | $IF_1$ | $IF_1$ | $IF_1$ | $IF_1$ |

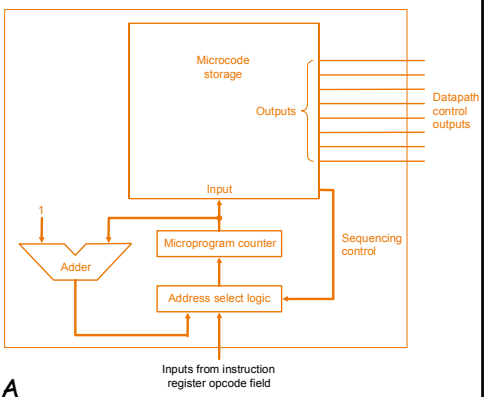
A systematic approach to FSM sequencing/control



# Microcontroller

- A stripped-down processor for sequencing and control
  - control states are like µPC
  - µPC indexed into a µprogram ROM to select an µinstruction
  - well-formed control-flow architecture
  - fields in the µinstruction maps to control signals

  Why not also support µprogram-visible states and ALU ops?
- Very elaborate mcontrollers have been built
  - some µcontrollers had full-fledged ISAs
  - µISAs for CISC machines in many ways inspired RISC ISA




[Based on figures from P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

# Performance Analysis

- $T_{wall\text{-}clock}$ = No.Instructions × CPI × $T_{clk}$
  - "No.Instruction" is fixed for a given ISA and application mix
- For a fixed ISA and application mix it is meaningful to compare

$$T_{avg\text{-}inst} = CPI \times T_{clk}$$

or

$$MIPS = IPC \times f_{clk}$$

frequency in MHz

million instructions per second

instructions per cycle



# Performance Analysis

- Single-Cycle Implementation

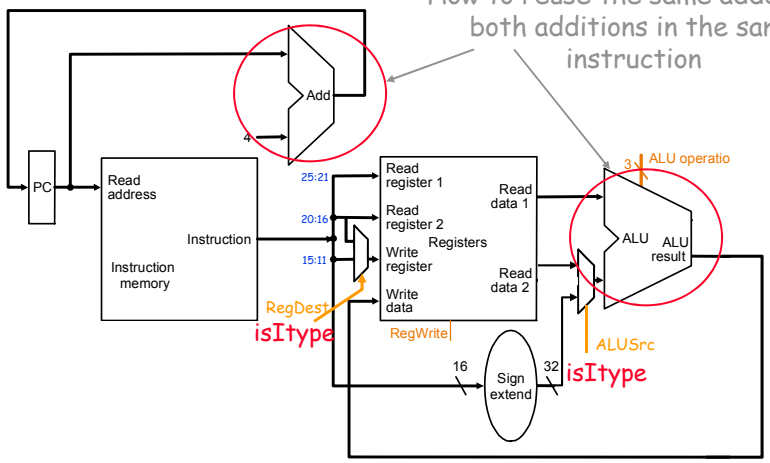
  1 × 1,667MHz = 1667 MIPS
- Multi-Cycle Implementation

  $IPC_{avg}$ × 20,000 MHz = 2235 MIPS

  what is average behavior?
- Assume: 25% LW, 10% SW, 45% ALU, 15% Branch and 5% Jumps
  - weighted arithmetic mean of CPI ⇒ 8.95
  - weighted harmonic mean of IPC ⇒ 0.1117

# Reducing Datapath by Resource Reuse

How to reuse the same adder for both additions in the same instruction

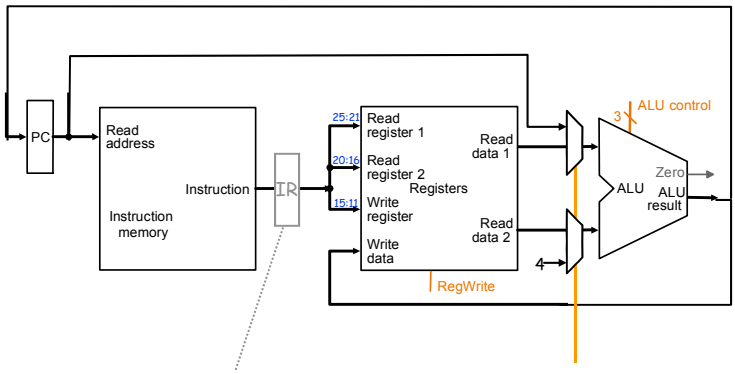



Previous example of reuse by mutually exclusive conditions



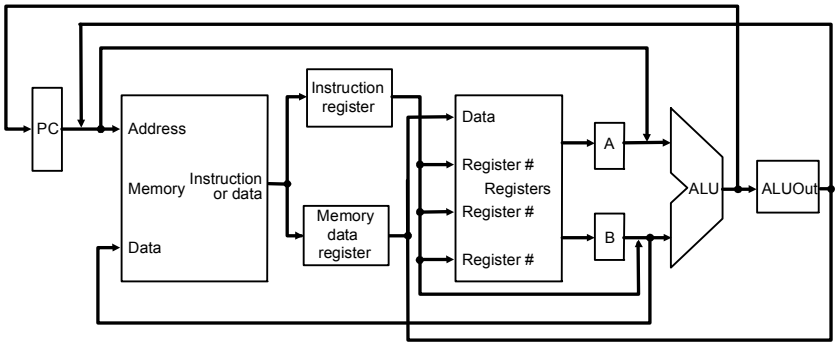


# Reducing Datapath by Sequential Reuse




to IR or not to IR?

# Removing Redundancies

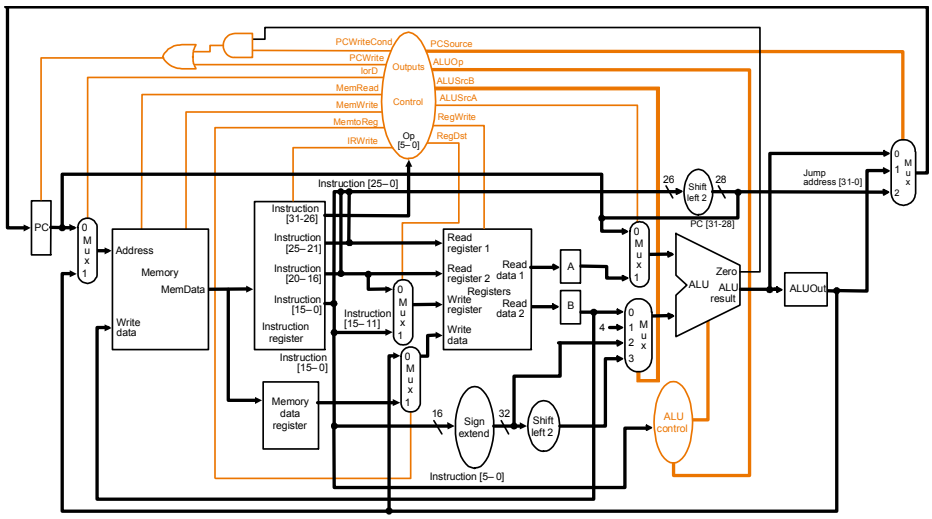



- Could also reduce down to a single register read-write port!
- Do we want to remove all redundancies?





# Control Points



# New Sequential Control Signals

| | When De-asserted | When asserted |
|---|---|---|
| ALUSrcA | 1st ALU input from **PC** | 1st ALU input from 1st **RF** read port (latched in **A**) |
| IorD | **PC** supplies memory address | **ALUOut** supplies memory address |
| IRWrite | **IR** latching disabled | **IR** latching enabled |
| PCWrite | no effect | **PC** latching enabled unconditionally |
| PCWriteCond | no effect | **PC** latching enabled only if branch condition is satisfied |

*When both PCWrite and PCWriteCond are de-asserted, **PC** latching is disabled*



# New Sequential Control Signals

| signal | | effect |
|---|---|---|
| ALUSrcB[1:0] | 00 | 2nd ALU input from 2nd **RF** read port (latched in **B**) |
| | 01 | 2nd ALU input is 4 (for PC increment) |
| | 10 | 2nd ALU input is sign-extended **"IR[ 15:0 ]"** |
| | 11 | 2nd ALU input is sign-extended **"IR[ 15:0 ],00"** |
| PCSource[1:0] | 00 | next **PC** from ALU |
| | 01 | next **PC** from **ALUOut** |
| | 10 | next PC from **IR** (jump target) |

# Old Control Signals (similar to single-cycle)

| | When De-asserted | When asserted |
|---|---|---|
| RegDest | RF write select according to IR[20:16] | RF write select according to IR[15:11] |
| RegWrite | RF write disabled | RF write enabled |
| MemRead | Memory read disabled | Memory read port return load value |
| MemWrite | Memory write disabled | Memory write enabled |
| MemtoReg | Steer ALU result (latched in ALUOut) to RF write port | steer memory load result (latched in MDR) to RF write port |



# Synchronous Register Transfers

- Synchronous state with latch enables
  - PC, IR, RF, MEM
- Synchronous state that always latch
  - A, B, ALUOut, MDR
- One can enumerate all possible combinational "register transfers" in the datapath
- For example starting from PC
  - IR ⇐ MEM[ PC ]
    requires IRWrite=1, IorD=0, MemRead=1
  - PC ⇐ PC,JumpTarget
    requires PCWrite=1, PCSource=2'b10
  - PC ⇐ PC+ALUSrcB
  - MDR ← MEM[ PC ]
  - ALUOut ← PC + ALUSrcB
    ……

# Useful Register Transfers

- PC ⇐ PC+4
- PC ⇐ ALUOut (if PCWriteCond is asserted)
- PC ⇐ PC[ 31:28 ],IR[ 25:0 ],2'b00
- IR ⇐ MEM[ PC ]
- A ← RF[ IR[ 25:21 ] ]
- B ← RF[ IR[ 20:16 ] ]
- ALUOut ← A + B
- ALUOut ← A + sign-extend (IR[ 15:0 ])
- ALUOut ← PC + ( sign-extend (IR[ 15:0 ]) <<2 )
- MDR ← MEM[ ALUOut ]
- MEM[ ALUOut ] ⇐ B
- RF[ IR[ 15:11 ] ]⇐ALUOut,
- RF[ IR[ 20:16 ] ]⇐ALUOut
- RF[ IR[ 20:16 ] ] ⇐ MDR

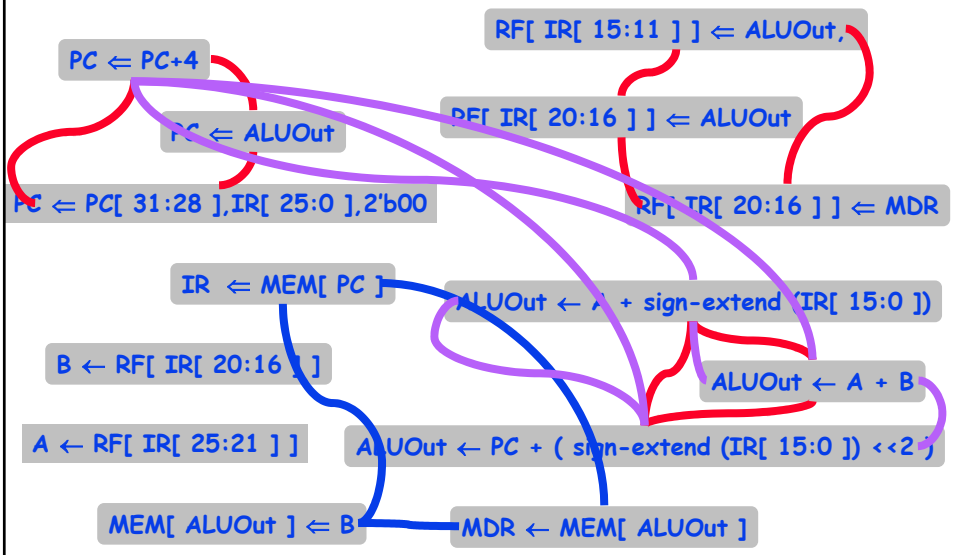


# RT Sequencing: R-Type ALU

- IF
  - IR ⇐ MEM[ PC ]
  - PC ⇐ PC+4
- ID
  - A ← RF[ IR[ 25:21 ] ]
  - B ← RF[ IR[ 20:16 ] ]
- EX
  - ALUOut ← A + B
- MEM
  -
- WB
  - RF[ IR[ 15:11 ] ] ⇐ ALUOut

```
if MEM[PC] == ADD rd rs rt
        GPR[rd] ← GPR[rs] + GPR[rt]
        PC ← PC + 4
```

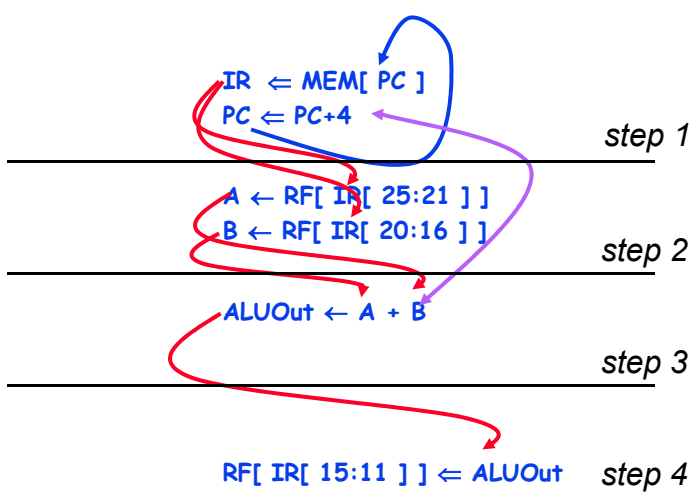
# RT Datapath Conflicts




Can utilize each resource only once per control step (cycle)



# RT Sequencing: R-Type ALU



# RT Sequencing: LW

- IF
  - IR ⇐ MEM[ PC ]
  - PC ⇐ PC+4
- ID
  - A ← RF[ IR[ 25:21 ] ]
  - B ← RF[ IR[ 20:16 ] ]
- EX
  - ALUOut ← A + sign-extend (IR[ 15:0 ])
- MEM
  - MDR ← MEM[ ALUOut ]
- WB
  - RF[ IR[ 20:16 ] ] ⇐ MDR

```
if MEM[PC]==LW rt offset16 (base)
    EA = sign-extend(offset) + GPR[base]
    GPR[rt] ← MEM[ translate(EA) ]
    PC ← PC + 4
```



# RT Sequencing: Branch

- IF
  - IR ⇐ MEM[ PC ]
  - PC ⇐ PC+4
- ID
  - A ← RF[ IR[ 25:21 ] ]
  - B ← RF[ IR[ 20:16 ] ]
  - ALUOut←PC+( sign-extend (IR[ 15:0 ])<<2 )
- EX
  - PC ⇐ ALUOut (only if condition is met)
- MEM
- WB

```
if MEM[PC]==BEQ rs rt immediate16
        target = PC + sign-extend(immediate) x 4 +4
        if GPR[rs]==GPR[rt] then    PC ← target
                            else    PC ← PC + 4
```

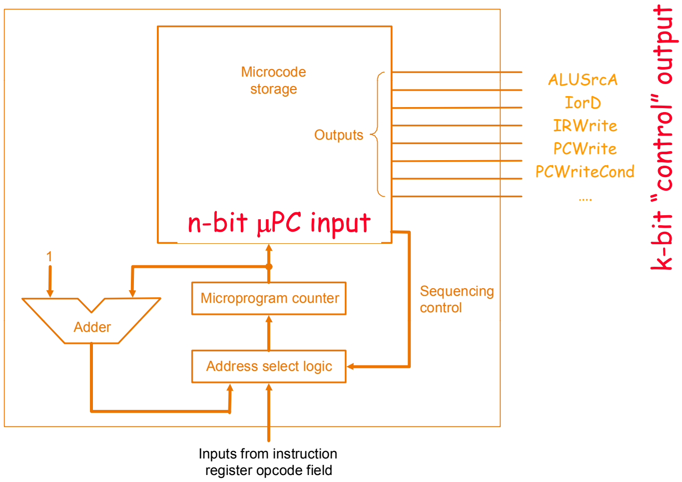
# Combined RT Sequencing

| | R-Type | LW | SW | Branch | Jump |
|---|---|---|---|---|---|
| common steps | **start:** IR ⇐ MEM[ PC ]<br>PC ⇐ PC+4<br>next | | | | |
| | A ← RF[ IR[ 25:21 ] ]<br>B ← RF[ IR[ 20:16 ] ]<br>ALUOut ← PC + ( sign-extend (IR[ 15:0 ]) <<2 )<br>case opcode | | | | |
| opcode dependent steps | ALUOut ← A + B<br>next | ALUOut ← A + sign-extend (IR[ 15:0 ])<br>next | ALUOut ← A + sign-extend (IR[ 15:0 ])<br>next | PC ⇐ ALUOut<br>goto start | PC ⇐ PC[31:28], IR[25:0], 2<br>goto start |
| | RF[ IR[15:11] ] ⇐ ALUOut<br>goto start | MDR ← MEM[ALUOut]<br>next | MEM[ALUOut] ← B<br>goto start | | |
| | | RF[IR[15:11]] ⇐ MDR<br>goto start | | | |

RTs in each state corresponds to some setting of the control signals

---

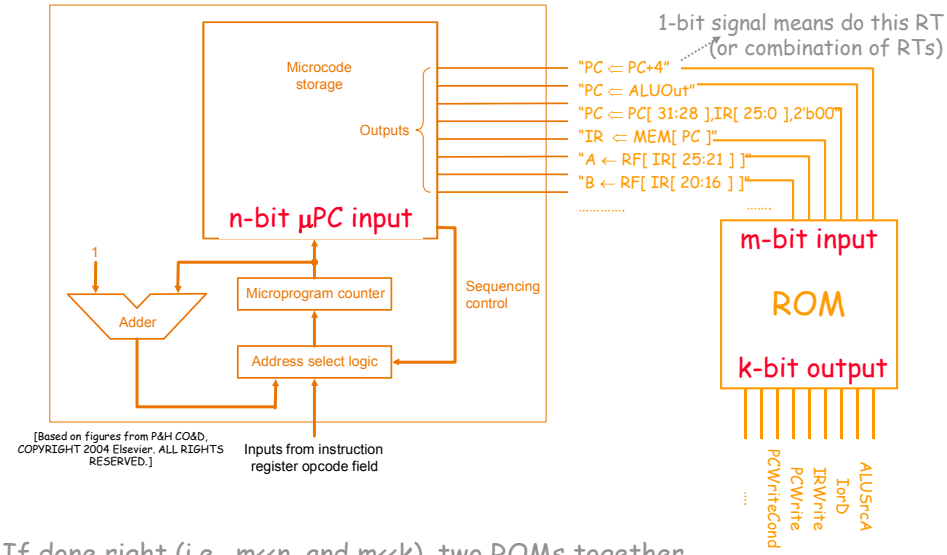
# Horizontal Microcode



[Based on figures from P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

Control Store: $2^n \times k$ bit (not including sequencing)

# Vertical Microcode




[Based on figures from P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

If done right (i.e., m<<n, and m<<k), two ROMs together ($2^n \times m + 2^m \times k$ bit) should be smaller than horizontal microcode ROM ($2^n \times k$ bit)



# Microcoding for CISC

- Can we extend the μcontroller and datapath
  - to support a new instruction I haven't thought of yet
  - to support a complex instruction, e.g. polyf
- Yes, and probably more
  - if I can sequence an arbitrary RISC instruction then I can sequence an arbitrary "RISC program" as a μprogram sequence
  - will need some μISA state (e.g. loop counters) for more elaborate μprograms
  - more elaborate μISA features also make life easier
- μcoding allows very simple datapath do very powerful computation
  - a datapath as simple as a Turning machine is universal
  - μcode enables a minimal datapath to emulate any ISA you like (with a commensurate slow down)

# Nanocode and Millicode

- Nanocode
  - a level below µcode
  - µprogrammed control for sub-systems (e.g., a complicated floating-point module) that acts as a slave in a µcontrolled datapath
  - e.g., the polyf sequence may be generated by a separate nanocontroller in the FPU
- Millicode
  - a level above µcode
  - ISA-level subroutines hardcoded into a ROM that can be called by the µcontroller to handle really complicated operations

    e.g., to add polyf to MIPS, one may code up polyf as a software routine that is called by the µcontroller when the polyf opcode is decoded
- In both cases, we avoid complicating the main µcontroller with polyf support

The power of abstractions!!



# Nanocode Concept Illustrated